Final Project

Josh Seligman
joshua.seligman1@marist.edu

October 27, 2022

1 Hospitals and Residents Stable Matching Problem

1.1 The Algorithm

In the hospitals and residents stable matching problem, the goal is to assign residents to hospitals given the preferences of both sides so that all assignments are stable. In this context, the term "stability" means that for each resident, there is no hospital that is available that is higher on a resident's list compared to that resident's current assignment. The reason stability is in the terms of the residents is because the residents propose to the hospitals on their preference lists and the hospitals have the ability to either provisionally accept or reject the residents based on their resident preferences and current capacity. In other words, hospitals may have a preferred resident that is available, but that resident may not want to go to that hospital and, therefore, will end up elsewhere.

1.2 Asymptotic Analysis

The pseudocode for the hospitals and residents stable matching algorithm is provided in Algorithm 1. The algorithm starts off by assigning all residents and hospitals to be completely free, which are $O(r)$ and $O(h)$ operations, respectively. Next, line 8 is the condition for a while loop that runs until the residents list is empty. Since residents are being picked off one-by-one as done in line 9, the loop will run for each resident, which makes it run on average r times. Line 10, similar to line 8, also defines a while loop. This time, however, it is iterating over the resident's preferences, which means the while loop runs h times for each iteration of the outer loop. The if-statement block on lines 12-16 is inside of the inner loop, and contains a statement to get the least preferred assigned resident, which is an $O(r)$ operation as it may have to loop through all of the residents at worst case. The condition and other 2 assignments in the if-statement block are constant time assignments. The assignment on line 17 sets the assignment for the resident, which runs in constant time. Next, just like line 12, the check if the hospital is full a constant time check. Line 19 is the same as line 13 and is an $O(r)$ operation. Next, there is a loop defined on line 20 that iterates through the remaining residents in the list of hospital preferences, which at worst case is about r iterations. It also has an $O(r)$ operation that is executed once to get the starting index. Lastly, there is a constant time assignment on line 21, and the removal function calls are $O(h)$ and $O(r)$ operations, respectively, as the hospitals and residents are being iterated through in each call. Overall, when putting it all together, the runtime of the original stable matching problem for residents and hospitals is $r*h*(r+r+r*(h+r)) = rh*(2r+rh+r^2) = 2r^2h+r^2h^2+r^3h = O(r^2h+r^2h^2+r^3h)$.

Algorithm 1 Hospitals and Residents Stable Matching Algorithm

```
procedure StableMatchOriginal(residents, hospitals)
    for r of residents do
        r.assignment ← null    // Residents start off unassigned
    end for
    for h of hospitals do
        h.assignments ← [ ]    // Hospitals initially have no assignments
    end for
    while !residents.isEmpty() do
        r ← residents.dequeue()    // Get the next resident in line to be assigned
        while r.assignment == null && !r.preferences.isEmpty() do
            h ← r.preferences.dequeue()    // Try the resident's next top preference
            if h.isFull() then
                r′ ← h.getLeastPreferredAssignedResident()
                r′.assignment ← null    // Set the least preferred assigned resident to be free
                residents.enqueue(r)    // Add the resident back to the list to be reassigned
            end if
            r.assignment ← h    // Provisionally assign r to h
            if h.isFull() then
                s ← h.getLeastPreferredAssignedResident()
                for i ← h.preferences.indexOf(s) + 1, len(h.preferences) − 1 do
                    s′ ← h.preferences[i]
                    s′.preferences.remove(h)    // Remove h from preferences of s′
                    h.preferences.remove(s′)    // Remove s′ from preferences of h
                end for
            end if
        end while
    end while
end procedure
```

This can be simplified to $O(r^3)$ because there are typically a lot more residents than hospitals, so r^3 becomes the dominant term in the expression.

