4-2 Milestone Three: Enhancement Two: Algorithms and Data Structure

Dustin Morris

CS-499 Computer Science Capstone 23EW2

Prof Brooke

11/20/2023

Introduction:

The artifact I have chosen for Enhancement Two: Algorithms and Data Structure is enhancing the search algorithm of my Python application that handles data about pets in animal shelters. To enhance the search algorithm, I utilized the development environment PyCharm. My application is written using Python, which is a high-level programming language. I decided to use this application due to its large number of algorithms and programming logic already created. I originally created this application during the 22EW4 term in class CS-340 Client/Server Development.

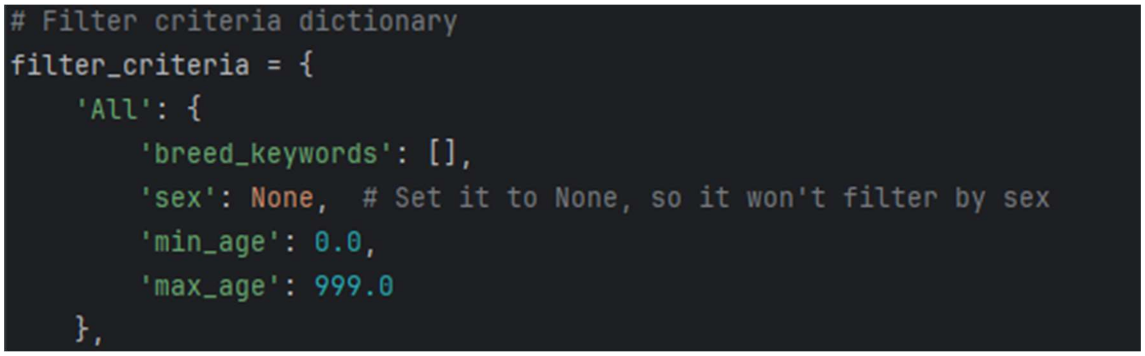
I determined this was the best project to enhance stemmed from how popular Python's programming language is, ranking as the 4th most popular language according to a 2021 Overflow survey. Python's widespread use in web and game development, areas where I excel. Along with its prevalence in emerging fields like machine learning and artificial intelligence (AI) solidified my decision. With the recent influx of AI and machine learning happening around the world, joining a team in either domain could have potential contributions to groundbreaking innovations in the future.

Overview:

Within my application I showcase a large amount of my skills using the Python programming language within PyCharm. The initial state of my application was cluttered and inefficient. To improve this application, I restructured the code within PyCharm, focusing on optimizing functions within the Data Structure. The application heavily relies on filtering to display data from a database. To address the clutter, I organized the filters outside the functions, placing them before my page layout, and following my library and document imports this enhanced readability and customization, an ideal step which is something new I learned. Another aspect of enhancing was more optimizing, and this further enhanced my search algorithms. I achieved optimization by working on changing my data structure, to do this I learned I can create a dictionary using `**filter_criteria**`, for the data inside the columns for the animal's details stored in the tables of the database such as **`Breed: `, `Sex: `, `Name: `**. In the Figure 1 is a snippet of my filter criteria for the default filtered results under the 'All' selection from a list of radio buttons. When opening the application this filter criteria will display all animals in the database table filtering only ages 0 through 999 on the dashboard.

Figure 1

Filter criteria used in data table.

