Aren Wells Project Report

Approach

For this project, I started out by setting up all the structures I needed to keep track of all the data presented in the input file. My goal was to create structures that would allow for easy use of a modified Gale-Shapley algorithm, which I will discuss in detail later on. Ultimately, I decided on using a linked list to store the all the departments, where each node, a depNode, in the list contains the index, name, number of vacancies, pointer to the next depNode in the list, and a pointer to a preference List of that department. To store all applicants, I decided to use a linked list as well, where each node, an appNode, contains the index, name, current assignment, pointer to next appNode in the list, and a pointer to a preference list of that applicant. The use of a linked list to store my departments and applicants allowed for easy insertion, searching, and printing. Beyond this, I now needed to figure out how to store the preference list of each applicant or department, inside a depNode or appNode. My solution to this was to implement the preference lists of each applicant and department as a queue. This queue, called a prefQ, is made up of two pointers to prefQNodes, a front and rear, which contain a name and a pointer to the next prefQNode in the prefQ. This would allow for easily accessing the things at the top of the preference list and checking if a list is empty, which is crucial for use of a modified Gale-Shapley to find well-formed pairs. At this point, I had set up the structures needed to store all the data about the department and applicants from the input file. Next, I needed to decide how I was going to store the actual pairings of job applicants to departmental vacancies. For this, I decided to use a graph, called a pairGraph. A pairGraph has a number of vertices and a pointer to an array of pairLists. This is essentially holding the adjacency list of each vertex in the graph. A pairList node contains a vertex, which holds a name of an applicant, and a pointer to the next pairListNode in the list. I decided I could create a graph with V vertices, where V is the number of departments with vacancies. I could then, when an assignment needs to be made, just add an "edge" to the given department's pairGraph, by inserting the applicant to be assigned to that department, to the array storing that department's pairList in the pairGraph. Thus in the end, each pairList contains the names of the applicants assigned to that department.

Reading in the input file

My approach for reading in the input file is pretty straightforward. Since we are given all the specifications for the input file and what should be in a correct one, essentially I just looped through the file line by line, verifying that everything in the input is formatted correctly. I use getDepsAndApps to read in my file. This simply creates department lists and applicant lists, and a preference list for each applicant and department upon encountering these things in the input file. If any part of the input file is not formatted correctly, an error is returned, telling the user what went wrong and on what line it went wrong on. After our getDepsAndApps has ran, we should have a depList, where each department in the list has an index, a name, number of vacancies, and a preference list of applicants. We should also have an appList, where each applicant in the list has an index, a name, and what department they are currently assigned to. GetDepsAndApps also returns an integer value representing the number of departments, which we will use to create our pairGraph.

Dep Lists and App Lists Algorithms and Complexities

Algorithms for operations on depNode's and appNode's are basically one in the same. Before making a node, upon encountering the name of the department or applicant, we must first verify that it is a valid name. To do this we call isValidDepName/isValidAppName. Given a string represented by a

