Chandler Stevens

CSC 3430: Algorithm Design and Analysis

Dr. Arias

Wednesday, January 29th, 2020

Stable Matching

I implemented the Gale-Shapley Stable Matching Algorithm in Python 3. The source code is appended to the end of this file as the Appendix. The time complexity for my algorithm is indeed an order of O(n2), which required the usage of several maps (known as dictionaries in Python 3) since maps have a constant time lookup of O(1). I decided to generalize the problem from matching pairs of people to simply matching the preferences of any two sets that each have n elements.

I coded a main function with several helper functions that provide a console user interface for interactivity with the primary algorithm. These helper functions include user input validation, elements and preferences gathering, possible stable matching display, and empiric time measurement. Utilizing some of these helper functions, it is possible to bypass the console interface and directly call the functions with hardcoded datasets as commented in examples provided at the end of the Appendix.

As expected under the best case in which no elements have conflicting top preferences, testing datasets from n of 2 up to n of 5 resulted in an almost perfectly linear time complexity at O(n) as shown in Figure 1. Also as expected with the worst case in which there is a complete conflict of all preferences, testing the same datasets resulted in a clearly exponential time complexity that approximates to O(n2).

Figure 1. Empiric time measurement in microseconds from n of 2 up to n of 5

Exercise 1.8

Can a man or a woman end up better off by lying about his or her preferences?

Give an example of a set of preference lists for which there is a switch that would improve the partner of a woman who switched preferences.

As shown in Table 1, Rey has a list of her actual preferences, but she also provides a false list in which she claims that she prefers Poe as her second choice instead of Finn. She does this in the hopes of ultimately being able to end up with Ben.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Finn | Ben | Poe | Rose | Zorii | Rey (True) | Rey (False) |
| Rey | Rose | Rey | Finn | Finn | Ben | Ben |
| Rose | Rey | Rose | Ben | Ben | Finn | Poe |
| Zorii | Zorii | Zorii | Poe | Poe | Poe | Finn |

Table 1. List of men's preferences and women's preferences

To setup a test case to compare with, a possible stable matching using the true preferences of Rey must be determined first:

1. Finn proposes to Rey, who accepts
2. Ben proposes to Rose, who accepts
3. Poe proposes to Rey, who rejects
4. Poe proposes to Rose, who rejects
5. Poe proposes to Zorii, who accepts

Therefore, Rey ends up with Finn, but would more so prefer being with Ben.

Now, a possible stable matching using the false preferences of Rey can be determined:

1. Finn proposes to Rey, who accepts
2. Ben proposes to Rose, who accepts
3. Poe proposes to Rey, who accepts
4. Finn proposes to Rose, who accepts
5. Ben proposes to Rey, who accepts
6. Poe proposes to Zorii, who accepts

Therefore, Rey ends up better off by lying since she is now with her true love Ben.

Therefore, it is possible for an individual to compromise truthfulness of preferences to achieve a more desirable outcome.

Appendix

# Chandler Stevens  
# CSC 3430: Algorithm Design and Analysis  
# Python 3 solution for stable matching problem  
# implemented with the Gale-Shapley Stable Matching Algorithm O(n^2)  
  
# Import functions from timeit module for performance testing  
**from** timeit **import** timeit, repeat  
  
  
# Purpose: Get a valid element name from the user  
# Parameters: (string) message representing prompt to user  
# (list of strings) takenNames representing taken names  
# Returns: (string) name representing valid name  
**def getValidName(**message, takenNames**):** # Ask user until valid input given  
 **while True:** # Prompt user for input  
 name **=** input**(**message**)** # If user input is not empty and is not taken  
 **if** len**(**name**) >** 0 **and** name **not in** takenNames**:** # Then, return the user input integer  
 **return** name  
 # Otherwise, notify user of out of invalid name  
 **else:** print**("INVALID: Please enter a valid name.")**# Purpose: Get a valid integer from the user  
# Parameters: (string) message representing prompt to user  
# (integer) minNum representing minimum allowed number  
# (integer) maxNum representing maximum allowed number  
# Returns: (integer) num representing valid integer  
**def getValidInteger(**message, minNum, maxNum**):** # Ask user until valid input given  
 **while True:** # Try to convert input to integer  
 **try:** # Prompt user for input and convert to integer  
 num **=** int**(**input**(**message**))** # If user input integer is in valid range  
 **if** minNum **<=** num **<=** maxNum**:** # Then, return the user input integer  
 **return** num  
 # Otherwise, notify user of out of range input integer  
 **else:  
 raise** ValueError  
 # Catch value error and notify user of non-numerical input  
 **except** ValueError**:** print**("INVALID: Please enter an integer\n" +  
 "from " +** str**(**minNum**) + " to " +** str**(**maxNum**) + ".")**

