Research Paper - Course Selection Program

Kamden Ashmore, Garrett Kemper, Kishan Patel, Andrew Teh

University of Rhode Island

CSC 212: Data Structures and Abstractions

Jonathan Schrader

April 22, 2024

Microcosm

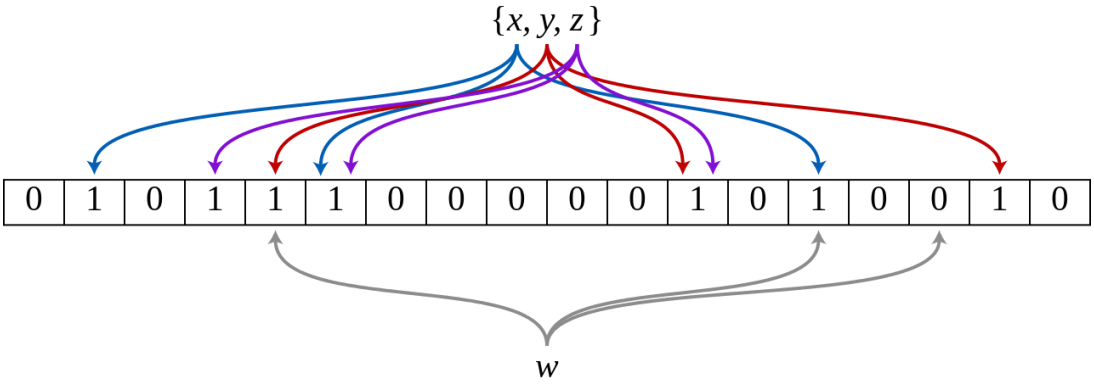
The intended audience for this program is for college or highschool guidance counselors. When planning schedules for large amounts of students, it can be very time consuming. To solve this problem, this program allows Guidance counselors to efficiently create schedules for a large number of students. This saves time that would otherwise be spent manually doing this task, allowing for a more efficient schedule system for whatever school uses it.

Data Structures and Algorithms

Bloom Filter

The Bloom Filter is a probability based data structure created by Burton Howard Bloom in 1970. The main functionality of this structure is to quickly test whether an element is in a set. Since being probabilistic, it does have the possibility of returning false positives. However, a false negative is impossible for this structure. Thus, the output of a query in a Bloom Filter is only “possibly in set” or “definitely not in set.”

A Bloom Filter consists of a bit array of length m. An empty bloom filter has a bit array of all zeros. To add an element, the element is hashed k times and the indices found through those hashes are all set to 1. To query an element, the same hash functions are used to check whether all of the indices are set to 1. If any of the indices are 0, then the element is definitely not in the set as adding it would have made those indices all 1. If all of the indices are 1, then it is possible that the element is in the set. There is a possibility that the addition of other elements may have caused those indices to be set to 1 by coincidence, hence why it is only able to say it is possibly in the set.



https://doris.apache.org/docs/1.2/data-table/index/bloomfilter/

The advantages of this structure is that it is highly space and time efficient. Using a Bloom Filter can greatly reduce the space needed for a large set as it does not actually store the elements in it. Additionally, insertions and queries can be very quick compared to a large set as it does not need to make any comparisons to other elements and only needs to compute the hash functions. As a result, Bloom Filters have a time complexity of O(k) and a space complexity of O(m), where k is the number of hash functions and m is the number of bits in the bit array. This makes Bloom Filters much more efficient than other structures such as binary trees and searching algorithms.

The main disadvantage of this structure is that it is possible to return false positives. Some level of accuracy has to be sacrificed in order to achieve higher efficiency for this structure. However, it is possible to minimize the chances of this. The probability of a false positive can be calculated using the equation $P = (1 - (1 - \frac{1}{m})^{kn})^{k}$ where P is the probability, m is the length of the bit array, k is the number of hash functions, and n is the expected number of elements. The ideal bit array length can be calculated using the equation $m = -\frac{n\,ln(P)}{(ln2)^2}$ and the ideal number of hash functions using $k = \frac{m}{n}ln(2)$ where P is your desired probability. Another disadvantage to Bloom Filters is that elements cannot be removed

from the set. If needed, there are alternate variations of Bloom Filters that can allow for the removal of elements.