character pointer, we iterate through each character, checking to see if it satisfies certain criteria. A dep name must only contain letters from English alphabet, underscores, and spaces. An applicant name can only contain letters and spaces. If either of these functions encounter a character that doesn't meet the specified criteria, 0 is returned. Otherwise we assume the name is valid and 1 is returned. This will take at most O(n) time where n is the number of characters in the string representing a name. Nodes get created and added to the linked lists by calling insertApp/insertDep. This will recursively enter a new node onto the end of the list by first checking if head is empty. If so we malloc space for the new node and initialize it with the given parameters. Otherwise, we recursively call the insertDep/insertApp function on the next item in the list. This will be done until a node pointing to null is found, representing the end of the list, meaning we can now malloc space for a new node and initialize it, adding it at the end of the list. These functions will return a pointer to the head of the list. Dep lists and app lists both also support findDepByName/findAppByName and findDepByIndex/findAppByIndex. These functions, given the head of a list and a name/index to be found, will simply iterate though each node in the list to find the node with the given value. It will return a pointer to the node containing the value, or null if the value is not in any node in the list. Note that so far, all dep list and app list functions have a complexity of O(n), because we encounter each node in the list at most once. To print out our departments and applicants I use printDepsAndApps. Given a pointer to the depHead and appHead (heads on dep List and app List), we simply loop through each node until the end, printing its index, name, and vacancies, followed by its prefList, which uses printQueue to display all the names. The same thing is then done for each node in the app List, except we only have the index, name, and preference list to print. This function will take O(n+m) time where n is the number of depNodes in the dep list, and m is the number of appNodes in the app list. Later on in our implementation of a modified Gale-Shapley, we must select a department who still have open vacancies and hasn't offered a position to all applicants. We do this with the findDep function. Given a pointer to the depHead, we loop through the list until the end, and if we encounter a node who still has open vacancies and whose preference list is not empty, return a pointe to that node. Otherwise we know no such node exists in the dep List and we return null. This will take at most O(n) time where n is the number of depNodes in the linked list. After we have ran the program and found our well-formed pairings, we must free, or de-allocate, all space associated with depLists and appLists. We can do this using freeDepNodes/freeAppNodes. Given the head depNode or head appNode respectively, these functions will simply loop through every node in the depList or appList and free the allocated space. This takes at most O(n) time because we encounter each node in the list only once when we free them. As you can see, most algorithms performing operations on dep lists and app lists, run in linear time.

Preference Queue Algorithms and Complexities

When we first create our depNodes and appNodes, we initialize their priority queues. This can be done using initQueue, which has O(1) complexity. This mallocs space for a new prefQ, and sets the prefQ's front and rear pointers to null. A pointer to this prefQ is returned. Our prefQ supports basic queue operations such as enque, deque, isEmpty, and printQueue. Enque adds a new node by calling newNode, which mallocs space and initialize for a prefQNode and returns a pointer to this node. Enque then loops through the prefQ to find the correct spot in the queue for the node. This takes at most O(n) time, where n is number of nodes in the prefQ. Deque simply returns the name store in the prefQNode on the front of the prefQ list, and then frees that prefQNode. This takes O(1) time. IsEmpty, O(1) complexity, simply checks to see if the prefQ's front pointer is null, if so then the prefQ is empty. To print the contents of out prefQ, which we do when we printDepsAndApps, we loop through the prefQ starting at the front, until the front and rear are the same, printing the name stored in each node. However, if front and rear are both null, then our queue is empty. This takes at most O(n) time, where n

is number of prefQNodes in the prefQ. Finally, there is checkPref. This function is used in our implementation of Gale-Shapley to see if one string occurs before another in a given prefList. It simply loops through the passed in prefList until temp and rear are equal, and if it encounter curr before poten, representing curr assignment and potential assignment, then it will return 1. Otherwise poten is higher in the prefList and 0 is returned. Complexity of at most O(n), where n is number of prefListNodes in the prefList. Like our depLists and appLists, we must also free all allocated space after we use our prefLists. To do this we use freeQueue, which is called on each iteration of freeDepNodes/freeAppNodes to free its entire prefQueue as well. This takes O(n) time where n is number of nodes in pref queue, because we simply loop through each node in the list once, freeing it when we encounter it.

