**SI507 Project Document**

Tianhao Liang

**Project code**

Link to github repo: <https://github.com/liangth35/SI507project>

README.md about instructions for running the code is included in the repo.

Required packages:

bs4

flask

requests

pandas

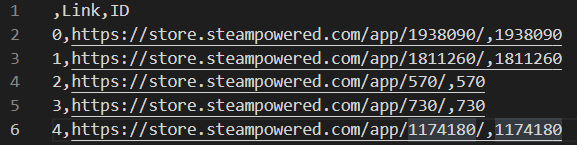
plotly

**Data sources**

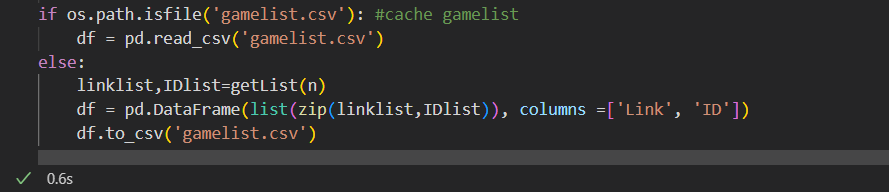
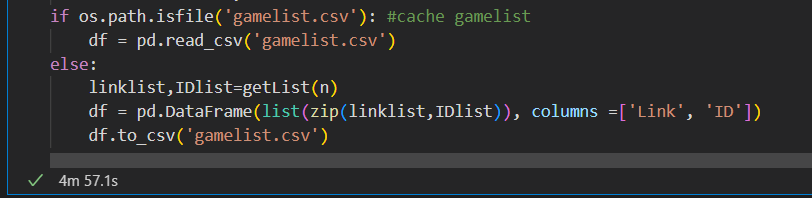
First, I crawled the searching page on steam to obtain a list of games and the urls of their store pages. The url I used for crawling was [https://store.steampowered.com/search/?ignore\_preferences=1&category1=998&os=win&filter=globaltopsellers&page=%d](https://store.steampowered.com/search/?ignore_preferences=1&category1=998&os=win&filter=globaltopsellers&page=%25d), in which %d was replaced by the page numbers. There are about 25 games on each searching page. I scraped 404 pages in total and got 6654 valid game id–url pairs.

I obtained the html files of these pages using request.get, and I used beautifulsoup to extract the game ids and urls form the html files. I put the game ids and urls into a pandas dataframe. (The urls were used for crawling later and game ids were used as dictionary key in the next part.)

I implemented caching in this part by saving the dataframe as a csv file. When executed, if the program finds the csv file, it directly reads from the file instead of scraping the pages again. This is part of the cached csv file:

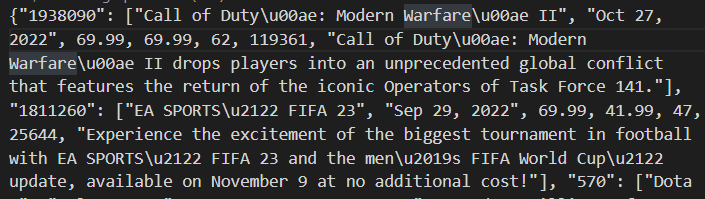


And this is a screenshot of my code that implemented caching. It took 5 minutes to execute the program for the first time, but it took only 0.6 seconds for the second time.

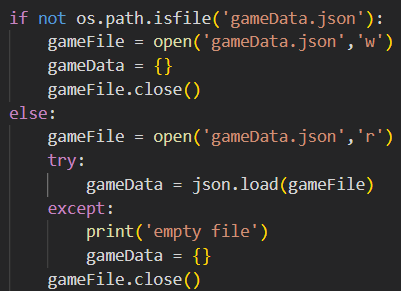


For the next step, I scraped the urls I obtained in the previous part, there were 6654 urls and I applied request.get to all of them and extracted the **name, release date, original price, discounted price, rating, review number, and description** of each game using beautifulsoup.select. Among those fields, release date, original price, discounted price, and rating would be used as the for dimensions of the 4d-tree, so users can sort the games by one of these fields. I saved the game data into a dictionary, with the game id being the key. The resulting dictionary has 6654 items.

