Homework 2: Backtracking Search

Due February 5th at 10pm

Submit the **simple_exchange.py** and **constrained_exchange.py** files along with your question responses on Gradescope. **Please do not rename your Python files.**

Part 1: Gift Exchange (80 points)

Your goal is to write a program to create matches for a gift exchange, which is a problem that requires *backtracking search*. In a gift exchange, each participant gives a gift to exactly one other participant.

You should write your code for this section in the file **simple_exchange.py**. You should add function calls to **main** to test your code as you go. Submit this file on Gradescope. **Please keep the original name when you submit.**

Reading Participant Data

Our program will read in a list of participants from a plain text file. Write a function called **parse_data** that takes a file name as an argument, reads data from the file, and returns a list of participants.

Strip trailing new lines from the participant names, for tidiness.

Setting Up the Search Problem

We will represent the search problem with a dictionary. The goal is to find a dictionary that contains a key for each participant with a value representing a valid gift assignment.

Write a function called **set_up** that takes the list of participants. It should return a dictionary containing a key for each participant, paired with the value None. A key with a None value represents a gift-giver that has not yet been paired with a gift-receiver.

Validating Matches

We will need a function that tells us whether an assignment between two participants is valid. Right now, the only reason that an assignment would be invalid is if the person is assigned to themselves. In Part 2, we will add more constraints.

Write a function called **blocked**. It should take a gift-giver and a gift-recipient and return True if the giver is blocked from giving that participant (i.e., they are the same person). It should return False otherwise.

Assign

To find a valid assignment of givers and receivers, we can use **backtracking search**. The intuition is that we guess an assignment and then check if any conflicts arise. If they do, we undo that guess and try out a different assignment.

Backtracking search is a recursive algorithm that operates as follows. Given an assignment dictionary A:

- If A is already filled completely, return A.
- If A can't be solved (e.g., a person is assigned to themselves), abort.
- Otherwise, select *one* gift-giver that has not been assigned a recipient. Generate a list of possible next assignment dictionaries using the list of unassigned participants. For each possible next assignment dictionary, recursively apply the search function:
  - If any next assignment dictionary produces a valid solution, return that assignment dictionary
  - If there is no way to complete the assignment dictionary correctly, abort and return

This approach will work as long as the list of participants is reasonably small. You will need to write two functions: **assign**, the main backtracking search function, and **next_pairs**, a helper function that returns a list of possible next assignment dictionary/unassigned participant list pairs.

Next Pairs

Start by writing the **next_pairs** helper function. It should take an assignment dictionary, a list of unassigned gift-receivers, and the name of a gift-giver (a key in the assignment dictionary).

It should produce a list of all possible assignments to that gift-giver, using the list of remaining unassigned gift-receivers. Each possible assignment should be represented as a pair containing an assignment dictionary and a list of unassigned participants.

For each possible gift-receiver, you will need to do the following:

- Check to make sure the gift-giver is not blocked from giving to that recipient
- Make a copy of the existing assignment dictionary
- Update the copied dictionary with the new assignment of the recipient to the gift-giver
- Remove the recipient from the list of unassigned recipients

Finally, you should **randomize** the list of possible assignments before returning it. This keeps participants from giving to the same people year after year.

Assign

Now you can write your main problem-solving function, **assign**. **assign** takes an assignment dictionary and a list of unassigned participants and implements the backtracking search algorithm discussed above. It searches for a valid assignment by selecting an unassigned gift-giver, calling **next_pairs** to get the list of possible assignments, and exploring each possible assignment recursively.

If the unassigned participant list is empty, a valid assignment has been found, and the function should return the assignment dictionary.

If the list of possible next assignments is gone through without success, this is a failure: the function should return None to indicate the lack of a success.

Validate Matches

Once you have implemented everything above, your program should be correctly finding valid matches. Write a function to double-check this called **validate_matches**. It should go through an assignment dictionary and check that each participant is assigned exactly once as a gift-giver and as a gift-receiver, and that no one is giving to themselves.

It should return True to indicate a valid assignment dictionary and False otherwise.

Exporting Matches

You can use the provided **export_matches** function to write your generated gift assignments to file. It takes a dictionary of assignments as its argument.

Part 2: Adding Constraints

In this part, we will modify our gift-matching program to take in additional constraints on the matches. Maybe some participants usually give each other gifts outside of the exchange. Or maybe they don't get along. We will rework our program so that it incorporates these additional restrictions.

Write the code for this section in **constrained_exchanged.py**. It reads data from **constraints.txt** and writes data to *matches.txt*.

Reading Constraints From File

Modify your **parse_data** function so that it reads constraints from the input file. Each participant name is now followed by a list of other participants that that person is blocked from giving to. The participant's name is separated with a tab character from this list; the list itself is comma-separated.

You will need to read this data from file and construct a constraint dictionary. The constraint dictionary contains a key for each participant associated with a list of that participant's blocked recipients.

Your function should now return a list of participants and a constraint dictionary.

Using Constraints

Next, modify your **blocked** function so that it uses the information in the constraint dictionary. A giver is now blocked from giving to themselves or to anyone in their constraint dictionary list.

(You will also need to pass the constraint dictionary as an argument to **assign** and **next_pairs** so that it is accessible to **blocked**.)

Question

If you've implemented everything above, your program should now be able to assign matches that respect the given constraints. However, backtracking search can be inefficient with large search problems.

Can you think of a way to improve the efficiency of our solver? Hint: think about how we select a gift-giver to "guess" about.

Part 3: Reading Reflections

These questions ask you to reflect on Chapter 1 of *You Look Like A Thing And I Love You*.

Question 1 (10 points)

In Chapter 1, Shane discusses four signs of AI doom: indications that a problem is not amenable to AI. But right now, whether we like it or not, many, many people are trying to solve problems with AI.

Consider three plausible contemporary scenarios of automating grading in a university:

(1) The computer science department is currently overwhelmed with the number of students who would like to take CS courses. In order to increase the number of students who can take Intro CS, the instructor team decides to use autograding for all assignments. The autograder uses the same test suite that the human instructors used to use to assess whether a program executes correctly. However, it cannot give feedback on programming style.

(2) The philosophy department is concerned about implicit bias in grading essays. Perhaps instructors are swayed by their own preferences or beliefs when assessing student work. Moreover, philosophy, like CS and many STEM fields, is a discipline that is heavily white and male. To combat the possibility of implicit bias in grading, the philosophy department trains an algorithm to grade essays, using samples of student work from previous years.

(3) The psychology department would like to offer a more flexible Intro to Statistics course. Currently, the course requires students to program in R, annoying students who prefer Python or Matlab. To relax this requirement, the department builds a system that translates code from these languages into R so that they can grade all submissions using their R test suite. To train this system, they scrape programs from Github, which may or may not include statistical analysis.

Discuss which of Shane's four signs of AI doom apply to each scenario.

Question 2 (10 points)

Compare and contrast the appropriateness of the use of technology in these scenarios.