# Purpose: Get set X and set Y and the preferences from the user  
# Parameters: None  
# Returns: (integer) n representing count of elements in each set  
# (map) setX representing elements and preferences of set X  
# (map) setY representing elements and preferences of set Y  
**def getPreferences():** # Initialize maps for set X and set Y  
 # Key: (string) Name of element  
 # Value: (list of strings) Names of preferred elements from other set  
 setX **= {}** setY **= {}** # Map each map to letter key to allow for code reuse  
 sets **= {"X":** setX, **"Y":** setY**}** # Ask user for integer value of n from 2 to 100  
 n **=** getValidInteger**("How many elements are in each set?\n\t"**, 2, 100**)** # Ask user for the name of each element for both sets  
 **for** setLetter **in** sets**:** # Start notification with appropriate word  
 **if** setLetter **== "X":** word **= "First"  
 else:** word **= "Now"** # Notify user of which set currently needs element names  
 print**("\n" +** word **+ ", provide the elements from set " +** setLetter **+ ":\n")** # Ask user for element name  
 **for** i **in** range**(**1, n **+** 1**):** sets**[**setLetter**][**getValidName**("What is the name of element #" +** str**(**i**) + " from set " +** setLetter **+  
 "?\n\t"**,  
 list**(**sets**[**setLetter**]**.keys**()))] = []** # Ask user for the preferences of each element for both sets  
 **for** setLetter **in** sets**:** # Start notification with appropriate word  
 **if** setLetter **== "X":** word **= "\nNext"** # Set opposite set to map setY  
 opposite **=** sets**["Y"]  
 else:** word **= "Lastly"** # Set opposite set to map setX  
 opposite **=** sets**["X"]** # Notify user of which set currently needs element preferences  
 print**("\n" +** word **+ ", provide the the preferences for " +  
 "the elements of set " +** setLetter **+ ":")** # Extract map for current set  
 \_set **=** sets**[**setLetter**]** # Ask the user for preferences for each element  
 **for** element **in** \_set**:** # Notify user of which element currently needs preferences  
 print**("\nNow for the preferences of " +** element **+ ":\n")** # Prepare list of un-prioritized element names from opposite set  
 preferences **=** list**(**opposite.keys**())** # While there are still multiple preferences to prioritize  
 **while** len**(**preferences**) >** 1**:** # Prompt user for next highest preference of current element  
 print**("Type the number of the next highest preference for " +** element **+ ":")** # Display the remaining preferences to prioritize  
 **for** i, y **in** enumerate**(**preferences, start**=**1**):** print**(**y **+ " (" +** str**(**i**) + ")")** # Remove user input from remaining un-prioritized list and  
 # append to list of respective list of element in map  
 \_set**[**element**]**.append**(**preferences.pop**(** getValidInteger**("\t"**, 1, len**(**preferences**)) -** 1**))** # Remove sole remaining un-prioritized preference and  
 # append to list of respective list of element in map  
 \_set**[**element**]**.append**(**preferences.pop**())** # Return n count, set X map, and set Y map  
 **return** n, setX, setY  
  
  
# Purpose: Determine a possible stable matching between preferences of set X  
# and preferences of set Y with both sets of equal count n  
# Parameters: (integer) n representing count of elements in each set  
# (map) setX representing elements and preferences of set X  
# (map) setY representing elements and preferences of set Y  
# Returns: yMatches (map) representing a possible stable matching  
**def galeShapleyAlgorithm(**n, setX, setY**):** # Declare map to represent prioritized preferences of each element of set Y  
 # Key: (string) Name of element y of set Y  
 # Value: (map) Name and index of preference from set X of y  
 # Note: Maps are almost guaranteed a time complexity of O(1) for lookups,  
 # which offers high optimization of performance at the expense of  
 # space complexity, especially with nested maps as done here  
 yPreferences **= {}** # Populate yPreferences map  
 # Iterate through each element y in set Y and keep track of current index  
 **for** i, yTuple **in** enumerate**(**list**(**setY.items**())):** # Declare map to represent the prioritized preferences of y  
 # Key: (string) Name of preference x from set X  
 # Value: (integer) Index (priority level) of x  
 priority **= {}** # Populate priority map  
 # Iterate through each x in the preferences list of y  
 **for** j, preference **in** enumerate**(**yTuple**[**1**]):** # Add name of x and index of x to priority map  
 priority**[**preference**] =** j  
 # Add name of y and prioritized preferences map of y to yPreferences map  
 yPreferences**[**yTuple**[**0**]] =** priority  
  