2 Variation of Hospitals and Residents Stable Matching Problem

2.1 The Algorithm

One variation of the hospitals and residents stable matching problem is when only the residents rank the hospitals and the hospitals do not rank the residents. When making the assignments, stability is defined by not having a resident who is assigned to one hospital that would rather be in another hospital that is still available. Similar to the original algorithm, hospital capacity would be nice to have, but is not a necessity if not enough residents prefer the hospital to meet its capacity. In other words, overall resident happiness is the most important variable to consider when making the assignments. As displayed in Algorithm 2, the method needed to make the assignments requires a greedy approach and taking the first possible outcome that works for the data. It is for this reason that each resident is initially assigned to their top choice hospital and then gets trimmed down to meet the respective capacity of each hospital.

2.2 Asymptotic Analysis

Listing 2 provides the C++ implementation of the variation algorithm to the stable matching problem. First, lines 4-9 include a for-loop that assigns each resident to their top choice hospital, which is an $O(r)$ operation since the loop is iterating over each resident. Next, lines 16 and 17 define nested loops that iterate over each hospital for each level of assignments the hospitals have. Since the number of levels is a constant factor, the loop is just an $O(h)$ loop that is just being executed more than once. The while loop defined on line 19 contains a check to see if the hospital is over capacity in the given range, and the condition runs in constant time because both functions being called are constant time operations. However, based on the condition, the loop will run until the hospital is no longer over capacity, which means that enough residents have to be removed to meet the condition, making the loop an $O(r)$ loop. Lines 21 and 22 are both constant time dequeue operations. Inside the if-statement though, there is a call to the *priorityAdd* function, which is an $O(r)$ function that has to iterate over the linked list to find the appropriate spot to place the element. Therefore, the nested loops on lines 16-33 run in $O(h*r*r) = O(hr^2)$. Since there are typically more residents than hospitals, the r^2 term dominates the expression and makes the loop run in $O(r^2)$ time. Next, the loops on lines 36-44 iterates through each hospital and assigns the appropriate residents to that hospital, which is the same thing as saying that each resident is formally assigned to a hospital. This is an $O(r)$ operation since each resident has to be considered. The last component of the algorithm is the swapping mechanism, which is defined on lines 68-145. First, the while-loop defined on line 69 continues until no swaps were made in the previous iteration. As the objective of the greedy algorithm was to get as high of a resident happiness score as possible, the number of times this loop runs should be minimal, which is why it is being classified as an added coefficient to the performance of the loop's body. Next, the nested for-loops defined on lines 72 and 73 set up each resident to be compared to the other residents, which is an $O(r^2)$ operation as the residents array has to be gone through for each pass of the outer loop. The entire body of the nested for-loops consists entirely of comparisons and assignments, which all runs in constant time. Therefore, the functionality of swapping elements runs in $O(r^2)$ time. Overall, when combining each component of the algorithm, the runtime for the variation algorithm is $r + r^2 + r^2 = r + 2r^2$ is $O(r^2)$.