Pair Graph Algorithms and Complexities

When we create our pairGraph, we use the function createPairGraph. Given the number of vertices, V, which is our number of departments (which we know because it gets returned from our function that parses the input file), and a pointer to the depHead, we malloc space for a new graph, initialize its vertices, and malloc space for the array of pairLists for V pairLists. Then we iterate through each index in the array of pairLists, and find the department whose index corresponds to the index in the array using findDepByIndex. We can then set the head of the pairList at that index in the array equal to a new pair list node with the department's name. This can be done by calling newPairListNode, given a string representing a name. NewPairListNode mallocs space for a new pairListNode and initializes its vertex to the given name. It then returns a pointer to a new pairListNode, complexity is O(1). After this, we should have a pairGraph with V vertices, and the head of each pairList in the array of pairList's should be the name of a department. CreatePairGraph returns a pointer to the newly created pairGraph. Its complexity is at most O(V), because when we create it our graph only has the V vertices, and no other edges, E. To add an edge to our graph, or add a "pair", we use addPair. This function, given a pointer to our graph, a string representing a department, a string representing an applicant, and a pointer to the head of the depList, assigns the given applicant to the end the of given department's pairList. To do this we call findDepByName with the depHead and department name, which gives us the depNode of the department with that name. We can use this depNodes index to find the index in the array of pairLists corresponding to this department's pairList since our depNode indexes are the same as their index in the array. Now that we have that, we can loop to the end of the pair list at the index in the array corresponding to the department, and add a newPairListNode with the applicant's name to the pairList. This will represent a pairing between the department and the applicant, since the applicant is now in the department's pairList. This function takes at most O(n + m), where n is the number of nodes in the depList pointed to by depHead, and m is the number of items already in the department's pairList. Deleting a pair from the graph takes the same amount of time and is essentially the same as adding, except instead of looping through the pairList at the index in the array corresponding to the department's index until the end, we loop through until we find a pairListNode in the pairList whose vertex is equal to the applicant we want to delete. When we find this node, we free it and return. To print the graph, we simply call printPairGraph. This will, given a pointer to the graph, print out a representation of the applicants assigned to each department. It does this by iterating through each index in the array of pairLists, and printing out the head of the list as the department, and then looping through the rest of the pairList, printing out the names of the applicants assigned that department. This will take O(V + E) time, where V is the number of vertices in the graph, and E is the number of edges. Just like our depLists, appLists, and prefQueue's, we must free all allocated space for our pairGraph as well. To do this we use freePairGraph, which simply loops through each vertex in the graph, and going through its pairList array, freeing each node in the array. Finally after freeing all nodes in our pairList array we can free the graph itself.

Implementation of Modified Gale-Shapley to assign applicants to vacancies. Algorithm and Complexities

Using functions and other things previously stated, I will now walk through my implementation of a Gale-Shapley algorithm, which guarantees that at termination every vacancy in every department gets assigned, and all pairings are well formed and stable. I implement this function using GSfindPairs. Initially, all applicants in the applicant list have a value of null in for assignedTo, meaning they are all unassigned. Also, all available positions at the departments in the department list are open. So using findDep, while there is a depNode in the department list who has vacancies greater than 0, meaning it has at least one open position, and this depNode's preference queue is not empty, check using isEmpty, meaning it hasn't offered a position to all applicants, select such a department, D. We deque D's prefList, and call findAppByName on the result to find the highest rated applicant whom D hasn't yet offered a job, let's call this applicant A. If A's assignedTo value is null, then this applicant hasn't yet been assigned to a department. If this is the case, we will call addPair to assign A to D. Next, we will decrease D's vacancies by 1, and changed A's assignedTo value to the name of D. Otherwise, if A's assignedTo is not null, then we know A is already assigned to some department, AssgnD. By using checkPref, we check to see if the applicant prefers D to AssgnD. If so, then we call deletePair to delete A from AssgnD's prefList, increment AssgnD's vacancies by 1, and change A's assignedTo back to null. Then use addPair to assign A to D's prefList, decrement D's vacancies by 1, and change A's assignedTo value to the name of D. We simply repeat this process until there are no more departments with open vacancies who haven't offered a position to every applicant. At this point we can return a pointer to the pairGraph, whose array of pairList's contains our well-formed pairings. These results are printed using printPairGraph. This algorithm terminates after at most |D|*|A| iterations of the while loop, meaning it has a complexity of O(|D||A|).

Logic behind Driver function

The driver itself is somewhat straight forward. First we check the command line arguments to make sure the user is running the program correctly. After this, we simply open our input file, and create an empty depNode head and appNode head, which we will send to our getDepsAndApps, where they will get initialized and populated. At this point we close the input file, and if the user only wants to see a listing of the departments and applicants, we print them out and exit the program. Otherwise, we create our pairGraph using the createPairGraph function, and then send it, along with the depHead and appHead to the GSfindPairs, which will run the modified gale-shapley algorithm to get us our well-formed pairs. At this point we simply print out the results to be displayed to the user, and then free all of our allocated space using freeDepNodes, freeAppNodes, and freePairGraph.

No group participation to critique, I was the only member. I think I did well though ☺