The Bloom Filter was chosen for this project due to its fast query speeds and space efficiency. This project is working with large amounts of data and having speed and efficiency is crucial in order for the project to be successful. Bloom Filters were utilized in order to store the classes that each student has already passed. These were larger sets and would have taken up significantly more space if a set was used. The program also makes many checks with these classes throughout, so a fast query is needed in order to make this process quick. Additionally, the false positives are not a significant concern as the main goal of these queries is to check that a class has not been taken already. It also was easier to mitigate the false positives since the possible entries to the Bloom Filters are known and the hash functions can be designed in a way that mitigates any overlap between entries.

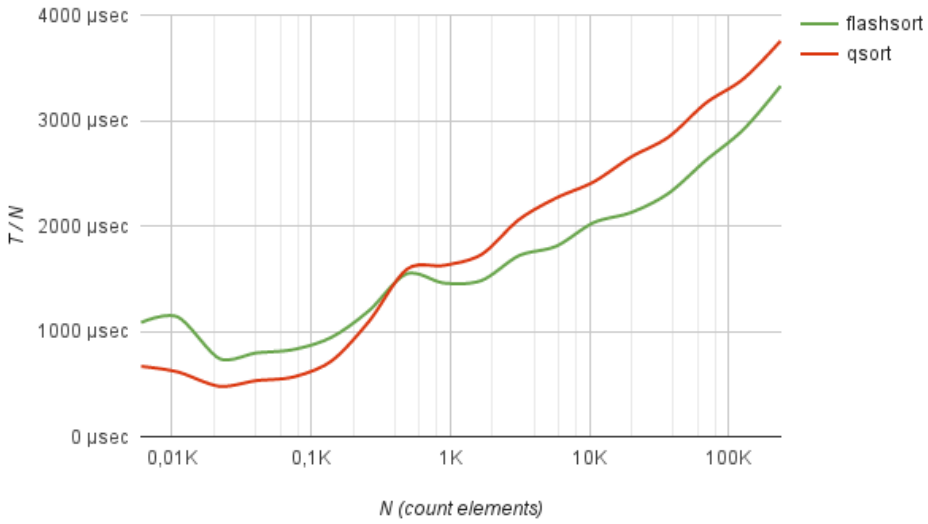
Flashsort

The flashsort algorithm was created by Karl-Dietrich Neubert in 1998. This in place sorting algorithm is very efficient with large, evenly distributed sets of data. It has a time complexity of sorting n elements in O(n) time. Traditionally the memory complexity of this algorithm is O(m) where m is the number of buckets and also m is 0.45 times n. We have altered this portion of the algorithm in our implementation and will go into further detail later in this paper.

The flashsort algorithm is built around the concept of divide and conquer. It accomplishes this by using buckets to separate similar data and sort it within those buckets. The previous steps are similar to any bucket sorting algorithm, but there are two things that separate flashsort and

bucket sort. Before we dive into that we must firstly create m buckets where m is 0.45 * n. Then each n element is put into the corresponding m bucket. Then, we must convert bucket counts to prefix sums and then rearrange elements based on prefix sums. Prefix sums are counts that help the algorithm efficiently rearrange the elements in the input array. To calculate the prefix sum let Lb represent the prefix sum for bucket b, where L0 = 0 and Lm = n, with m being the number of buckets. Lb stores the cumulative count of elements in all buckets up to and including bucket b. With this prefix sum we can rearrange the elements. To do this we would iterate through the input array. For each element Ai we determine its bucket b based on its value. Place Ai in the position j such that $L_{b-1} < j \leq L_b$. In other words, place it after all elements in the previous buckets but before the elements in the current bucket.Lastly we increment Lb to reflect the new position for the next element in the same bucket. By performing this rearrangement, we effectively group all elements belonging to the same bucket together while maintaining their relative order within each bucket. This rearrangement portion of the algorithm is a key optimization from other traditional sorting algorithms.