3 Appendix

3.1 Original Algorithm

Algorithm 2 Hospitals and Residents Stable Matching Algorithm

```
procedure StableMatchVariation(residents, hospitals)
    for r of residents do
        topHosp ← r.preferences.head
        r.assignment ← topHosp    // Residents start with their top choice regardless of capacity
    end for
    for i ← 0, NUM_LEVELS − 1 do    // Have to check at all levels
        for j ← 0, len(hospitals) − 1 do    // Have to iterate through each hospital
            while hospitals[j].numAssigned(0, i) > hospitals[j].capacity do    // Continue until the hospital is no longer over capacity at the given level
                res ← hospitals[j].assignments[i].dequeue()    // Get the resident being removed
                res.preferences.dequeue()    // Remove the hospital from the resident's preferences
                if i < NUM_LEVELS − 1 then
                    newTopHospital ← res.preferences.head
                    newTopHospital.assignments[i + 1].addres    // Add the resident to their next top choice
                end if
            end while
        end for
    end for
    for i ← 0, len(hospitals) − 1 do
        for j ← 0, NUM_LEVELS − 1 do    // Iterate through all the residents assigned to the hospital
            cur ← hospitals[i].assignments[j].head
            while cur ! = null do
                cur.assignment ← hospitals[i]    // Formally assign the resident to the hospital
                cur ← cur.next
            end while
        end for
    end for
    swaps ← −1    // Initialize to -1 to enter the loop
    while swaps ! = 0 do
        swaps ← 0    // Start off the iteration with 0 swaps
        for i ← 0, len(residents) − 1 do
            for j ← i + 1, len(residents) − 1 do    // Compare all residents to all residents
                iHosp ← residents[i].assignment    // Get the current assignments
                jHosp ← residents[j].assignment
                canSwap ← true    // Assume we can swap and end up with a stable pairing
                if residents[i].rankings[jHosp.index] ! = 0 then
                    canSwap ← false    // Cannot swap because resident i didn't rank jHosp
                else if residents[j].rankings[iHosp.index] ! = 0 then
                    canSwap ← false    // Same but for resident j
                end if
                if canSwap then
                    curHappiness ← avg(residents[i].rankings[iHosp.index], residents[j].rankings[jHosp.index])
                    swapHappiness ← avg(residents[i].rankings[jHosp.index], residents[j].rankings[iHosp.index])
                    if swapHappiness > curHappiness then    // Swapping would result in a better net happiness
                        iHosp.remove(residents[i])    // Remove the residents from the hopsitals
                        jHosp.remove(residents[j])
                        iHosp.add(residents[j])    // Swap the residents and the hospitals they are assigned to
                        residents[j].assignment ← iHosp
                        jHosp.add(residents[i])
                        residents[i].assignment ← jHosp
                        swaps ← swaps + 1    // Increment the number of swaps made
                    end if
                end if
            end for
        end for
    end while
end procedure
```

```cpp
void generateStableMatches(ResidentArr* residents, HospitalArr* hospitals) {
    Queue<Resident*> residentQueue;

    // Add the residents to a queue for determining the next resident to propose
    for (int i = 0; i < residents->length; i++) {
        Node<Resident*>* n = new Node(&residents->arr[i]);
        residentQueue.enqueue(n);
    }

    std::cout << std::endl;

    while (!residentQueue.isEmpty()) {
        residentQueue.printQueue();

        Node<Resident*>* resident = residentQueue.dequeue();

        while (resident->data->getAssignment() == nullptr && !resident->data->getHospitalPreferences()->isEmpty()) {
            for (int j = 0; j < hospitals->length; j++) {
                std::cout << resident->data->getPreferencesArr()[j] << " ";
            }
            std::cout << std::endl;

            // Get the hospital at the front of the preference list
            Node<Hospital*>* preference = resident->data->getHospitalPreferences()->dequeue();

            // The hospital is full with more preferable residents, so the current resident is not getting in
            if (resident->data->getPreferencesArr()[preference->data->getIndex()] == 0) {
                // Clean up memory and try the next hospital
                delete preference;
                continue;
            }

            if (preference->data->isFull()) {
                // Get the lowest preferred assigned resident index
                preference->data->setLowestPreferredAssignedResidentIndex();

                // If the hospital is full, then replace the lowest preferred resident with the current resident
                Resident* lowest = &residents->arr[preference->data->getAssignedResidents()[preference->data->getLowestPreferredAssignedResidentIndex()].getIndex()];
                lowest->setAssignment(nullptr);

                preference->data->replaceLowest(resident->data);
                // Since the lowest preferred resident was replaced, we need to come back to it later
                residentQueue.enqueue(new Node<Resident*>(lowest));
            } else {
                // Add the new resident to the
                preference->data->addResident(resident->data);
            }

            // Print the assignment as it happens
            std::cout << "Assigned " << resident->data->getName() << " to " << resident->data->getAssignment()->getName() << std::endl;

            if (preference->data->isFull()) {
                // Get the lowest preferred assigned resident index
                preference->data->setLowestPreferredAssignedResidentIndex();

                // We need to iterate through all of the residents
                for (int i = 0; i < residents->length; i++) {
```

```cpp
                    // Check if the current resident is less preferred compared to the lowest preferred resident
                    if (preference->data->getResidentPreferences()[residents->arr[i].getIndex()] > preference->data->getResidentPreferences()[preference->data->getAssignedResidents()[preference->data->getLowestPreferredAssignedResidentIndex()].getIndex()]) {

                        // Take the resident out of the running if that is the case
                        residents->arr[i].getPreferencesArr()[preference->data->getIndex()] = 0;
                        preference->data->getResidentPreferences()[residents->arr[i].getIndex()] = INT_MAX;
                    }
                }
            }

            // Clean up memory because the node is no longer needed
            delete preference;
        }

        // The resident is done for now and a new node has already been created if needed
        delete resident;
    }

    std::cout << "Finished algo" << std::endl << std::endl;

    // Print the final results
    std::cout << "Final results:" << std::endl;
    for (int i = 0; i < residents->length; i++) {
        if (residents->arr[i].getAssignment() != nullptr) {
            std::cout << "(" << residents->arr[i].getName() << ", " << residents->arr[i].getAssignment()->getName() << ")" << std::endl;
        } else {
            std::cout << "(" << residents->arr[i].getName() << ", nullptr)" << std::endl;
        }
    }
}
```

