Design Document 3: Pokedex Query Program
by: Gabriel Racz 101181470

The Pokedex Query Program is a tool used to search a .csv file populated by Pokemon for Pokemon that share the properties that the user specifies. In this version, the user can search for Pokemon by primary typing, storing the results of each query in memory. When desired, the user can choose to save the results of all previous queries to a file. The program takes advantage of multi-threading to allow the program to remain responsive, even while queries and saves are taking place in the background.

Algorithm design

To store the Pokemon data, a PokemonType struct is used with fields representing the different character and numerical data that each Pokemon has. The searching algorithm reads one line from the .csv file at a time, creating a new Pokemon with the corresponding comma seperated data. The parsePokemon() function is used to read the formatted data from the .csv translating the text into usable data in the program. To properly read the character data, the function uses several loops to read one character from the file at a time, checking with each iteration that the last character read was not a comma. Once the comma is detected, the function begins reading the data into the next Pokemon field. For numerical data, the program trivially reads in the integer values for each statistic using one formatted input string through fscanf(). The function returns a pointer to the newly created, and fully intialized, Pokemon variable. As this function reads each Pokemon one line at a time, the FILE* constantly keeps track of the current position within the .csv file.

Threads are managed using a system similar to a queue. The assignment of work to the threads is done based on the first in first out system. For searching, an array that stores 5 pthread pointers is used. For the first 5 searches, a new thread is created each time, it’s pointer being stored in the corresponding array index. After 5 searches have been conducted, the program joins the oldest thread and creates a new one at its index. The thread schedule continues in this cycle, allowing for 5 searches to be conducted simultaneously while the UI continues to remain responsive. The amount of threads can easily be changed based on the system and the performance requirements. For the provided .csv with 800 lines, even 5 searching threads is overkill. For saving, the same system is used, except there are only 3 saving threads available in the thread pool. A mutex lock is always used by both the searching and saving threads when accessing or changing the global data to avoid a deadlock or race condition.

The searchPokemon() function searches linearly through the .csv file to find matches to the user inputted type. It compares the string stored in the current Pokemon’s “type1” to the user’s input. If a match is found, the Pokemon pointer is added to a dynamically sized array storing the current thread’s matches, if not, the Pokemon pointer is freed and the loop proceeds to the next iteration. Once the entire .csv has been explored, the thread is ready to save it’s matches to the masterList containing all of the matches for the previous queries conducted by the user. The masterList is also dynamically sized to allow for an arbitrary amount of searches. A loop appends each of the thread’s matched Pokemon to the end of the masterList. A mutex guards the outside of this loop, allowing only one searching thread at a time to append to the masterList. The mutex locks the entire loop to minimize internal system calls and improve performance when compared to the unlocking and locking being done in every loop iteration. Once the masterList is updated with the current type-search, the thread frees it’s array and returns.

To save the queries to a file, a thread is created that calls the saveMaster() function. This function loops through the entirety of the masterList and prints each Pokemon's data to a user specified file. Outside of the loop, a mutex is used to ensure no other thread can access the masterList (including active searches) while the save thread writes the data to disk. As the requirements outlined in Use Case 3, any currently ongoing searches will not be included in the save. This is because either the search thread reached the mutex before the save thread and completed its search, or the save reached the mutex first and wrote to file before the search thread could save its matches to the masterList.

On exit, all active threads are joined, all dynamically allocated arrays are freed, and all files are closed. The program displays the total amount of queries conducted and the files created then exits.