Problem Set 1: How Close Was That Election?

Handed out: Monday October 22, 2018
Pset Due: **9pm on Wednesday, October 31, 2018**
Checkoffs: **November 5th - November 15th**

Introduction: A Look Into Election Results

In the United States, presidential elections are determined by a majority Electoral College vote. For this problem set, we're going to see if it is possible to change the outcome of an election by shuffling voters around to different states. Given simplified election results dating back to 2008, your job is find how close an election really was, that is the smallest number of voters needed to change an election outcome by relocating to another state. This is an interesting variation of the complementary knapsack problem presented in class, which we will explore more thoroughly below. **You may find 6.0002 Lecture 1 to be helpful for this problem set.**

Getting Started

- Do not rename files, change function/method names, or delete/edit given docstrings.
- You will need to keep **`test_ps1.py`** and all election result text files the same folder as **`ps1.py`**.
- Please review the Style Guide on Stellar. **Points will be deducted** if there are violations.

Problem 1: Creating a State Class (4%)

Before we dive into the knapsack problem, we first want to be able to represent all the states in the U.S. through a **`State`** class holding each state's respective election results for a given election year.

You can expect to initialize the class with the information given in each election results file:

```
State, # Democrat votes, # Republican votes, # Electoral College votes
```

Here is an example of how you would initialize a **`State`** instance:

```
State('AL', 941173, 692611, 9)
```

Implement a **`State`** class in with the following attributes: **`state`**, **`ec`**, **`winner`**, and **`margin`**. You can assume there will never be a tie.

You should now pass the test **`test_1_state_class`** in test_ps1.py.

Problem 2: Loading Election Data (4%)

Load an election result from any of the following tab-delimited data files: **`2008_results.txt`**, **`2012_results.txt`**, or **`2016_results.txt`**.

You can expect the data on each line to be formatted in sets of: `State`, `# Democrat votes`, `# Republican votes`, `# Electoral College votes`. Each `State` has a 2 letter string abbreviation and all other values are integers.

Here are the first few lines of **`2000_results.txt`**:

```
State Democrat   Republican EC_Votes
AL    941173     692611     9
AK    167398     79004      3
```

Implement the function **`load_election_results(filename)`** in **`ps1.py`**. It should take in the name of the data text file as a string, read in its contents, and return a list of corresponding **`State`** objects.

Hint: If you don't remember how to read lines from a file, check out the online python documentation here: https://docs.python.org/3/tutorial/inputoutput.html

Some functions that may be helpful:
`str.split`, `open`, `file.readline`

You should now pass the test **`test_2_load_election_results`** in test_ps1.py.

Problem 3: Election Helper Functions (10%)

You will be implementing several helper functions that will help us for the remainder of the problem set.

Implement the functions **`find_winner(election)`**, **`states_lost(election)`**, and **`ec_votes_reqd(election)`**.

For **`find_winner(election)`**, fill in the function such that it returns `('dem', 'gop')` if the Democratic candidate won, or `('gop','dem')` if the GOP candidate won. You may assume that there will be no ties.
For **`states_lost(election)`**, fill in the function such that it returns a list of states lost by the losing candidate (that is, the states won by the winning candidate).

For **`ec_votes_reqd(election)`**, fill in the function such that it returns the number of additional Electoral College votes needed by the losing candidate in order to win. To win, a candidate must have 1 more than half the total number of EC votes, i.e. if there are 538 electoral votes, a candidate must have >= 270 electoral votes in order to win the election.

You should now pass the test cases **`test_3_find_winner`**, **`test_3_states_lost`**, and **`test_3_ec_votes_reqd`** in test_ps1.py.

Problem 4: Greedy Election (14%)

We will first use a greedy algorithm to explore how to find the solution to our original problem. One way to change an election outcome is to choose a subset of states that originally won the election (returned by **`states_lost`**), and move voters of the opposite party into those states in order to upset the original winner of the state. In this way, the original losing candidate of the election wins over those new states so that they can become the new overall election winner. We will call the states needed to flip the election outcome "swing states."

One way we might build our list of swing states is to always choose the state with the narrowest win margin from from **`states_lost`**. By choosing the state with the narrowest (smallest) margin, we find the state that needs the least number of new voters from the opposing party in order to win over its EC votes. The greedy algorithm continues to choose new states until it reaches (or just exceeds) `ec_votes_needed` (the output of **`ec_votes_reqd`**).

For example, if you have state A with 2 EC votes that won by a margin of 1000, and state B with 20 EC votes that won by a margin of 2000, the greedy algorithm would choose state A since it had the narrowest win margin (1000<2000).

Implement a greedy algorithm for moving voters in **`greedy_election`**. The function should return a list of swing states where voters should be moved into. This should only return a subset of **`States`** that were originally lost by the losing candidate in the election (returned by **`states_lost`**).

You should now pass **`test_4_greedy_election`** in test_ps1.py.

Problem 5: Dynamic Programming Election Shuffling (28%)

In the problem above, our greedy algorithm makes a series of locally optimal choices. However, in some cases, this may not lead to the overall optimal solution. For instance, consider the

results of the 2012 election. The greedy algorithm returns [NH, NV, FL, DE, NM, IA, VT, ME, RI] as a list of swing states, requiring a total of 768,385 voters that must be relocated to shift the election outcome. A dynamic programming algorithm that returns the optimal solution instead finds the list [FL, NH, OH, VA], needing only 429,526 voters.