 # Free memory to reduce space complexity  
 **del** setY, j, preference, priority  
  
 # Declare map to represent the currently unmatched elements of set X  
 # Key: (integer) Name of element x from set X  
 # Value: (list of strings) Preferences from set Y of x  
 xUnmatched **= {}** # Populate xUnmatched map  
 **for** i, x **in** enumerate**(**setX**):** # Add preferences of x to xUnmatched map  
 xUnmatched**[**x**] =** setX**[**x**]** # Free memory to reduce space complexity  
 **del** setX  
  
 # Declare map to represent current existing matches  
 # Key: (string) Name of element of set Y  
 # Value: (string) Name of element of set X  
 yMatches **= {}** # Declare map to represent the currently matched elements of set X  
 # Key: (integer) Name of element x from set X  
 # Value: (list of strings) Preferences from set Y of x  
 xMatched **= {}** # Initialize count of unmatched elements of set X to n  
 xUnmatchedCount **=** n  
  
 # While there are still unmatched elements of set X  
 **while** xUnmatchedCount **>** 0**:** # Get the first unmatched element x of set X  
 xTuple **=** list**(**xUnmatched.items**())[**0**]** # Extract the name of x  
 x **=** xTuple**[**0**]** # Extract the list of preferences from set Y  
 xPreferences **=** xTuple**[**1**]** # Initialize counter index to zero  
 i **=** 0  
 # While x is still unmatched and  
 # there are still untested preferences of x to check  
 # Note: This single nested while loop is what makes this  
 # algorithm have a time complexity of O(n^2)  
 **while** x **in** xUnmatched **and** i **<** n**:** # Extract the first remaining preference y  
 y **=** xPreferences**[**i**]** # If y is not already matched with any other element of set X  
 **if** y **not in** yMatches**:** # Then, match x with y  
 # Set the current match of y as x  
 yMatches**[**y**] =** x  
 # Move x name from the xUnmatched map to the xMatched map  
 xMatched**[**x**] =** xUnmatched.pop**(**x**)** # Decrement the count of unmatched elements of set X  
 xUnmatchedCount **-=** 1  
 # Otherwise, if y is already matched with a  
 # different element of set X  
 **else:** # Then, check if y prefers x over the current match of y  
 # Determine the current match of y  
 yCurrentMatch **=** yMatches**[**y**]** # Extract the list of preferences of y from the yPreferences map  
 yPreference **=** yPreferences**[**y**]** # If the priority level of x is higher than  
 # the priority level of the current match of y  
 **if** yPreference**[**x**] <** yPreference**[**yCurrentMatch**]:** # Move the current match of y from the xMatched map  
 # to the xUnMatched map  
 xUnmatched**[**yCurrentMatch**] =** xMatched.pop**(**yCurrentMatch**)** # Then, match x with y instead  
 # Set the current match of y as x  
 yMatches**[**y**] =** x  
 # Move x name from the xUnmatched map to the xMatched map  
 xMatched**[**x**] =** xUnmatched.pop**(**x**)** # Note: Do not change xUnmatchedCount since the  
 # net gain/loss is zero since one element  
 # from x was matched while one other was unmatched.  
 # Otherwise, if y does not prefer x over the current match of y  
 **else:** # Increment the counter index to check  
 # the next remaining preference  
 i **+=** 1  
  
 # Return the final result of a possible stable matching  
 # between the preferences of set X and set Y  
 **return** yMatches  
  
