Problem Set 1: How Close Was That Election?

The questions below are due on Friday April 15, 2022; 02:33:00 PM.
Checkoff start: Apr 08 at 09:00AM
Checkoff due: Apr 15 at 05:00PM

Download Files

Pset Buddy

You do not have a buddy assigned for this pset.

Objectives:

- Practice dynamic programming via a modified knapsack problem
- Apply computing and data analysis to a real-world social science problem

Collaboration:

- Students may work together, but students are not permitted to look at or copy each other's code or code structure.
- Each student should write up and hand in their assignment separately.
- Include the names of your collaborators in a comment at the start of each file.
- **Please refer to the collaboration policy in the Course Information for more details.**

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 moving voters around to different states. Given simplified election results dating back to 2012, your job is to find how close an election really was by finding 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:

Download **ps1.zip** from the problem set website. This folder contains the following files:

- **ps1.py**: code skeleton
- **state.py**: helper class representing the election results for a given state.
- **test_ps1.py**: basic tests for your code
- Several files with election result data in the form: ***_results.txt** and ***_results_brute.txt**

Make sure all these files are saved in the same directory. Please do not rename these files, change any of the provided helper functions, change function signatures, or delete docstrings.

Please review the Style Guide for good coding practices.

---

1) Loading Election Data

In **state.py**, you'll find that we have already implemented the `State` class for you. Familiarize yourself with the class, as you'll find it helpful throughout the pset. We will use the class attributes `state.dem` and `state.rep` to store the number of Democrat and Republican votes for each state.

In this problem, we will implement a function that allows us to load election results from text (.txt) files containing election data. We will be storing information regarding a state using the State class, given to you in `state.py`.

Each line in a data file contains the following information, each separated by a tab character:

- State name (2-letter state abbreviation),
- Number of Democrat votes,
- Number of Republican votes,
- Number of Electoral College votes

Each file begins with a header, that you should ignore when loading the election. Here are the first few lines of **2012_results.txt**:

```
State    Democrat    Republican    EC_Votes
AL    795695    1255925    9
AK    122640    164676    3
```

Implement the function `load_election(filename)`. It should take in the data text filename as a string, read its contents excluding the header, and return a list of corresponding State objects.

Hint: If you don't remember how to read lines from a text file, check out the online Python documentation here.

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

Example: Debug your implementation by running the following lines on your console, also located at the bottom of **ps1.py** under `if __name__ == "__main__"` :

```
>>> year = 2012
>>> election = load_election("%s_results.txt" % year)
>>> print(len(election))
51
>>> print(election[0])
In AL, 9 EC votes were won by rep by a 460229 vote margin.
```

1.1) Testing

You should now pass the test `test_1_load_election` in **test_ps1.py**.

---

2) Election Helper Functions

- `election_winner(election)`: Fill in the function such that it returns `('dem', 'rep')` if the Democratic candidate won the election (i.e. receives **more electoral college (EC) votes** than the Republican candidate), or `('rep', 'dem')` if the Republican candidate won.
    - You may assume that there will be no ties.
- `winner_states(election)`: Fill in the function such that it returns a list of states won by the winning candidate.

- `ec_votes_to_flip(election, total=538)`: Fill in the function such that it returns the number of additional Electoral College votes required 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 may assume that `total` is always even

Implement the `election_winner(election)`, `winner_states(election)`, and `ec_votes_to_flip(election)` functions.

Example: Debug your implementation by running the following lines on your console, also located at the bottom of **ps1.py** under `if __name__ == "__main__"` :

```
>>> winner, loser = election_winner(election)
>>> won_states = winner_states(election)
>>> names_won_states = [state.get_name() for state in won_states]
>>> reqd_ec_votes = ec_votes_to_flip(election)
>>> print("Winner:", winner)
Winner: dem
>>> print("Loser:", loser)
Loser: rep
>>> print("States won by the winner: ", names_won_states)
States won by the winner:  ['CA', 'CO', 'CT', 'DE', 'DC', 'FL', 'HI', 'IL', 'IA', 'ME', 'MD', 'MA', 'MI',
'MN', 'NV', 'NH', 'NJ', 'NM', 'NY', 'OH', 'OR', 'PA', 'RI', 'VT', 'VA', 'WA', 'WI']
>>> print("EC votes needed:",reqd_ec_votes, "\n")
EC votes needed: 64
```

2.1) Testing

You should now pass the test cases `test_2_election_winner`, `test_2_winner_states`, and `test_2_ec_votes_to_flip` in **test_ps1.py**.