The pros of flashsort mainly focus on its memory efficient approach that hinges around large data sets. Additionally, the algorithm is in place, meaning that if a data point is in front of another piece of data with the same value it will remain in that same order. Flashsort being an optimization of bucket sort, another divide and conquer algorithm, focuses on sorting large data sets very efficiently. Having an average sort time of O(n), with uniformly distributed data sets, puts it ahead of a lot of other sorting algorithms.



The cons of quicksort revolve around its worst case scenario where data sets are not evenly distributed or have consistent outliers. On top of that flashsort is designed so that you would need to know the range of data you would be getting as input. Without this information flashsort will not function as intended. Finally, flashsort is a very complicated algorithm to implement so if a program doesn't fully require a flashsort it is not smart to force a flashsort implementation.

We chose flashsort in hopes that we could further improve this program to not only take in hundreds of data points, but thousands. Essentially, turning this program from a program marketed to only highschools to a program that could handle the biggest college. We are utilizing this algorithm to sort the students in ascending credit order so that we can prioritize higher credit students when picking classes for them. Additionally, the efficiency of this algorithm stuck out to us as a big potential upside.

Range Tree

A range tree is an ordered tree data that efficiently reports all points within a given range and is typically used in two or higher dimensions. Range trees were first introduced by Jon Louis Bentley in 1979 and are an alternative to the k-d tree with range trees offering faster query times but worse storage. That being said, range trees are mostly used for very efficiently solving multidimensional range queries.

A Range tree involving 1 dimension is essentially a balanced binary search tree in which at any node, the left sub-tree will have the values less than or equal to the node value and in the right tree the value is greater than the node value. When extending to higher dimensions, range trees are built recursively by constructing a balanced binary search tree on the first coordinate of the points, and then for each vertex v in the tree building a (d−1) dimensional range tree on the points contained in the subtree of v which produces a time complexity of O(n log^d n). In order to solve a range query, the tree is recursively traversed and reports the maximum value of the nodes that is within the interval of the query. This range query is very efficient with having the time complexity O(log n) where n is the number of data points.

Range trees are most effective at performing range queries in multidimensional data sets. They efficiently and quickly return a set of values within a specified range and are mostly ideal in applications such as spatial databases and computational geometry. The time complexity of a range query is given as O(log n + k) and storage of O(n*log^d-1 n), with d, indicating the dimension of the space, n being the number of points in the tree and k being the number of points retrieved for the query. Since range trees are constructed from a balanced binary search tree, this ensures efficient search, insertion, and deletion operations with the time complexity being O(logn).

The main disadvantages of the range tree are that they have a limited area of uses and that it requires lots of memory. While range trees are efficient at range queries, they may not be the best choice for other types of queries or operations such as nearest neighbor or exact match queries. Since range trees are effective at retrieving points within a certain range in a multidimensional space, therefore, other types of queries involving different types of data may be more suitable. Range trees also require a lot of storage because of the structure of the tree, especially in higher dimensions or larger datasets. Since range trees are balanced trees, additional storage is needed to perform rotations or rebalancing operations when necessary.

The range tree was chosen for this project because of its effectiveness in answering range queries. The project requires retrieving classes within a specified range based on the capacity of the class. The Range tree stores the percentage of how much the class is filled with Nodes containing percentages and names of classes. The classes with less than or equal to the percentage go to the left while the higher ones go to the right. The implementation of the range tree seemed the easiest to comprehend when trying to find what classes should be filled in first due to the class itself and the capacity of it.

Implementation

Choosing our scope

One of the first steps taken towards this project was determining the scope to be achieved. A class selection algorithm can be heavily simplified or highly complex depending on the restrictions put in place when first creating it. The goal of this was to have enough complexity for the program to be dynamic enough to be useful but also at a level that can be

implemented within the given timeframe. The following are some decisions that were made towards determining the scope of this project.

This implementation follows the 2023 Computer Science B.S. curriculum at URI. This decision was made to limit the number of classes that were included in the input and to simplify determining which classes the students need to take. The program adds classes from this curriculum to all students that are input. Realistically there are various curriculums that could be implemented, but that was not the focus of this project.