  
# Purpose: Display the possible stable matches of a stable matching  
# Parameters: (map) result representing a stable matching  
# Key: (string) Name of element y from set Y  
# Value: (string) Name of element x from set X  
# Returns: yMatches (map) representing a possible stable matching  
**def displayResult(**result**):** print**("\n\nA possible stable matching between set X and set Y is:")** # Display each match as "x matched with y"  
 **for** y **in** result**:** print**(**str**(**result**[**y**]) + " matched with " +** str**(**y**))**# Purpose: Measure execution time of Gale-Shapely Stable Matching Algorithm  
# Parameters: (integer) n representing count of elements in each set  
# (map) setX representing elements and preferences of set X  
# (map) setY representing elements and preferences of set Y  
# (boolean) single representing whether to time a single execution  
# Returns: time (float) Execution time in microseconds  
**def measurePerformance(**n, setX, setY, single**):  
 if** single**:** # Return the time of a single execution  
 **return** timeit**(lambda:** galeShapleyAlgorithm**(**n, setX, setY**)**,  
 number**=**1**)  
 else:** # Return the shortest/fastest time of the three repetitions  
 **return** min**(** # Perform three repetitions of one million iterations each  
 repeat**(lambda:** galeShapleyAlgorithm**(**n, setX, setY**)))**# Purpose: Main function to prepare two sets and match them  
# Parameters: None  
# Returns: Nothing  
**def main():** # Get the elements of set X and set Y and the preferences  
 n, setX, setY **=** getPreferences**()** # Get a possible stable matching between the preferences of set X and set Y  
 stableMatching **=** galeShapleyAlgorithm**(**n, setX, setY**)** # Display the computed stable matching  
 displayResult**(**stableMatching**)** # Ask user whether to proceed to execution time analysis  
 **if** input**("\nWould you also like to measure the performance? " +  
 "(y/n)\n\t")**.lower**() == "y":** # Ask user whether to perform average or single measurement  
 **if** input**("\nDo you want to time a single execution or\n" +  
 "time the average of three million executions? " +  
 "(s/a)\n\t")**.lower**() == "a":** single **= False** word **= "average"  
 else:** single **= True** word **= "single execution"** print**("Measuring " +** word **+ " performance ...")** # Display execution time in microseconds rounded to 6 decimals  
 print**("\nThe stable matching was determined in about\n" +** str**(**round**(**measurePerformance**(**n, setX, setY, single**)**, 6**)) +  
 " microseconds.")** # Do not immediately close program on user  
 input**("Press any key to exit ...")**# # RESULT OF BEST CASE: NO CONFLICTING TOP PREFERENCES  
# displayResult(  
# galeShapleyAlgorithm(4,  
# {"Ben": ["Rey", "Jyn", "Padme", "Leia"],  
# "Cassian": ["Jyn", "Leia", "Rey", "Padme"],  
# "Anakin": ["Padme", "Jyn", "Rey", "Leia"],  
# "Han": ["Leia", "Jyn", "Padme", "Rey"]},  
# {"Rey": ["Ben", "Anakin", "Han", "Cassian"],  
# "Jyn": ["Cassian", "Han", "Anakin", "Ben"],  
# "Padme": ["Anakin", "Han", "Cassian", "Ben"],  
# "Leia": ["Han", "Cassian", "Ben", "Anakin"]}  
# )  
# )  
#

# # TIMING OF BEST CASE: NO CONFLICTING TOP PREFERENCES  
# print("The stable matching was determined in about\n" +  
# str(  
# round(  
# measurePerformance(4,  
# {"Ben": ["Rey", "Jyn", "Padme", "Leia"],  
# "Cassian": ["Jyn", "Leia", "Rey", "Padme"],  
# "Anakin": ["Padme", "Jyn", "Rey", "Leia"],  
# "Han": ["Leia", "Jyn", "Padme", "Rey"]},  
# {"Rey": ["Ben", "Anakin", "Han", "Cassian"],  
# "Jyn": ["Cassian", "Han", "Anakin", "Ben"],  
# "Padme": ["Anakin", "Han", "Cassian", "Ben"],  
# "Leia": ["Han", "Cassian", "Ben", "Anakin"]},  
# False  
# ),  
# 6)  
# )  
# + " microseconds.")  
#   
# # RESULT OF WORST CASE: COMPLETELY CONFLICTING TOP PREFERENCES  
# displayResult(  
# galeShapleyAlgorithm(4,  
# {"Ben": ["Leia", "Padme", "Jyn", "Rey"],  
# "Cassian": ["Leia", "Padme", "Jyn", "Rey"],  
# "Anakin": ["Leia", "Padme", "Jyn", "Rey"],  
# "Han": ["Leia", "Padme", "Jyn", "Rey"]},  
# {"Rey": ["Han", "Anakin", "Cassian", "Ben"],  
# "Jyn": ["Han", "Anakin", "Cassian", "Ben"],  
# "Padme": ["Han", "Anakin", "Cassian", "Ben"],  
# "Leia": ["Han", "Anakin", "Cassian", "Ben"]}  
# )  
# )  
#   
# # TIMING OF WORST CASE: COMPLETELY CONFLICTING TOP PREFERENCES  
# print("The stable matching was determined in about\n" +  
# str(  
# round(  
# measurePerformance(4,  
# {"Ben": ["Leia", "Padme", "Jyn", "Rey"],  
# "Cassian": ["Leia", "Padme", "Jyn", "Rey"],  
# "Anakin": ["Leia", "Padme", "Jyn", "Rey"],  
# "Han": ["Leia", "Padme", "Jyn", "Rey"]},  
# {"Rey": ["Han", "Anakin", "Cassian", "Ben"],  
# "Jyn": ["Han", "Anakin", "Cassian", "Ben"],  
# "Padme": ["Han", "Anakin", "Cassian", "Ben"],  
# "Leia": ["Han", "Anakin", "Cassian", "Ben"]},  
# False  
# ),  
# 6)  
# )  
# + " microseconds.")  
  
main**()**