---

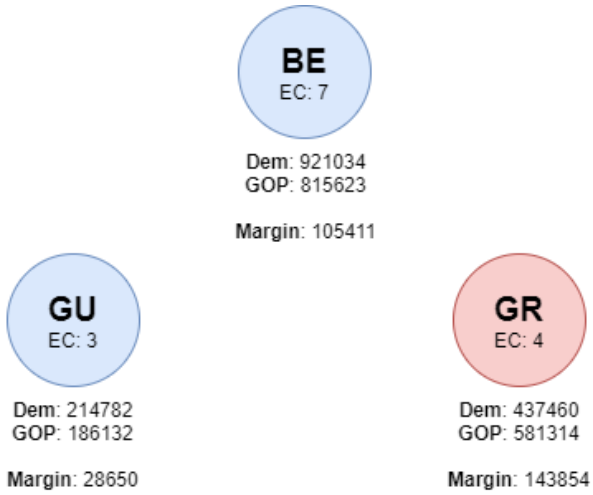
3) Brute Force Election Shuffling

3.1) Overview

We will explore algorithms for flipping the election result by relocating voters to different states, moving the fewest number of voters possible. For simplicity, we will restrict ourselves to only moving losing-candidate voters from losing-candidate states into winning-candidate states (in order to flip the states into losing-candidate states). We will first implement this through a brute force approach, by calculating every combination of moves possible and returning the combination that flips the election with the fewest number of voters moved.

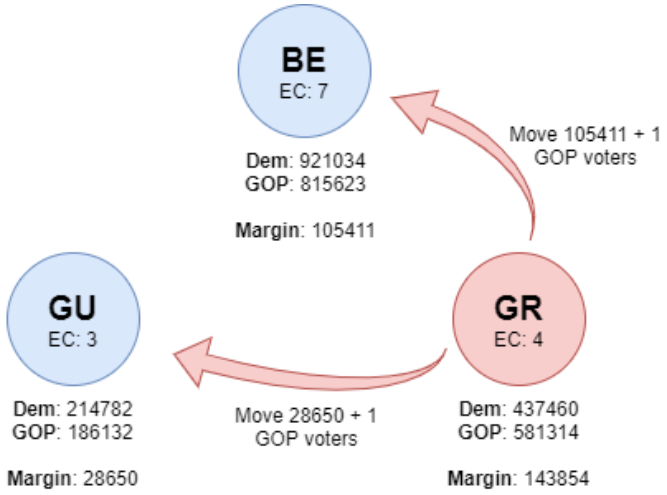
Let's walk through a small illustrative example:

Say we only have 3 states: **Bellissippi (BE)** with 7 EC votes, **Guttagia** (GU) with 3 EC votes and **New Grimson** (GR) with 4 EC votes. This gives us a total of 14 EC votes, meaning that any party must win at least (total/2 + 1) = *8 EC votes* in order to win the election, which we will call `ec_votes`. In our toy example, the recent election gave the following results, resulting in a Democrat win (dem: 10 EC votes, rep: 4 EC votes).



We will now move voters to change the EC votes until the Republicans win. Remember that in order to win the election we must move `margin+1` voters total. In this situation there are 3 possible ways we can move voters from GR, in order to change the state-level outcomes:

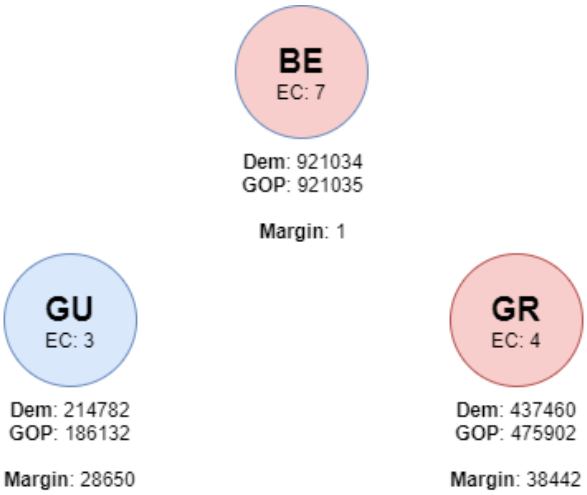
- Flip BE: Move 105411 + 1 Republican voters into BE
- Flip GU: Move 28650 + 1 Republican voters into GU
- Flip both: Move 105411 + 1 Republican voters into BE **and** 28650 + 1 Republican voters into GU



The results of each of these potential moves give us:

Option	# Voters Moved	EC Outcome	Election flipped?
Flip GU	28651	Dem: 7, GOP: 7	No!
Flip BE	105412	Dem: 3, GOP: 11	Yes!
Flip GU and BE	134063	Dem: 0, GOP: 14	Yes!

