Project Report: Finding Minimal Payment

Yousef Suleiman (I worked individually)

Due: 4/26/24

Overview

This Python script will return optimal itemization plan and its associated minimum payment when given a set of promotions and an input representing a cart of items.

Items are each associated with a ticket price correlating to the item's unit price.

Promotions can be applied if a minimum set of specific items are purchased. If they are, the set of items can be sold at a fixed price lower than the sum of their ticket prices.

Structure

File Parsing

`parse_input(path, encoding)`

Reads item purchases from a file where the first line contains the number of items and the remaining lines contain an `item_id`, `amount` being purchases, and `ticket_price` all separated by white space. It returns a dictionary mapping item IDs to tuples of quantity and price.

`parse_promotions(path, encoding)`

Reads promotion data from a file where the first line contains the number of promotions. The remaining lines contain the number of types of items in the promotion, pairs of `item_id` and required `amount`, and `total_price`. It returns a list of promotions where each promotion includes a dictionary mapping item IDs to required amounts and its total promotional price.

Promotion Application Logic

`is_eligible(cart, merchandise)`

Checks if the current cart meets the requirements for a given promotion.

`apply_promotion(cart, merchandise)`

Applies a promotion to the cart, updating the quantities of items based on the promotion, and returns the new cart state.

Cost Calculation

`calculate_output(purchases, promotions)`

Computes the minimum payment by considering all combinations of promotions and direct purchases using a depth-first search approach along with memoization. The function also keeps track of the itemization of purchases (either direct or through promotions).

Writing Output

`write_output(path, encoding, verbose, purchases, promotions, itemization, price)`

Writes the detailed itemization and total cost to an output file and optionally prints this information based on the verbose flag.

Command-Line Interface

The script uses `argparse` to handle command-line options for input and output file paths, encoding, verbosity, and an option to turn off memoization. For detailed instructions on how to use the script's parameters, refer to `README.md`.

Analysis of Algorithm Complexity

The script's complexity mostly comes from the recursive `calculate_output` function, which needs to consider every combination of applying promotions to the current shopping cart state. This part is computationally intensive, especially for large input sizes with many promotions.

Memoization Strategy

Memoization is done using Python's built in dictionary. Because the `cart` is internally represented by a dictionary and because dictionary items are not hashable, a `frozenset` is made of each cart state.

Inside each dictionary entry, a mapping `cart` -> `(itemization, price)` is stored where `price` is the total price following that itemization strategy and `itemization` is a list of `(matcher, is_promotion)`. The value `matcher` is either an `(item_id, amount)` representing ticket priced purchase or it is an index into the `promotions` list. The value `is_promotion` indicates which case it is.

Scalability and Efficiency

The script is effective for scenarios with a reasonable number of items and promotions but could face performance issues with extremely large datasets due to the exponential growth of possible combinations to evaluate.

Memoization helps mitigate these performance costs by ensuring that each unique state of the cart is computed only once.

Test Cases

1. Demonstrating Correctness

This test case is small so it is easy to see that the correct branch is chosen when traversing the tree of possibilities. This test case is in the file `simplt-input.txt` and is the same test case as the one in the project writeup.

```
python script.py -i simple-input.txt -v
Elapsed time: 0:00:00 seconds
# Itemization:

1. Buy 2 of item 7 at ticket price $2.0

2. Apply promotion buy 1 of item 7, 2 of item 8 for $10.0.

Total: $14.0
```

The same input is also shown when memoization is turned off:

```
python script.py -i simple-input.txt -v -nm
Elapsed time: 0:00:00 seconds
# Itemization:

1. Buy 2 of item 7 at ticket price $2.0

2. Apply promotion buy 1 of item 7, 2 of item 8 for $10.0.

Total: $14.0
```

2. Performance

This test case is not too large but large enough to see how memoization effects scaling. This test case is in the file `input.txt`.

```
python script.py -v
Elapsed time: 0:00:00.008012 seconds
# Itemization:

1. Buy 1 of item 1 at ticket price $2.5

2. Buy 3 of item 2 at ticket price $12.0
...

1014. Apply promotion buy 1 of item 7, 2 of item 8 for $10.0.

Total: $217.5
```

This optimal itemization strategy is much longer with 1014 steps. The time for this test case was less than a second. Now, demonstrating with memoization turned off:

```
Elapsed time: 0:00:11.858997 seconds
# Itemization:

1. Buy 1 of item 1 at ticket price $2.5

2. Buy 3 of item 2 at ticket price $12.0
...

17. Apply promotion buy 2 of item 1, 1 of item 2 for $15.0.

Total: $217.5
```

The time is now nearly 12 seconds which is much greater than before. The strategy found without memoization was interestingly much shorter with 17 steps. However, the total prices are both $217.5 indicating that they are both optimal.