Listing 1: Original Stable Matching Algorithm (C++)

3.2 Variation Algorithm

```cpp
void generateStableMatches(ResidentArr* residents, HospitalArr* hospitals) {
    std::cout << std::endl;

    for (int i = 0; i < residents->length; i++) {
        Node<Hospital*>* cur = residents->arr[i].getHospitalPreferences()->getHead();

        // Add each resident to their top choice hospital
        cur->data->addResident(&residents->arr[i], 0);
    }

    for (int i = 0; i < hospitals->length; i++) {
        hospitals->arr[i].getAssignments()[0].printList();
    }

    // Go through each level of the hospitals
    for (int i = 0; i < Hospital::NUM_LEVELS; i++) {
        for (int j = 0; j < hospitals->length; j++) {
            // Make sure the hospital is not over capacity
            while (hospitals->arr[j].getNumAssignedRange(i) > hospitals->arr[j].getCapacity()) {
                // Get the resident and remove the hospital from its list
                Node<Resident*>* res = hospitals->arr[j].getAssignments()[i].dequeue();
                Node<Hospital*>* h = res->data->getHospitalPreferences()->dequeue();
```

```cpp
                delete h;

                // Add the resident to its next preferred hospital if possible
                if (i < Hospital::NUM_LEVELS - 1) {
                    res->data->getHospitalPreferences()->getHead()->data->getAssignments()[i + 1].priorityAdd(res, i + 1);
                } else {
                    delete res;
                }
            }
        }
    }

    // Formally assign each resident to the hospitals
    for(int i = 0; i < hospitals->length; i++) {
        for (int j = 0; j < Hospital::NUM_LEVELS; j++) {
            Node<Resident*>* cur = hospitals->arr[i].getAssignments()[j].getHead();
            while (cur != nullptr) {
                cur->data->setAssignment(&hospitals->arr[i]);
                cur = cur->next;
            }
        }
    }

    std::cout << "Finished main algo" << std::endl << std::endl;

    // Print the final results
    std::cout << "Initial results:" << std::endl;
    for (int i = 0; i < residents->length; i++) {
        if (residents->arr[i].getAssignment() != nullptr) {
            std::cout << "(" << residents->arr[i].getName() << ", " << residents->arr[i].getAssignment()->getName() << ")" << std::endl;
        } else {
            std::cout << "(" << residents->arr[i].getName() << ", nullptr)" << std::endl;
        }
    }
    std::cout << std::endl;

    // Compute the happiness indices for both residents and hospitals
    std::cout << "Resident Happiness: " << computeResidentHappiness(residents) << std::endl;
    std::cout << "Hospital Happiness: " << computeHospitalHappiness(hospitals) << std::endl;

    std::cout << std::endl;


    // Make any needed adjustments to increase resident happiness
    // Create a variable to keep track of the number of swaps made
    int swaps = -1;
    while (swaps != 0) {
        swaps = 0;

        for (int i = 0; i < residents->length; i++) {
            for (int j = i + 1; j < residents->length; j++) {
                // Get the current assignments
                Hospital* iHosp = residents->arr[i].getAssignment();
                Hospital* jHosp = residents->arr[j].getAssignment();

                bool canSwap = true;
                // Make sure jHosp is a preference for resident i
                if (jHosp != nullptr && residents->arr[i].getPreferencesArr()[jHosp->getIndex()] == 0) {
                    // Cannot swap if resident i does not want to go to jHosp
                    canSwap = false;
                } else if (iHosp != nullptr && residents->arr[j].getPreferencesArr()[iHosp->getIndex()] == 0) {