Comma-Separated Value (CSV) files are used to input the data through command line arguments. This would use two CSV files, one holding the student data and one holding the class data. These files have predetermined columns to hold data such as class names, schedule times, and classes the students have passed. CSV is a common file type for storing large tables of data and it would simplify testing this program with a large dataset. Additionally, it is reasonable to assume that schools would be able to have their class and student data stored or converted to a CSV file format if they were to adopt this program.

	A	B	C	D
1	Name	Year	Credits	Classes Taken...
2	Garrett Kemper	2	54	MTH180
3	Kamden Ashmor	3	60	CSC211
4	Kishan Patel	2	32	CSC211, MTH180
5	Andrew Teh	1	15	MTH180, CSC211
6	Jonathan Schrac	4	108	CSC211, CSC212, MTH180, CSC340
7	John Doe	1	0	
8	Jimmy Neutron	4	180	CSC106, CSC110, CSC211, CSC212, CSC301, CSC305, CSC340, CSC411, CSC412, CSC440, CSC477, CSC310, CSC372, CSC402, MTH180, MTH141, MTH142, MTH215, PH
9	Emily Capwell	4	77	WRT104, WRT227, CSC106, CSC340, CSC211, PHY112, CSC372, CSC110, PHY112, WRT104, CSC412, WRT104
10	Alexis Reynoso	4	124	CSC110, CSC440, MTH141, CSC211, CSC110, CSC310, PHY112, WRT104, MTH215, CSC110, CSC402, CSC310, CSC477, PHY111, CSC110, PHY111, CSC301, CSC106, CSC
11	Susannah Manc	2	51	CSC372, CSC110, WRT104, CSC106, CSC402, CSC106, CSC110, CSC110
12	Yehuda Slagle	1	38	PHY112, CSC340, MTH141, PHY111, CSC372, CSC477
13	Keon Caswell	1	38	WRT227, CSC440, MTH215, MTH180, WRT104, CSC310
14	Martin Bravo	2	35	PHY112, PHY111, WRT104, CSC372, CSC412
15	Gregorio Schaffe	2	48	WRT227, MTH180, CSC372, CSC310, CSC340, WRT227, CSC211, CSC211
16	Jelani Mena	3	50	CSC411, CSC402, MTH141, CSC402, MTH141, CSC212, CSC340, MTH180
17	Monserrat Hadle	1	15	CSC301, CSC310

Student Data Table

	A	B	C	D	E	F	G	H	I	J	K
1	Class Name	Session Number	Credits	Capacity	MWF/TTh	TimeStart	TimeEnd	Prereqs...			
2	CSC106	1	4	30	TTh	1400	1515				
3	CSC106	2	4	30	TTh	1530	1645				
4	CSC110	1	4	70	TTh	1530	1645	CSC106			
5	CSC211	1	4	95	TTh	1230	1345	CSC110			
6	CSC212	1	4	120	TTh	1100	1215	CSC211, MTH140			
7	CSC301	1	4	30	MWF	1000	1050	CSC212			
8	CSC301	2	4	30	TTh	800	915	CSC212			
9	CSC301	3	4	30	TTh	110	1215	CSC212			
10	CSC305	1	4	30	TTh	930	1045	CSC212			
11	CSC305	2	4	30	TTh	1100	1215	CSC212			
12	CSC305	3	4	30	TTh	800	915	CSC212			
13	CSC340	1	4	30	MWF	1200	1315	MTH180, MTH141, CSC212			
14	CSC340	2	4	30	MWF	1500	1615	MTH180, MTH141, CSC212			
15	CSC411	1	4	60	MWF	1700	1750	CSC212			
16	CSC412	1	4	30	TTh	1400	1515	CSC212			
17	CSC412	2	4	30	MWF	1600	1650	CSC212			
18	CSC440	1	4	30	TTh	930	1045	CSC212, CSC340			

Class Data Table

The input data for the students was chosen to include a name, their year, their credits, and a list of the classes they have taken. The class data was chosen to include the class name, session number, credits, capacity, whether it is Monday Wednesday Friday or Tuesday Thursday, a start time, an end time, and a list of prerequisite classes. To simplify the usage of the times, it was chosen that they would be input in military time and stored as integers. Another simplification made to the scheduling is that classes must either be Monday, Wednesday, Friday or Tuesday, Thursday. This was chosen in order to simplify the algorithm and the data to be input. Another important note is that the prerequisites are all required to take the class. Curriculums often include options such as choosing between two prerequisites or having the option to take it concurrently. This program does not take this into consideration as it would open up a lot of cases and make the data input significantly more complex.