From this, we can easily pick out the second option as the combination of moves that minimizes voters moved with the constraint that the election must be flipped.



3.2) Implementation

Now we will write code to generalize this procedure to any number of states. Note that the number of potential move combinations grows exponentially with the number of states and so, while it may be theoretically possible to get a result using this method for any number of states, we will find that in practice our code runs too slowly. We will provide you with a helper function, `combinations`, to generate all possible combinations of an array. (Note that this function could also be implemented by hand, either recursively or iteratively. For further information on this, and the computational complexity involved, please check sections 9.6 and 12.5 in the text.)

Implement the following pseudocode:

```
initialize variables to hold the best combo and the minimum voters moved *so far*

find possible_move_combinations using helper function "combinations"
for combo in possible_move_combinations:
        if the sum of new EC votes >= EC votes required and number voters moved < minimum_so_far
                update best combo and the new minimum voters

return a tuple of the best combo as a list of the 'swing states' and the number of voters moved, or ([], 0)
if there is no move that can be made
```

Implement a brute force algorithm for moving voters in `brute_force_swing_states`. The function should return a tuple with a list of swing states where voters should be moved into and the number of voters moved. This should only return a subset of States that were originally lost by the losing candidate in the election (returned by `winner_states`). Return a tuple of an empty list and 0 if there are no swing states.

Example: Debug your implementation by running the following lines on your console, also located at the bottom of **ps1.py** under `if __name__ == "__main__"` :

```
>>> brute_election = load_election("60002_results.txt")
>>> brute_won_states = winner_states(brute_election)
>>> brute_ec_votes_to_flip = ec_votes_to_flip(brute_election, total=14)
>>> brute_swing, voters_brute = brute_force_swing_states(brute_won_states, brute_ec_votes_to_flip)
>>> names_brute_swing = [state.get_name() for state in brute_swing]
>>> ecvotes_brute = sum([state.get_ecvotes() for state in brute_swing])
>>> print("Brute force swing states results:", names_brute_swing)
Brute force swing states results: ['BE']
>>> print("Brute force voters displaced:", voters_brute, "for a total of", ecvotes_brute, "Electoral College
```

```
votes.\n")
Brute force voters displaced: 105412 for a total of 7 Electoral College votes.
```

3.3) Testing

You should now pass `test_3_brute_swing_states` in **test_ps1.py**.

---

4) Dynamic Programming Election Shuffling

In the problem above, our brute force algorithm considers all possible solutions in order to find the correct solution. However this approach takes an extremely long time to run. For instance, try running your brute force algorithm on the **2020_results.txt** file. Use `ctrl+C` to stop the code running once you get bored of waiting for it to complete. This approach has computational complexity on the order of `O(2^n)`, where `n` is our number of states. A dynamic programming algorithm that returns the optimal solution is a lot less computationally expensive.

We will set up this problem as the complementary knapsack problem. We want to find a subset of `winner_states` (ie. swing states) which collectively contribute at least the minimum required number of EC votes to flip the election outcome, while moving as few total voters into these states. Think of each `State` as a potential object to include in your knapsack collection, the `State`'s number of `ec_votes` as its weight, and the `State`'s `margin` plus one (voters moved in to flip) as its value. Finding the optimal swing states is the complementary knapsack problem as **we are trying to minimize the total value (voters moved) of our knapsack while staying above a certain threshold for total weight (`ec_votes`)**. It is different from the original knapsack problem that tries to maximize the total value (voters moved) of our knapsack while staying below a certain threshold total weight (`max_ec_votes`).

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 original knapsack problem in `max_voters_moved`. Then we will take the complement of the list returned in `max_voters_moved` to get our final solution in `min_voters_moved`.

4.1 or 4a) `max_voters_moved`

Implement a dynamic programming algorithm in `max_voters_moved`. This function should return the list of `States` with the largest number of total voters needed to relocate in order to get **at most** `max_ec_votes` and the total number of voters that are relocated (as a tuple including a list of `State` objects and an integer). If every state has a number of EC votes that is greater than `max_ec_votes`, return the empty list and 0.

A `max_voters_moved` solution that does not use a dynamic programming approach will be penalized during checkoffs.