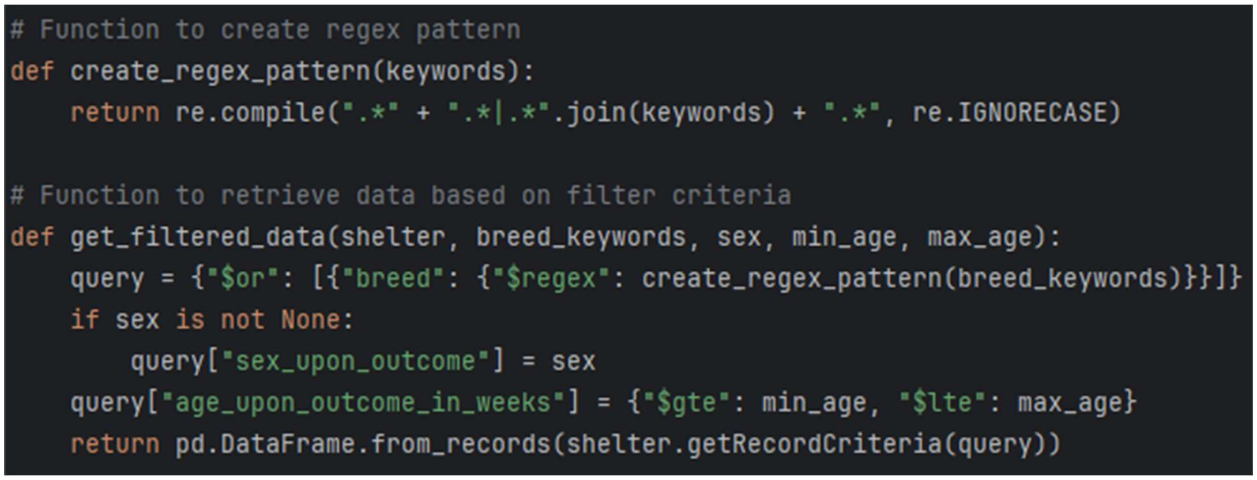


```
# Filter criteria dictionary
filter_criteria = {
    'All': {
        'breed_keywords': [],
        'sex': None,  # Set it to None, so it won't filter by sex
        'min_age': 0.0,
        'max_age': 999.0
    },
```

I list my dictionary after my libraries and imports for easier customization since it is located in one place and I can see my filter criteria before writing the code that utilizes the criteria I have in place, this plays a role in how I optimized my Python code, it prevents me having to use duplicate queries in Python, this also shortened my code making it easier to maintain and debug if any issues were to arise. Having my dictionary and filtering listed together in one place also saves time by not having to scan through different layouts and callbacks to match filtering in a centralized manner. Lastly, I created regex patterns using, `**create_regex_pattern(keywords)**`. Figure 2, my regex implementation code snippet.

Figure 2

Regex function for example query.



```python
# Function to create regex pattern
def create_regex_pattern(keywords):
    return re.compile(".*" + ".*|.*".join(keywords) + ".*", re.IGNORECASE)

# Function to retrieve data based on filter criteria
def get_filtered_data(shelter, breed_keywords, sex, min_age, max_age):
    query = {"$or": [{"breed": {"$regex": create_regex_pattern(breed_keywords)}}]}
    if sex is not None:
        query["sex_upon_outcome"] = sex
    query["age_upon_outcome_in_weeks"] = {"$gte": min_age, "$lte": max_age}
    return pd.DataFrame.from_records(shelter.getRecordCriteria(query))
```

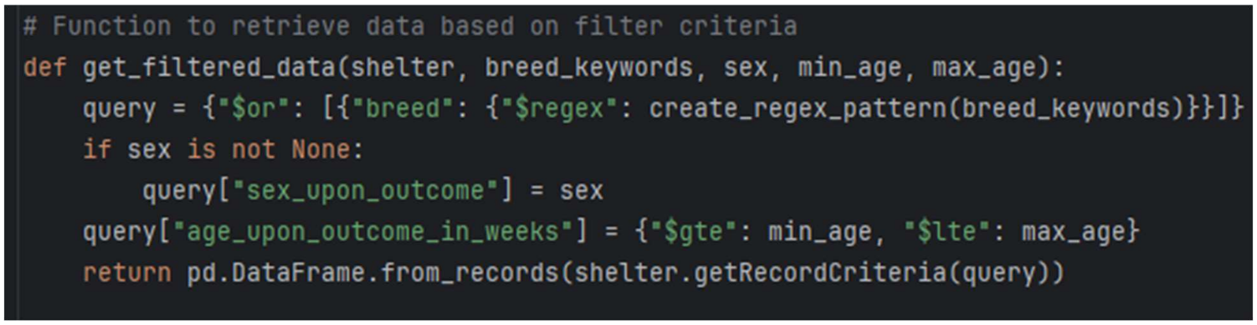
The block of code is a function that creates a regex pattern based on a list of keywords. The input parameter that you want to match is from `**keywords**`. With `**re.compile**`, this part of the function compiles the final regex pattern. I also include `**re.IGNORECASE**`: This flag is included to make the pattern case-insensitive, so it matches keywords regardless of their case. You can also see the regex function being implemented in the function, `**query = {“$or”: [{“breed”: {“$regex”: create_regex_pattern(breed_keywords)}}]}`,** this line is creating a MongoDB query using the `**$or**` operator. It specifies that the “breed” field in the database must match the regex pattern generated by the `**create_regex_pattern**`. These enhancements transformed the original mediocre search algorithms and data structure into a well-optimized, modular, and readable application by utilizing regex patterns to enable more flexible matching for filters such as “breed” using `**query = {"$or": [{"breed": {"$regex": create_regex_pattern(breed_keywords)}}]}**` which also constructs a pattern that shows partial and case-insensitive matches enhancing the flexibility of your search, this also provides a more inclusive and user-friendly search experience. The regex function further optimized my data structure and algorithms by reducing the need for redundant queries by allowing a more inclusive match and encapsulating logic of matching multiple keywords, simplifying the overall structure of my code. The overall structure of my code was simplified as using regex patterns allows you to express complex pattern matching in a concise and readable manner.