Some restrictions on the class assignment algorithm is that it cannot assign classes that overlap to a student. Also, each student cannot take more than 19 credits. The student must also have taken all of the classes' prerequisites and the algorithm will not assign a class that the student has already passed. The goal of the program is to make sure all students have at least 12

credits. These restrictions were taken to be realistic to how the schedules work. It also gives a way to measure the effectiveness of this program.

The output of this program was chosen to be a CSV file that has a list of each student and the classes they are enrolled in. Additionally, the program will output any “errors” it encounters to the console as classes are assigned. These errors include when a class is full, low enrollment classes, and students that do not meet the minimum credits. These errors can be used by guidance counselors and administrators to know any changes that they may need to make towards the schedule of the classes. If all of the classes are full, it suggests that more sessions need to be opened up to meet the needs of the students. Low enrollment may suggest that there are too many classes or there is a significant overlap with other classes.

Determining how structures will be used

The main structures and algorithms that were used to complete this project are Flash Sort, Range Trees, and Bloom Filters. Additional structures include Sets and Vectors.

Flash Sort was implemented in order to quickly sort the large set of students based on their credits. Flash Sort is a type of bucket sort which allows for the data to be split up and sorted individually. Additionally, since Flash Sort is an in-place sorting algorithm, it is much more space efficient. This is an important factor for this project because one of the goals is to be able to input a large amount of data efficiently. Additionally, Flash Sort is more efficient with uniformly distributed data. This is also good for this implementation as the student dataset is large and relatively uniform. This implementation of Flash Sort sorts a vector of Students based on their amount of credits and divides the list into 100 buckets.

```
    // Classification phase  push students into buckets
    for (int i = 0; i < studentsSize; ++i) {
        int bucketIndex = static_cast<int>(students->at(i).getCredits() / 10);
        buckets[bucketIndex].push_back(students->at(i));
    }
```

Snippet of Implementation of Flash Sort

The Range Tree was implemented into this project as a way of storing, sorting, and querying the percentage of how filled each class was. Based on the implementation of a Binary Search Tree, the Range Tree Implementation takes in a vector of classes. Classes with low percentages of enrollment would go to the left, and classes with high percentages of enrollment would go to the right. The usage of a Range Tree allows for Range Queries which return a series of elements within given boundaries. This provides an efficient method of queuing the classes that were either filled or had low enrolment. This is useful for producing potential errors for the user to fix. The Range Queries in this implementation is intended to return two different vectors of classes. One of classes with 100% enrollment and one with classes of less than 30% enrollment.

```
void RTree::inRange(RTNode* root, float low, float high, std::vector<string>& result) {
    if (root == nullptr)
        return;

    if (root->perc_full >= low && root->perc_full <= high) {
        result.push_back(root->name);
        inRange(root->left_ptr, low, high, result);
        inRange(root->right_ptr, low, high, result);
    }
    else if (root->perc_full < low) {
        inRange(root->right_ptr, low, high, result);
    }
    else {
        inRange(root->left_ptr, low, high, result);
    }
}
```