We will set up this problem as the complementary knapsack problem. Our objective is to choose a subset of **`lost_states`** such that we get the required number of Electoral College votes to change the election outcome, while relocating the minimum number of voters. Think of each **`State`** as a potential object to include in your collection, the **`State`**'s number of `ec_votes` as its weight, and the **`State`**'s `margin` as it's value. This is the *complementary* knapsack problem as we are ***maximizing*** **weight** while ***minimizing*** **value**.

We will divide this problem into two parts that together will solve the complementary knapsack problem. First, we will implement a dynamic programming algorithm to solve the traditional knapsack problem in **`dp_move_max_voters`**. Then we will take the complement to the list returned in **`dp_move_max_voters`** to get our final solution in **`move_min_voters`.**

Implement a dynamic programming algorithm in **`dp_move_max_voters`** to find the list of `States` with the *largest* number of voters needed to relocate in order to get **AT MOST** `ec_votes`. If this is not possible, return the empty list.

A `dp_move_max_voters` solution that does not use dynamic programming will receive zero points for this problem.

Notes:

- If you try using a brute force algorithm or plain recursion on this problem, it will take a substantially long time to generate the correct output.
- You may implement your algorithm using the top-down recursive method with memoization, or the bottom-up tabulation method. The former was covered in lecture, but we will accept either method.
- **The `memo` parameter in `dp_move_max_voters` is optional. You may or may not need to use this parameter, depending on your implementation.**
    - If you decide to use this parameter make sure that you check if it is set to the default parameter, **`None`**, and assign it to an empty dictionary.

Our dynamic programming function, **`dp_move_max_voters`**, provides us with the list of states that were lost by a wide margin. However, we want to find the states that were lost by narrow margins as we want to minimize the number of voters needed to flip the results of the election.

Implement **`move_min_voters`** which returns a list of **`States`** that comprise our "swing states." Swing states should include the states that require the minimum numbers of voters to be relocated in order to win the required number of EC votes to change the election result.

Hints:

- Consider the total list of states lost and remove the states that were lost by a large margin. This function should call **`dp_move_max_voters`**, with the parameter `ec_votes` set to (total #EC votes lost - `ec_votes_needed`).
- **`move_min_voters`** is analogous to the complementary knapsack problem that was discussed in lecture. **`dp_move_max_voters`** solves the associated knapsack problem. Imagine the **`State`** objects are items you are packing. What is the objective function? What is the weight limit in this case? What are the values of each item? What is the weight of each item?

Example:
Given the list of states that were lost by the GOP in the 2012 election, and the number of electoral votes needed for them to win the election:

- Your `dp_move_max_voters` function should return
  `>>> ['NY', 'MA', 'MD', 'DC', 'WA', 'HI', 'VT', 'NJ', 'CA', 'IL', 'CT', 'RI', 'OR', 'MI', 'MN', 'WI', 'NM', 'ME', 'PA', 'IA', 'DE', 'NV', 'CO']`
- Your `move_min_voters` function should return
  `>>> ['FL', 'NH', 'OH', 'VA']`

You should now pass **`test_5_move_min_voters`** in test_ps1.py. If you fail **`test_5_knapsack`**, this is a sign that you are not correctly using dynamic programming. Come to office hours for help!

Problem 6: Valid Voter Shuffling (10%)

In Problems 4 and 5, we found the swing states that could be targeted in order to sway the election. By moving voters into each of those states we successfully change the election outcome, making the losing candidate our new winner.

In this problem, you will find a mapping of voters of the form `{(from_state, to_state) : voters_moved}` such that voters are from states that were won by the losing candidate to each of the states in swing states. To flip the outcome of that state, (margin + 1) new voters must relocate to that state.

Note: States that were won by the losing candidate should not be lost after this voter shuffling, nor tied (`margin` must be greater than 0).

Implement **`flip_election(election, swing_states)`**. The function should return a tuple with (1) a dictionary with keys `(from_state, to_state)` and corresponding `voters_moved` values, (2) the number of Electoral College votes gained by the shuffling, and (3) the number of relocated voters necessary. If there is no valid way to reshuffle voters, returns `None`.

You should now pass **`test_6_flip_election`** in test_ps1.py.

Checkoff: (30%)

Attend Office Hours between 11/5 and 11/15 to receive your checkoff.

Hand-In Procedure

1. Save

Make sure your code is saved with the name **ps1.py**.

2. Time and Collaboration Info

At the start of ps1.py, in a comment, write down the number of hours (roughly) you spent on the problems in that part, and the names of the people you collaborated with. For example:

```
# Problem Set 1
# Name: Jane Lee
# Collaborators: John Doe
# Time: 8 hours
... your code goes here ...
```

3. Sanity checks

After you are done with the problem set, run your files to make sure they run without errors. **Make sure that any calls you make to different functions are under if __name__ == '__main__'**. This will make your code easier to grade and your graders and TAs happy.

4. Submit

Upload all your files to the “Problem Sets” page linked from Stellar. If there is some error uploading, please try the troubleshooting techniques outlined on the site, and as a last resort email the file to 6.0002-staff@mit.edu.
You may upload new versions of each file until the 9pm deadline, but anything uploaded after that will get a score of 0, unless you still have enough late days left.