In this part, I also implemented caching. This time, I used a json file to cache the dictionary. This is part of my cached file:



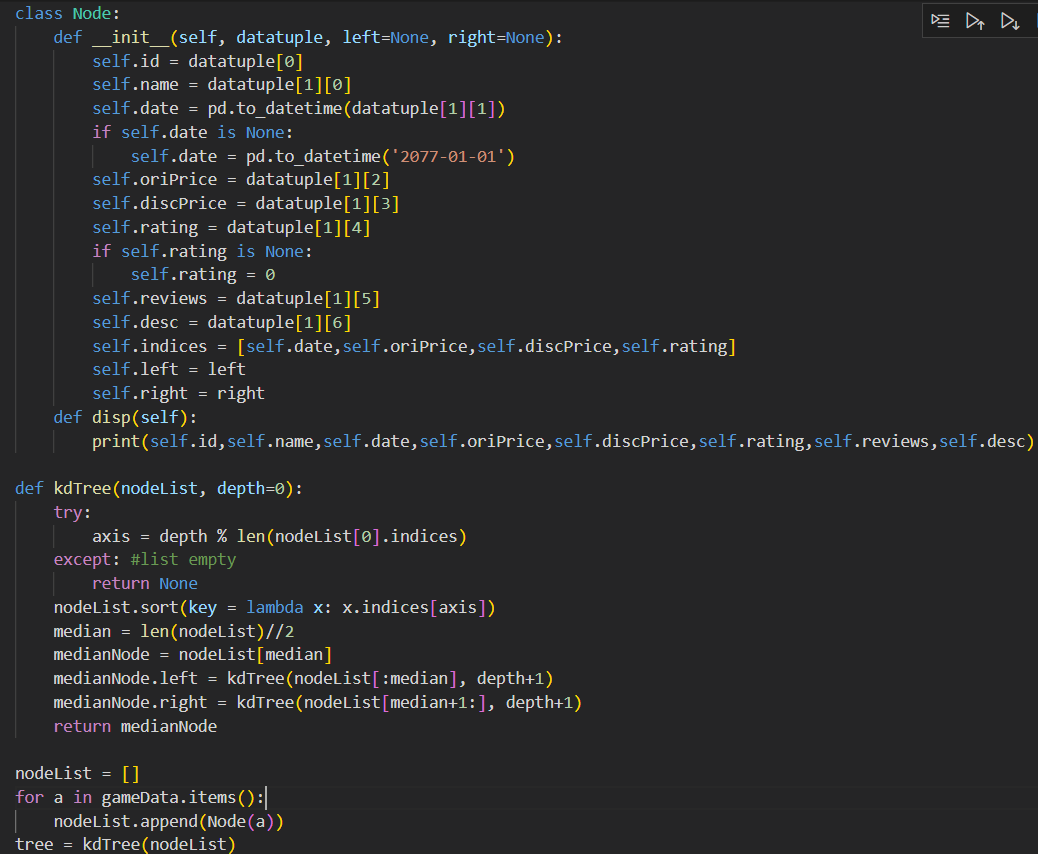
And this is a screenshot of my code that implemented caching. It took half an hour to execute the program for the first time, but it took only 0.2 seconds for the second time.





**Data Structure**

I have already implemented the 4d-tree. It is a variant of binary search tree, in which nodes are partitioned in 4 dimensions in turn as depth grows. I organized the game data into a class called Node. Other than the game data, each node has left and right child, which are set to None by default. I used 4 numeric attributes in game data: release date, original price, discounted price, and rating, as the 4 dimensions of this 4d-tree. When constructing the tree, the dimension was calculated from the depth first. Then, the nodelist was sorted in that dimension and partitioned into two halves. These two halves of nodes were fed to two recursive calls of the kdTree function and the returned results became the left and right child node. At last, the root node of the tree was returned.



**Interaction and Presentation Options**