In summary, using regex functions in my code optimized the search algorithm by providing a more flexible, case-insensitive, and accurate approach to pattern matching. This cleared any redundancy and simplified code logic that contributed to streamlined data retrieval process. These optimizations collectively enhanced the readability and performance of my Python application for managing pet data in animal shelters.

For the algorithmic logic of time complexity involves examining operations that scale with the input size. My applications primary operations involve filtering and querying data from my MongoDB database and performing operations on pandas DataFrames. The time complexity of these operations is dependent on the underlying implementation of the database queries and DataFrame manipulations. In Figure 3, I use (`**shelter.getRecordCriteria(query)**`), the time complexity of these operations is regulated by the efficiency of the underlying database and pandas library.

Figure 3

Time complexity runs off efficiency in this example for the getRecordCriteria.



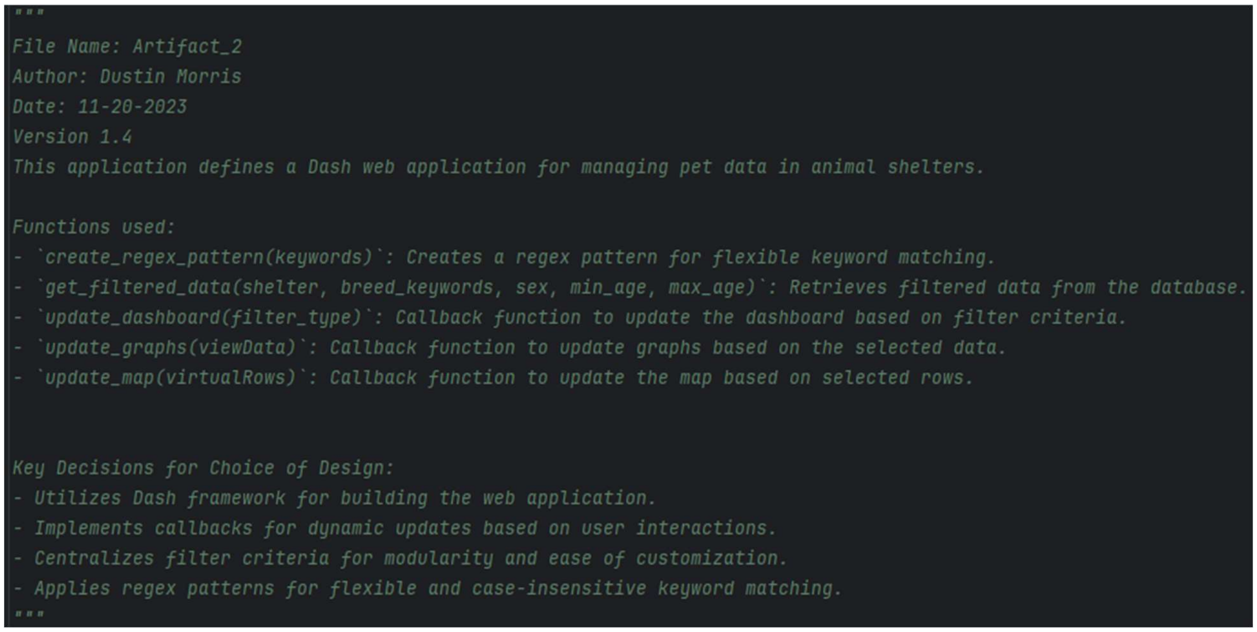
```
# Function to retrieve data based on filter criteria
def get_filtered_data(shelter, breed_keywords, sex, min_age, max_age):
    query = {"$or": [{"breed": {"$regex": create_regex_pattern(breed_keywords)}}]}
    if sex is not None:
        query["sex_upon_outcome"] = sex
    query["age_upon_outcome_in_weeks"] = {"$gte": min_age, "$lte": max_age}
    return pd.DataFrame.from_records(shelter.getRecordCriteria(query))
```