```

```
                // Same but for resident j and iHosp
                canSwap = false;
            }

            if (canSwap) {
                // Compute the current average happiness among the 2 residents
                int iCurHappiness = 0;
                if (iHosp != nullptr) {
                    iCurHappiness = residents->arr[i].getPreferencesArr()[iHosp->getIndex()];
                }

                int jCurHappiness = 0;
                if (jHosp != nullptr) {
                    jCurHappiness = residents->arr[j].getPreferencesArr()[jHosp->getIndex()];
                }
                double curHappiness = (double) (iCurHappiness + jCurHappiness) / 2;

                // Compute the average happiness if the 2 residents swapped
                int iSwapHappiness = 0;
                if (jHosp != nullptr) {
                    iSwapHappiness = residents->arr[i].getPreferencesArr()[jHosp->getIndex()];
                }

                int jSwapHappiness = 0;
                if (iHosp != nullptr) {
                    jSwapHappiness = residents->arr[j].getPreferencesArr()[iHosp->getIndex()];
                }
                double swapHappiness = (double) (iSwapHappiness + jSwapHappiness) / 2;

                // Conduct a swap if needed
                if (swapHappiness > curHappiness) {
                    // Remove the residents from the hospital lists
                    if (iHosp != nullptr) {
                        iHosp->removeResident(&residents->arr[i], Hospital::NUM_LEVELS - residents->arr[i].getPreferencesArr()[iHosp->getIndex()]);
                    }

                    if (jHosp != nullptr) {
                        jHosp->removeResident(&residents->arr[j], Hospital::NUM_LEVELS - residents->arr[j].getPreferencesArr()[jHosp->getIndex()]);
                    }

                    // Add the j resident to the i hospital
                    if (iHosp != nullptr) {
                        iHosp->addResident(&residents->arr[j], Hospital::NUM_LEVELS - residents->arr[j].getPreferencesArr()[iHosp->getIndex()]);
                    }
                    residents->arr[j].setAssignment(iHosp);

                    // Add the i resident to the j hospital
                    if (jHosp != nullptr) {
                        jHosp->addResident(&residents->arr[i], Hospital::NUM_LEVELS - residents->arr[i].getPreferencesArr()[jHosp->getIndex()]);
                    }
                    residents->arr[i].setAssignment(jHosp);

                    // Increment swaps
                    swaps++;

                    std::cout << "Swapped " << residents->arr[i].getName() << " and " << residents->arr[j].getName() << std::endl;
```

```
                        }
                    }
                }
            }
            std::cout << "Swaps made: " << swaps << std::endl;
        }

    // Print the final results
    std::cout << std::endl;
    std::cout << "Final results:" << std::endl;
    for (int i = 0; i < residents->length; i++) {
        if (residents->arr[i].getAssignment() != nullptr) {
            std::cout << "(" << residents->arr[i].getName() << ", " << residents->arr[i].
                getAssignment()->getName() << ")" << std::endl;
        } else {
            std::cout << "(" << residents->arr[i].getName() << ", nullptr)" << std::endl;
        }
    }
    std::cout << std::endl;

    // Compute the happiness indices for both residents and hospitals
    std::cout << "Resident Happiness: " << computeResidentHappiness(residents) << std::endl;
    std::cout << "Hospital Happiness: " << computeHospitalHappiness(hospitals) << std::endl;
}
```

Listing 2: Variation Stable Matching Algorithm (C++)