Snippet of the Range Query in the Range Tree implementation

The Bloom Filter was implemented in order to have a space efficient method of storing the classes that the student has taken while also providing quick insertion and query times. This implementation of a Bloom Filter consists of a bit array of length 200. There are three unique hash functions that intake a string and return an integer between 0 and 199. The filter has a constructor that initializes the bit array to all zeros. It also has an insert method that takes a string and sets the bit array to 1 at the indices returned from each of the hash functions. The query method takes in a string and returns false if the bit array is zero at any of the indices given from the hash functions and returns true otherwise. This allows for a rapid query of whether an element is possibly in the set or definitely not. This also greatly reduces the size compared to storing each of these classes in a set or vector, which helps decrease the memory usage of this program.

```
79 void BloomFilter::insert(string value){
80     // cout<<value<<endl;
81     //Sets k bits to 1 based on the k unique hash functions
82     this->bits[hash1(value)]=1;
83     this->bits[hash2(value)]=1;
84     this->bits[hash3(value)]=1;
85 }
86
87 bool BloomFilter::query(string value){
88     //Checks if any of k bits are equal to 0 then value is not in the set
89     if(this->bits[hash1(value)]==0
90     || this->bits[hash2(value)]==0
91     || this->bits[hash3(value)]==0)
92         return false;    //Value is definitely not in set
93     else
94         return true;   //Value is possibly in set
95 }
```

Snippet of BloomFilter implementation

This program also uses two classes, Student and Class. The “Class” class stores all of the information of a class, including its name, session, credits, capacity, schedule, and its prerequisites. This class also stores a count of how many students are enrolled in it and has an additional method, addStudent(). This method checks if there is room to add a student to the

class. If it is able to, it increases the enrolled counter and returns true. Otherwise, it returns false. The “Student” class stores the information about each student, including their name, year, credits, and the classes they have taken. It also holds the number of credits they are currently enrolled in and a vector of Classes that they are enrolled in. The classes they have passed are stored in a Bloom Filter. Student also has additional methods such as addClassPassed() which takes in either a Class or a string and inserts its name into the Bloom Filter. It also has an enroll() method that takes in a Class and attempts to enroll the student in it. This method checks if the student has enough credits available, if the class has the capacity, if the student has passed all the prerequisites, and if the class overlaps with any other classes they are taking. If any of these tests fail, it returns false. If not, the class gets added to the Student’s vector of classes enrolled, their number of credits enrolled increases, and the method returns true.

```
17 Student::Student(string name, int _year, int credit) {
18     this->name = name;
19     this->year = _year;
20     this->credits = credit;
21     this->creditsEnrolled = 0;
22     this->classesPassed = BloomFilter();
23 }
```

```
Class::Class(string _name, int session, int credit, int maximum, bool MonWedFri, int start, int end){
    this->name = _name;
    this->sessionNum = session;
    this->credits = credit;
    this->capacity = maximum;
    this->enrolled = 0;
    this->MWF = MonWedFri;
    this->startTime = start;
    this->endTime = end;
}
```

Student.h and Class.h files

To test this program, two CSV files were created. One holds a list of classes required by the URI BS Computer Science curriculum and includes mostly accurate data based on the scheduling times of their respective classes that are being held in the Fall 2024 semester. The second file holds a list of students. Some are manually entered to provide test cases such as

students who have taken no classes and students who have taken every class available. There are then another 300 randomly generated students with randomly selected classes from the list of available classes.

Problems encountered and changes in plans

Along the process of developing this program, there were various steps that had to be reassessed as problems were encountered. Originally, the Flash Sort implementation was going to sort into only four buckets, one for each grade level. However, this greatly reduced the efficiency of the sorting method. In order to make the sorting method more efficient, the number of buckets was increased to 100. Another problem that was encountered were errors between the Class and Student classes. At one point, both classes included each other. This caused errors as it would create a loop between a Student in a Class in a Student. This was resolved by making the Class independent of Students and putting all of the enrollment checks in the Student class.

When creating the Student class, it was originally intended for their classes enrolled to be stored in a bloom filter. However, this was not possible with the current implementation of the bloom filter since it only stored the class name and not any information about the session number of the time it is scheduled for. Additionally, it would be much more harmful to have a false positive in this case as it would cause issues in scheduling. Since it was determined that a student would only enroll in a small number of classes, it was decided to use a vector instead as memory usage and search speeds would not be as harmful in a small vector.

