Project 3 Report

Administrative

Team Name: Goons of Doral Academy

Team Members and GitHub user names:
Juan Esquivel - TrucoJuan
Fernando Amado-Pupo - feamado
Lucas Rodriguez - lucasrodr1010

Link to GitHub repo: https://github.com/feamado/DSAFinalProject

Link to video demo: https://www.youtube.com/watch?v=VHd5vB3dR9s

Extended and Refined Proposal:

Problem: Parsing a large CSV file with data on over 100,000 stars and organizing and presenting this data in a palatable manner such that users could easily sort by different metrics and categories, making connected graphs given a small stream of stars organized by distance.

Motivation: People who are interested in astronomy but inexperienced may turn to star visualization sites to learn more about stars, their relative location in space, and their properties. Most of these sites generally only have a few tools available for sorting by feature, and don't allow for searching for specific parameters such as luminosity. We wanted to create a program where we could easily search through hundreds of thousands of stars as well as sort them by certain properties, creating an educational and fun interactive experience for users that could potentially help them learn more about astronomy.

Features: Ability to sort using Quick sort and Heap sort by any numeric parameter in the database, longitude vs latitude graph of Stars as situated in the sky, while also showing apparent magnitude and color. Visualization of data sheets and comparing performance of both sorts.

Description of data: https://www.astronexus.com/hyg

The data set used is the HYG v3.7 database which is a 14MB .csv file with roughly 120,000 stars. This database includes all stars from the Hipparcos, Yale Bright Star, and Gliese catalogs. These stars collectively make up most, if not all, of the stars visible to the naked eye. Attributes for stars included within the data set include apparent visual magnitude, luminosity as a multiple of solar luminosity, cartesian , cartesian velocity coordinates, and its distance from Earth.

The data we collected was:
These variables are contained within the Star Class and the Heap ultimately holds stars with different variables That are compared depending on what perspective or sorting is desired by the user

- Relative cartesian coordinates to the earth
- Luminosity in units of solar luminosity or $L_{\odot}$
- CI(color_index), Unitless Logarithm of the ratio of individual intensities between the star's light and a standard
- Temperature calculated using Ballesteros's formula $T/K=4600(10.92(CI)+1.7+10.92(CI)+0.62)$.
- Distance from earth
- Calculated azimuthal and polar angles for plotting the night sky
- Apparent Visual magnitude or app_size, a unitless, logarithm of the ratio between the light projected by a star and the light blockages from cosmic dust nearby

Tools/Languages/APIs/Libraries used: For GUI: all in python using libraries tkinter, numpy, matplotlib, math, and datetime.

Algorithms implemented:

- Heap sort
- Quick sort

Additional Data Structures/Algorithms used:

- Max/Min Reheapifying Heap

Distribution of Responsibility and Roles:

Fernando Amado: In charge of backend, creating the foundation for the codebase, including the Star and Heap class, parsing data, and sorting algorithms

Juan Esquivel: In charge of GUI, graphing and interfacing with the backend APIs, as well as data processing (IE: getting the right coordinate transforms, colors, size from raw data for graphing)

Lucas Rodriguez: In charge of final report and documentation, gradually compiling information for the final report as well as assisting on implementation of backend algorithms such as quick sort

Analysis:

Any changes the group made after the proposal? The rationale behind the changes

The biggest 2 changes were the introduction of matplotlib for graphing the stars and changing the roles. Lucas and Fernando worked on back end and Juan worked on the GUI and graphing since the individual strengths of each member aligned better with these changes, and the introduction of matplotlib, as opposed to a different visualization tool again goes back to familiarity with the tool given the time constraint of the project.

For the GUI, the only operation that is not constant time is refreshing the table and calling on the quick or heap sort methods. Refreshing the table is O(n) where n is the size of the database, and the quick and heap sort methods are as described above.

__Back End Heap Methods_____

All methods depend on variables sort_mode and mode that determine whether the heap is a max or min heap respectively and what variable the heap is being "heaped" by
The key words or variables available for heaping and sorting by are:
{"lum" , "distance","temperature","ci","x","y","z","app_size"}