The size of the data and indexing strategy can have varying time complexities. Using `**re.compile**`, for pattern matching (`**create_regex_pattern**`), helps pre-compile the pattern, with potential performance improvement, the exact time complexity is dependent mostly on the implementation of the regex engine. The time complexity of creating a DataFrame from records is also regulated by the number of records and the complexity of them, given this information the pandas library is designed to be efficient in handling DataFrame operations which backs my decision to implement it.

In the header I added my intent and decision for the overall functionality of each node, this provides an easier ability for any programmer to edit my code. This header is included in all files in the project for my application, each files header is updated with the intent and decision based on those files current code, Figure 4.

Figure 4

Inline commenting for easy readability and understanding for future programmers to use when modifying code.

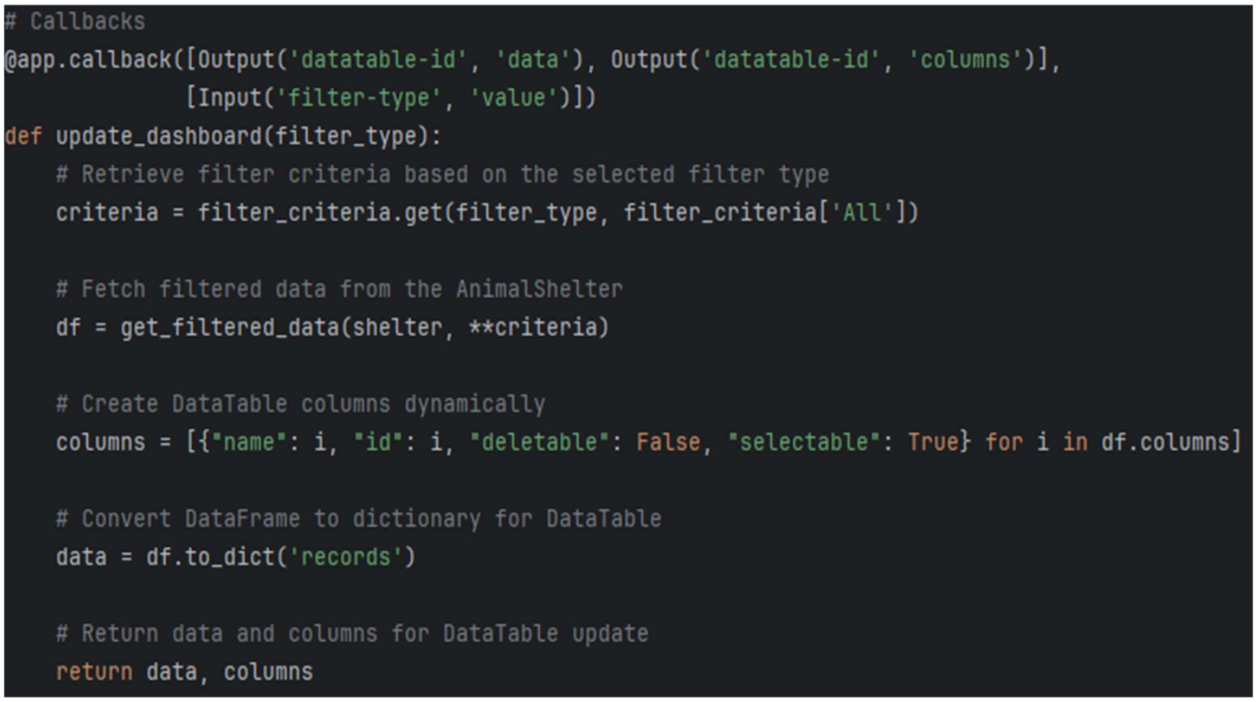


```
"""
File Name: Artifact_2
Author: Dustin Morris
Date: 11-20-2023
Version 1.4
This application defines a Dash web application for managing pet data in animal shelters.

Functions used:
- `create_regex_pattern(keywords)`: Creates a regex pattern for flexible keyword matching.
- `get_filtered_data(shelter, breed_keywords, sex, min_age, max_age)`: Retrieves filtered data from the database.
- `update_dashboard(filter_type)`: Callback function to update the dashboard based on filter criteria.
- `update_graphs(viewData)`: Callback function to update graphs based on the selected data.
- `update_map(virtualRows)`: Callback function to update the map based on selected rows.


Key Decisions for Choice of Design:
- Utilizes Dash framework for building the web application.
- Implements callbacks for dynamic updates based on user interactions.
- Centralizes filter criteria for modularity and ease of customization.
- Applies regex patterns for flexible and case-insensitive keyword matching.
"""
```