During the planning phase of the project's creation, we settled on using a range tree to manage and query what classes each student could take. The idea was there would be a tree containing all possible classes that could be taken, with higher level classes on the right and

lower level classes on the left. However it was not until implementation in which we began to see the flaw of our original idea. The way it would have been implemented would have assumed that all classes to the left of a given class would be a prerequisite, which in reality would not be true. Additionally, it would not make sense to mix different subjects together into one range tree, as adding a qualitative element to a binary tree is not valid. If multiple range trees were created per subject, this would result in a worse time complexity than if we simply checked each class per student, linearly. As we had already declared that we were using a Range Tree for our project, our best option was to simply find a new way to implement it. This was done by using it to store and query the percentage of how filled each class. In this new implementation, after all students were assigned their classes, a range tree was created based on the classes enrollment percentages. Lower percentages were to the left and higher were to the right. Range queries were used to find lists of classes that were full (bounds from 100% to 100%), and classes less than 30% full (bounds from 0% to 30%).

Usage

The usage of this program is to provide a quick and efficient way of assigning a large list of students into their necessary classes and providing feedback towards potential issues in the current class scheduling. This program takes in two CSV files, one holding student data and one holding class data. This allows for the input to be dynamic and easy to use. The program sorts the students based on their number of credits and assigns them to classes in the Computer Science curriculum, taking account for prerequisites and schedule overlaps. The classes are assigned starting with the highest number of credits to the lowest number of credits, giving priority to students who are closer to graduation and have higher need to meet their graduation

requirements. The assigning algorithm goes through a list of possible classes that the student can and starts attempting to assign classes. The algorithm continues to add classes until there are no more available classes for the student or the student reaches the maximum number of credits (19). If the student is unable to reach the minimum number of credits (12), it will be output to the console. After all students have been assigned to classes, any classes that are full are printed to the console. Additionally, any class that has a low enrollment rate (<30%) also gets printed to the console. These messages are to help the users of this program understand the needs of the students as it helps show which classes may need more sessions or other possible scheduling issues that are causing low enrollment. Lastly, an output CSV file is created holding a list of all students with a list of all the classes they are enrolled in.

```
1 Student Name,Credits Enrolled,Classes Enrolled
2 John Doe,0,""
3 Melody Broughton,4,"PHY111-2"
```

Student Name	Credits Enrolled	Classes Enrolled
John Doe	0	
Melody Broughton	4	PHY111-2

Example of output CSV file

```
CSC301-3 is less than 30% filled
CSC440-1 is less than 30% filled
CSC340-1 is less than 30% filled
CSC340-2 is less than 30% filled
MTH215-2 is less than 30% filled
```

```
CSC106-1 is filled
CSC106-2 is filled
MTH180-1 is filled
MTH180-2 is filled
MTH180-3 is filled
```

```
Anabel Pritchard only has 7 credits
Martin Bravo only has 3 credits
Davis Yee only has 7 credits
Jimmy Neutron only has 0 credits
```

Example of error outputs in the console

This program helps resolve the problem of providing an efficient and quick method of assigning a large set of students to schedules that fit their needs. The usage of CSV files allows for the program to be used dynamically with existing class scheduling programs. This program allows for the guidance counselors and school administrators to quickly comb through a large number of students and greatly reduces their workload of assigning students to classes. This program can also be built off of to support more conditions for schedules and fill the needs of more schools.

Use Cases

This program is designed specifically for guidance counselors to use for their students. This could be used by computer science guidance counselors at URI that are responsible for less than the entire student population. It would help them efficiently provide schedules and see problematic classes in terms of lack of enrollment. With some more development time this program could be expanded to schools across the country. Additionally, this should free up time for them to hopefully do more than just provide schedules and help with scheduling classes. Our hope is that guidance counselors can be more involved with their students. Turning them from a single use case that feels like they don't even have time for you, into what they were designed to be. A catch all, helping to point the way for students in need of any type of assistance.