Notes:

- If you try using a brute force algorithm or plain recursion on this problem, it will take a unacceptably long time to generate the correct output. Code that times out due to not implementing a dynamic programming solution will not be manually graded.
- You may find the lecture 1 notes and code helpful for an example of implementing a top-down dynamic programming knapsack problem.
- You may use either a bottom-up or top-down dynamic programming approach. Most good implementations of the top-down approach involve an additional helper function to allow passing in a dictionary for memoization.

Example: Debug your implementation by running the following lines on your console, also located at the bottom of **ps1.py** under `if __name__ == "__main__"` :

```
>>> total_lost = sum(state.get_ecvotes() for state in won_states)
>>> non_swing_states, max_voters_displaced = max_voters_moved(won_states, total_lost-reqd_ec_votes)
>>> non_swing_states_names = [state.get_name() for state in non_swing_states]
>>> max_ec_votes = sum([state.get_ecvotes() for state in non_swing_states])
>>> print("States with the largest margins (non-swing states):", non_swing_states_names)
States with the largest margins (non-swing states): ['WI', 'WA', 'VT', 'RI', 'PA', 'OR', 'NY', 'NM', 'NJ',
'NV', 'MN', 'MI', 'MA', 'MD', 'ME', 'IA', 'IL', 'HI', 'DC', 'DE', 'CT', 'CO', 'CA']
>>> print("Max voters displaced:", max_voters_displaced, "for a total of", max_ec_votes, "Electoral College
votes.", "\n")
Max voters displaced: 11353398 for a total of 268 Electoral College votes.
```

4.1.1) Testing

You should now pass `test_4_max_voters` in **test_ps1.py**. Failing this is a sign that your dynamic programming solution is not correct. If you are getting the correct list of states but an incorrect number of voters relocated, check to ensure you are moving the right number of voters to flip the states' elections such that the loser has more votes than the loser. Feel free to ask for help during office hours, we're here to help :)

4.2 or 4b) `min_voters_moved`

Our dynamic programming function, `max_voters_moved`, provides us with the list of states that **maximizes the total voters moved while staying below some threshold total number of EC votes**. By passing in carefully chosen parameters when calling `max_voters_moved` in `min_voters_moved`, we can use the resulting list to indirectly determine our list of swing states to flip the election result.

Consider the 2012 election. In `winner_states`, Obama is the original winner and Romney is the original loser. To flip the election `winner_states` are either kept by Obama (non-swing), or flipped to Romney (swing).

Let `delta_ec_votes` be the maximum number of EC votes Obama can keep if Romney wins instead. If we set our threshold number to `delta_ec_votes`, then `max_voters_moved` will return all the non-swing states. To help see this, if instead `max_voters_moved` picked a swing state in its output list, then it would waste some EC votes that could have been spent picking a state that was for Obama and moving the voters from there to the other side. Therefore, the swing states would be the states in `winner_states` that aren't in the list of non-swing states, i.e the complement of `max_voters_moved`. These swing-states minimize voter movement to flip the election.

Implement `min_voters_moved` which returns a list of `States` that comprise our "swing states" and the number of voters relocated to the swing states (as a tuple including a list of `State` objects and an integer). Swing states require the minimum number of total voters to be relocated in order for the original election loser to win the required number of EC votes to win the election.

Hints:

- This function should be relatively simple if you use `max_voters_moved` correctly in your implementation. If you are having trouble getting started talk to us at OH.
- Consider the full list of states won by Obama and remove the states that Obama should continue to win (the non-swing states). The remaining states are ones that Romney should now win (i.e. the swing states).
- This function should call `max_voters_moved`, with the parameter `max_ec_votes` set to (total #EC votes won by Obama – `ec_votes`).
- `min_voters_moved` is analogous to the complementary knapsack problem that was discussed in lecture. `max_voters_moved` solves the associated knapsack problem.

- If you have are returning the correct list of swing states but an incorrect number of voters moved, make sure to check that you are moving the right number of voters needed to flip a swing state.

Debug your implementation by running the following lines on your console, also located at the bottom of **ps1.py** under `if __name__ == "__main__"` :

```
>>> swing_states, min_voters_displaced = min_voters_moved(won_states, reqd_ec_votes)
>>> swing_state_names = [state.get_name() for state in swing_states]
>>> swing_ec_votes = sum([state.get_ecvotes() for state in swing_states])
>>> print("Complementary knapsack swing states results:", swing_state_names)
Complementary knapsack swing states results: ['FL', 'NH', 'OH', 'VA']
>>> print("Min voters displaced:", min_voters_displaced, "for a total of", swing_ec_votes, "Electoral College
votes. \n")
Min voters displaced: 429526 for a total of 64 Electoral College votes.
```

4.2.1) Testing

You should now pass `test_4_min_voters` in **test_ps1.py**.