I also improved my inline commenting of the code describing its purpose and functionality, further improving readability and aiding future modifications, see below image with a snippet of my code showing an example of my commenting, Figure 5.

Figure 5

Figure 2



```
# Callbacks
@app.callback([Output('datatable-id', 'data'), Output('datatable-id', 'columns')],
              [Input('filter-type', 'value')])
def update_dashboard(filter_type):
    # Retrieve filter criteria based on the selected filter type
    criteria = filter_criteria.get(filter_type, filter_criteria['All'])

    # Fetch filtered data from the AnimalShelter
    df = get_filtered_data(shelter, **criteria)

    # Create DataTable columns dynamically
    columns = [{"name": i, "id": i, "deletable": False, "selectable": True} for i in df.columns]

    # Convert DataFrame to dictionary for DataTable
    data = df.to_dict('records')

    # Return data and columns for DataTable update
    return data, columns
```

Conclusion:

I have showcased my knowledge and skills where I design and evaluate computing solutions that solve a given problem using algorithmic principles and computer science practices and standards appropriate to its solution, while managing the trade-offs involved in design choices (data structures and algorithms), this outcome is met by my use of regex patterns in **`create_regex_pattern'**, it showcases my understanding of algorithmic principles to create efficient and flexible search algorithms. The use of the dictionary I used (**`filter_criteria`**) that manages filter criteria demonstrates my understanding of computer science practices which aligns with the concept of using data structures to manage and organize data systematically. I centralized filter criteria in a dictionary that reduces redundancy, making the core more maintainable which involves a trade-off between upfront organization and long-term maintainability, I manage the design trade-offs by consolidating logic, implementing regex patterns, and organizing filter criteria centrally, this provides a more readable, efficient and modular solution while addressing the challenges present in my original code.

My code enhancements and this narrative align with this course outcome, it demonstrates my ability to apply algorithmic principles, adhere to computer science practices, and manage trade-ffs in my design choices, showcasing an understanding of said concepts.

Additionally, I showcase where I design, develop, and deliver professional-quality oral, written, and visual communications that are coherent, technically sound, and appropriately adapted to specific audiences and contexts, this outcome is met by my coherence in communication, providing a structured narrative that follows logical flow by first introducing the enhancement project, and discussing the rationale behind my choice of the Python application, while providing the enhancements made to the code which allows readers to follow my progression. The clear explanations of my code enhancements demonstrate my strong knowledge of technical concepts. Providing code snippets explaining the purpose and impact of each enhancement, combining this with the feedback received to include inline comments and header sections add clarity to readers which demonstrates a professional approach to code documentation. I include reflective elements where I discuss my decisions and acknowledge opportunities for optimization and highlight the successful outcomes. The code snippets add visual communication to further contribute to the clarity of my communication.

In closing, the enhancements I made to the Python application for managing pet data in animal shelters represent a large step in the right direction in terms of efficiency, flexibility, and readability. When I restructured the code within PyCharm, optimized the functions in the data structure and I improved the search algorithm. Centralizing the filter criteria with a dictionary with the use of regex patterns, I streamlined the logic of the code and gave it a more maintainable and modular solution. This shows I have a strong foundation in algorithmic principles and computer science practices, not only addressed in my challenges in the original code but shows my ability to understand choices in design including trade-offs. With these enhancements to my artifact, I provide an optimized, modular, and readable application.