Strengths

This program was made specifically focusing on the curriculum sheet of the bachelor of science computer science major at the University of Rhode Island. However, since the program reads in csv files for the input, the program can be used for other majors with other curriculum sheets as well. This allows the program to be scalable for other uses past our focus on computer

science B.S students with just a few changes to the implementation. The data structures that this program uses allow for very efficient outputs and memory usage when using our dataset of 300 random students up to 3000 students. The bloom filter data structure accounts for previous classes students have taken because of its faster query times and space efficiency compared to if a set was used instead. The flash sort data structure effectively sorts the students in ascending credit order so that we can prioritize higher credit students and is an in-place structure that is memory efficient as well. The range tree was effectively used in order to retrieve classes within a specified range based on the capacity of the class which allows for efficient range queries especially in our case using large datasets.

The output of the data is clear and produces a csv file regarding the student's name, have many credits they have newly been enrolled in, and also the classes that they have been newly enrolled into. The output is easy to read and provides all the necessary information for a student enrolling into classes. The output can then be used as a guideline or a first step in students enrolling into courses based on the focused curriculum sheet.

Weaknesses

Despite the many advantages, there are of course some shortcomings of this program. First of all, there is the fact that this program only acts as a guide to building schedules for students, and does not take into account the many preferences of the students, such as the amount of workload they want to take on that semester, the professors they would prefer, or what times works best for them to take classes. Moreover, this program does not account for classes that require concurrent enrollment. For example, this program may assign a student to PHY111, despite the fact that in reality, you can not sign up for this class without taking PHY185 during the same semester. To simplify the creation of the project, the day slots of all classes were either

on a Monday, Wednesday, and Friday schedule, or a Tuesday and Thursday schedule. While this is applicable to most classes, there are still a sizable amount of classes that don't follow this schedule, including labs, asynchronous, and hybrid classes. In addition, the program assumes that all students enrolled are full-time students, preventing this program from being applicable to part-time and students with disabilities.

Limitations

Considering the scope of this project, there were many instances in which we encountered challenges and trade-offs that influenced the program's design and functionality. The program relies entirely on the provided data being in the form of CSV files, which would require students and class data to be converted from the format it is stored in by the school to this rudimentary data format. Furthermore, the program can only be run from a command line interface, which while suitable for the context of a project, is inconvenient and requires a learning curve for the targeted audience of this program. It is likely that a typical guidance counselor would not be comfortable with the interface as it is right now. It is also probable that this program can not be properly extrapolated to datasets larger than ten thousand students. That is the program is best suited for smaller institutions or individual majors within colleges.

The Future

The future of this program relies on us to improve the scope. This could involve us adding a GUI that users could utilize to change credit requirements, and add other majors with their class requirements. This would be required if the program ever went public. Without this every user would have to have an understanding of coding to change the program to incorporate their specific school requirements. Currently the program is limited to bachelors of science

computer science students at URI and has gaps that need filling as well. Having a way for users to add their own schools specific requirements would help expand this program into something that would be essential to most schools. Additionally, a graphic of a specific student's classes could be output to help guidance counselors provide a quick image to students wanting to visually see their schedule in order to plan out their days. As of now classes cannot be taken away from students without rerunning the program. This could pose a problem if a student fails a class. We could improve this by adding a way for users to take classes away from their students if the case arises.

References

BasuMallick, C. T. (2023, October 8). *Bloom filter working, functions, and applications: Spiceworks - Spiceworks*. Spiceworks Inc. https://www.spiceworks.com/tech/big-data/articles/what-is-a-bloom-filter/

Apache Doris. *Bloomfilter index - apache doris*. Apache Doris RSS. (n.d.). https://doris.apache.org/docs/1.2/data-table/index/bloomfilter/

GeeksforGeeks. (2024, April 3). *Bloom filters - introduction and implementation*. https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/

GeeksForGeeks. (Updated September 13, 2023). *Exploring Range Trees*. Geeks For Geeks. https://www.geeksforgeeks.org/exploring-range-trees/

Topa, K. (2027, Jan 1). *The fastest sort method - flashsort*. Steemit. https://steemit.com/popularscience/@krishtopa/the-fastest-sort-method-flashsort

Rauf Tabassam *What is flash sort?*. Educative. (n.d.). https://www.educative.io/answers/what-is-flash-sort