__quick_sort(arr)__: O(n^2) In the worst case because partitions require scanning up to O(n) elements and poor partitions can leave n steps to adjust an item leading to O(n^2). In the average case, the partitions split up an array relatively evenly which means you get logn (proportional to even partitions) times roughly n partitions leading to O(nlogn)
insert(x): O(logn) Inserting into a heap requires comparison to parent elements which essentially divide the heap elements to be compared by half during every pass. This is because parents in our 1 index heap are always at floor(i/2). Visually, this can be interpreted as scaling the tree which is of height logn because it is a binary tree, so the path upwards, counted in comparisons is at most O(logn)
pop(x) : O(logn) bottom down comparisons or "scaling" the tree of height logn downwards to determine what largest element belongs at the top after a pop. A pop returns the first element in the heap, allows it to function as a priority queue
Re_heapify and re_heapify helper functions(mode,sort_mode): All elements are compared to their parent and readjusted according to heap property, from the root to the last parent. Because we chose top to bottom, the worst case is O(nlogn) but it apparently may be improved bottom down up to O(n). mode and sort_mode variables indicate the new status of the heap, whether it is a luminosity max heap or a temperature min heap. The average case should be O(n) however as it is uncommon that every single element be flipped especially as some data such as luminosity

and temperature tends to correspond to one another. This function also changes the state of the heap from to the values of the parameters(Ex: can change sorted lum max heap to sorted temperature min heap)

Function min_re_heapify_helper is O(nlogn) as well because it is the same exact function as re_heapify but with the comparison operators flipped. It is used to create the min_heap.
Function min_pop is O(logn) like the regular pop, because it is simply the same implementation of the pop function with the comparison operators flipped, as it is used to both sort a min_heap, or pop the minimum element, depending on whether the current state of the class is a max heap or a min heap. If this function is called for a function set to sort_mode == "max" it does nothing.
Heap_Sort(mode,sort_mode): O(nlogn) our implementation of the heap_sort utilizes the preexisting heap nature in which the data is loaded in, to sort it. A heap_sort is essentially a top rank/value element extraction across the length of the original array, so our implementation merely pops(our pop returns highest value) all the elements out until the original array the heap is located in is empty, into an auxiliary array. The heap classes' heap array is then assigned to this ordered auxiliary heap array which is in the correct sorted order. It is O(nlogn) because it takes n passes through the length of the list to call the pop function

__Data_Input
_create_star_data_heap O(logn): because it simply inserts elements in a stream into a heap and as described prior, insertion in our heap is O(logn)

___Comparison__Of Algorithms_____

Quicksort was significantly slower than the preexisting heap because it relies on its partitioning methods and recursion that lead to space complexity issues and needing to fine tune our inputs to the data. We even encountered issues were the size of the data could make the program crash and we had to filter out data members that were missing elements, or provide default values that would make them easy to sort. Since the data can flip around alot in our program I believe many cases approached the worst case of O(n^2) runtime.

HeapSort in general has a faster run time than Quicksort which made loading and changing heap orders, blazingly fast in comparison to the quicksort. More importantly, the preexisting heap structure made it a seamless transition into the sort, and the UI was allowed to update pretty rapidly as a result of it. The times here, also because all of our accommodations of structures around the heapsort. Heapsort is also a consistent sort with its average case equal to its best case.

Reflection:

As a group, how was the overall experience for the project?:

Stressful, as all members of the group were very busy with other technical classes (Fernando is a CpE major and Juan is a Physics major) and it was hard to find a moment to all meet in person

Did you have any challenges? If so, describe.

Not much went particularly wrong except a few errors that were really hard to find, and hard to pinpoint. The biggest challenge for Juan was learning how to build a GUI, since this was the first time he has ever worked on a front end project

Building an API to work with the frontend, and working with so many data values was sometimes frustrating. This is because the data set was sometimes incomplete, and it was hard to test correct output values from the backend until they either crashed or displayed incorrectly in the front end. For one of our final bugs it was a group effort between pattern recognition from incorrect data sets displaying in the front end and knowledge of the structure of the code in the back end to find it.

If you were to start once again as a group, any changes you would make to the project and/or workflow?

Definitely giving ourselves more time, especially since all of us were busy in ways we couldn't have foreseen a month or two ago. With more time, we definitely would've implemented a few more features such as graphing of the night sky depending on a given position in earth, visualizing constellations and paths between stars and auxiliary features such as being able to load different data or manipulate and output csv files.

Comment on what each of the members learned through this process.

Juan: Front end basics such as how to create interaction between user and program, working with APIs and organizing menus

Lucas: Important version control basics and how to properly time manage when working with a team for a deadline

Fernando: Importance of unit testing, pandas basics, designing an API for a Front-End co-worker/team member. Responding to adjusting expectations and learning to organize code for ease of use by co-workers/team-members that may depend on it. Learned alot about team coordination.