Readme_Garcias

Version 1 8/22/24

1. Team name: Garcias

2. Names of all team members:
   a. Rene Alzina
   b. Juan Chaia Gomez
   c. Michael Egan

3. Link to github repository: https://github.com/ralzina/Project1-TOC

4. Which project options were attempted:
   a. Knapsack
      i. Brute force
      ii. Backtracking
      iii. Best case

5. Approximately total time spent on project: 12 hours

6. The language you used, and a list of libraries you invoked:
   a. Language: Python
   b. Libraries:
      i. Collections for defaultdict.
      ii. Matplotlib for graphing
      iii. Pytest for testing
   c. Every other library was already in the template.

7. How would a TA run your program (did you provide a script to run a test case?)
   a. To run the program, you will position yourself in the root folder and run ```**uv run main.py**``` which will run the code with the input cnf file, take the time of each case, and generate a graph.
   b. To run tests, position yourself in the root folder and run ```**uv run pytest**``` which will run the test_knapsack_garcias.py script. This sets up the knapsack class and runs all test cases from the cnf file inside the tests folder.
   c. There's no script to run the test case as they can be run directly from the root folder with uv run pytest.

8. A brief description of the key data structures you used, and how the program functioned. We used a dictionary to map coin values to the amount of coins present. We also have a used dictionary that maps coin values to the amount of coins of that value that you use to reach the target.
   a. Brute force:

i. This recursive function tries using one of each coin recursively to try all combinations. In each recursive call, it subtracts the used coin to the target. If the target ever reaches exactly 0, there is a solution, but if you try all combinations and it never reaches exactly 0, there is no solution.
 b. Backtracking:
  i. This recursive function tries using one of each coin recursively. In each recursive call, it subtracts the used coin to the target. If the target ever reaches exactly 0, there is a solution, but if the current target is less than 0 then it stops checking. This pruning allows for a better time efficiency.
 c. Best case:
  i. This recursive function is very similar to backtracking, but now we add an extra variable to keep track of the best case. Every time you pick a coin to get closer to the target, you check if it's the best case yet. If it is, then this best case variable becomes the current combination of coins. If not, then you keep trying other combinations. At the end, even if you didn't reach the final target, you will still get the best case which is the combination of coins that gets closest to the target. However, if you do reach exactly the target, you simply return the combination of coins that adds up to the target.

9. A discussion as to what test cases you added and why you decided to add them (what did they tell you about the correctness of your code). Where did the data come from? (course website, handcrafted, a data generator, other)
 a. We decided to add multiple test cases to test the behavior of our program in different situations.
 b. There were some that tested when no coins were given and the target was 0, or when no coins were given and the target was greater than 0.
 c. Other test cases had the target be too big even if you used all the given coins.
 d. We also added test cases where there was a combination of coins that gave the exact target.
 e. The test cases were hand crafted by understanding the natural behavior of the bin packing problem and the expected outcome.
 f. Finally, we added another set of test cases specifically for the best case subproblem because this one doesn't just care if the problem is solvable or non-solvable. This actually cares about the combination of coins that gets you the closest to the target value, so we created a different set of tests which is inside the tests folder.
 g. We couldn't place our tests script inside a module_tests folder because pytest gave us an error since it couldn't import the modules from src because src was an external folder. The tests only worked when we placed our test script inside the root directory so that it could access src.

10. An analysis of the results, such as if timings were called for, which plots showed what? What was the approximate complexity of your program?

a. The approximate time complexity for the program is $O(2^m)$ where m is the total number of coins present since you're basically generating all possible subsets of all the coins present and looking for a subset whose numbers' sum is equal to the target.

b. All plots showed that regardless of how efficient your program is, the time complexity is still exponential.
   i. Brute force took the largest amount of time since it tries all combinations even if the combination is no longer valid.
   ii. Best case was next because it always tries returning an answer even if no combination would add up to target. This means it must try more combinations even if we already know that reaching the target is impossible with the given coins.
   iii. Backtracking is the fastest because it prunes and can stop trying coins if it determines the combination is no longer valid which speeds up the process. Also, since it's not best case, it doesn't try to find the best answer, it only cares if it's good or not so it can stop checking early.

11. A description of how you managed the code development and testing.
   a. The way we managed the code development was by looking at the helper function examples for the other types of problems as a guide to creating our own knapsack helper function.
   b. We then connected it to our knapsack class with the implementation of the methods for brute force, backtracking, and best case.
   c. By analyzing the cnf template, we made both the input file in the input folder and the file with test cases in the tests folder. The input file has cases of increasing complexity to graph the time it takes to solve each case. The tests simply ensure that the functions work in different scenarios. After running the program, it writes the results into the results folder. This folder has both the csv files with the results of all input cases, and also a plot of the time complexity as we increase the magnitude of the problem.
      i. Cnf format
         1. To make our cnf format clearer for the cnf files in our input and tests folder, I will explain how it's formatted.

            c <instance> <target> <solvable>
            p knap <n_unique_coins>
            <coin_value_i> <coin_amount_i>

            Where n_unique_coins refers to the amount of unique coins, and coin_value_i as well as coin_amount_i refer to the value, amount pairs for coin i, where i goes from coin 1 to coin n.

            <solvable> always has an S if the problem is solvable and an N if it's not solvable. This was helpful for our test cases to check the

output of our program. However, for our input, we put a ? which basically means we are not trying to check or compare, we just cared about running the program.

12. Did you do any extra programs, or attempted any extra test cases
    a. No test cases were provided for knapsack, so we had to develop our own.
    b. No helper function was provided for knapsack, so we had to develop our own.
    c. No knapsack script was provided, so we had to develop our own.
    d. We had to add a parsing function in dmaics_parser.py to parse our knapsack cnf file.
    e. We had to create our knapsack cnf input file.
    f. We updated constants to account for our input and test files, and made sure everything was saved properly into the results folder.