4.3) Electoral College Ties

In the event that there is a tie in the electoral college then the election is sent to congress to decide the winner. Under `if __name__ == "__main__"` make a small change to the given code for Problem 4.2 to calculate the min voters to move to **tie** in the electoral college (in 2020) rather than winning outright. Write down what you find and be prepared to share it during your checkoff.

You might also find it interesting to look at how many votes were needed to tie other years to get a sense of the results for 2020.

---

5) Valid Voter Shuffling

In Problems 3 and 4, we found the swing states that could be targeted in order to flip the election. The swing states were won by the original election winner, but we want to move voters into these swing states in order to flip the outcome of the election. In this problem, you will find a valid way to move voters from states that were won by the original election loser to each of the states in our swing states. However, the residents of some states consider their state ideal and would **not** be willing to move to any other state.

Your task is to find a mapping of the form `{(from_state, to_state): voters_moved}` indicating the number of voters being moved from a state won by the original election loser to a swing state. To flip the outcome of the swing state, remember that `margin + 1` new voters must relocate to that state.

This is an open-ended problem and there are many correct ways to approach it. Here is one:

1. Find the list of states that were won by the election loser and that are not ideal states (in the list `ideal_states`); let's call these the `losing_candidate_states`. These are the states we will move voters from.
2. Go through each state in `swing_states`, and move just enough voters from the `losing_candidate_states` into this state to upset the winner of the state.
    - For each state in `swing_states`, you'll need `state.get_margin()+1` voters to relocate to that state in order to upset the winner of the state.

- For each state in the `losing_candidate_states`, you may not relocate more than `state.get_margin()-1` voters, as that would change the outcome of the election of that state.

Important Notes:

- States that were won by the original losing candidate should continue to be won by the original losing candidate after voter shuffling (margin must be greater than 0).
- One state can supply voters for multiple states.
- Multiple states can supply voters for one state.
- Remember that you cannot move residents from `ideal_states`
- Make sure to refamiliarize yourself with the State class as it contains methods which can greatly simplify the bookkeeping for this function.

Implement `relocate_voters(election, swing_states, ideal_states)`. If there is a way to reshuffle voters, this function should return a tuple with

1. the number of relocated voters necessary.
2. a dictionary with keys `(from_state, to_state)` (where the tuple elements are the state abbreviations) and their corresponding `voters_moved` values.
3. the number of Electoral College votes gained by the shuffling.

If there is no valid way to reshuffle voters, the function should return `None`.

Debug your implementation by running the following lines on your console, also located at the bottom of **ps1.py** under `if __name__ == "__main__"`:

```
>>> print("relocate_voters")
>>> flipped_election = relocate_voters(election, swing_states)
>>> print("Flip election mapping:", flipped_election)
Flip election mapping: (42921, {('AK', 'AZ'): 10458, ('AK', 'GA'): 11780, ('AK', 'WI'): 13802, ('AR', 'WI'):
6881}, 37)
```

Note that the result above is one of several potential valid solutions, and that a valid solution produced by your code may look very different.

5.1) Testing

You can test your implementation of `relocate_voters` by running the test file. You should now pass `test_5_move_voters` in **test_ps1.py**. There are multiple correct reshuffling of voters that satisfies the requirements and flips the result of the election.

---

6) Hand-in Procedure

6.1) Time and Collaboration Info

At the start of each file, in a comment, write down the names of your collaborators. For example:

```
# Problem Set 1
# Name: Jane Lee
# Collaborators: John Doe
```

Please estimate the number of hours you spent on the Problem Set in the question box below.

8

Thank you!

You have infinitely many submissions remaining.

6.2) Submission

Be sure to run the student tester and make sure all the tests pass. However, the student tester contains only a subset of the tests that will be run to determine the problem set grade. Passing all of the provided test cases does not guarantee full credit on the pset. Make sure that any calls you make to different functions are under `if __name__ == '__main__'`.

You may upload new versions of each file until Apr 07 at 09:00PM, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.

When you upload a new file your old one will be overwritten.

Download Most Recent Submission

Select File No file selected

Code Submitted Successfully!

You have infinitely many submissions remaining.