Final Project

Josh Seligman

joshua.seligman1@marist.edu

November 5, 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
        $r.assignment \leftarrow null$    // Residents start off unassigned
    end for
    for $h$ of $hospitals$ do
        $h.assignments \leftarrow [\,]$    // Hospitals initially have no assignments
    end for
    while $!residents.isEmpty()$ do
        $r \leftarrow residents.dequeue()$    // Get the next resident in line to be assigned
        while $r.assignment == null$ && $!r.preferences.isEmpty()$ do
            $h \leftarrow r.preferences.dequeue()$    // Try the resident's next top preference
            if $h.isFull()$ then
                $r' \leftarrow h.getLeastPreferredAssignedResident()$
                $r'.assignment \leftarrow null$    // Set the least preferred assigned resident to be free
                $residents.enqueue(r)$    // Add the resident back to the list to be reassigned
            end if
            $r.assignment \leftarrow h$    // Provisionally assign r to h
            if $h.isFull()$ then
                $s \leftarrow h.getLeastPreferredAssignedResident()$
                for $i \leftarrow h.preferences.indexOf(s) + 1,\ len(h.preferences) - 1$ do
                    $s' \leftarrow h.preferences[i]$
                    $s'.preferences.remove(h)$    // 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. The asymptotic analysis of the runtime of this algorithm is explored in detail in Section 2.2.

Algorithm 2 Hospitals and Residents Stable Matching Algorithm

```
1: procedure StableMatchVariation($residents$, $hospitals$)
2:     for $r$ of $residents$ do
3:         $topHosp \leftarrow r.preferences.head$
4:         $r.assignment \leftarrow topHosp$    // Residents start with their top choice regardless of capacity
5:     end for
6:     $hasToCheck \leftarrow true$
7:     while $hasToCheck$ do
8:         $residentsToReassign \leftarrow newQueue$
9:         for $h$ of $hospitals$ do
10:            $h.sortResidents()$    // Sort the residents from worst to best for being able to leave
11:            while $h.isOverCapacity()$ do
12:                $residentToRemove \leftarrow h.assignments[h.numAssigned - 1]$    // Get best one to reassign
13:                $h.numAssigned \leftarrow h.numAssigned - 1$
14:                $residentToRemove.curPreferenceIndex \leftarrow residentToRemove.curPreferenceIndex + 1$
15:                $residentsToReassign.enqueue(residentToRemove)$    // Add it to the queue
16:            end while
17:        end for
18:        if $!residentsToReassign.isEmpty()$ then
19:            while $!residentsToReassign.isEmpty()$ do
20:                $res \leftarrow residentsToReassign.dequeue()$
21:                if $res.curPreferenceIndex < NUM\_LEVELS$ then    // Add to next preference
22:                    $res.getHospitalPreferences[res.curPreferenceIndex].addResident(res)$
23:                end if
24:            end while
25:        else
26:            $hasToCheck \leftarrow false$    // We have a stable situation
27:        end if
28:    end while
29:    $computeSwaps(residents)$    // Perform swaps to get best possible outcome
30: end procedure
```

2.2 Asymptotic Analysis

Listing 2 provides the C++ implementation of the variation algorithm to the stable matching problem in addition to the swapping algorithm in Listing 3 and how the residents compare themselves to each other in Listing 4. Before going through the algorithm, in addition to r being the number of residents and h being the number of hospitals, p will be used as the notation for the number of preferences ($p \leq h$) as all residents get a fixed number of hospital preferences. First, lines 5-8 of the variation algorithm contains a for-loop that provisionally assigns all residents to their top choice hospital, which is an $O(r)$ operation as the actual adding is a constant time operation that runs r times. Next, line 14 begins a while-loop that is dependent on the *hasToCheck* variable. The number of times the loop will run is unknown for now and will be temporarily noted with an x. Inside the loop, after created a linked list on line 17, there is a loop that iterates through all of the hospitals. First, the provisionally assigned residents get sorted by the availability of their lower preference hospitals using a quicksort algorithm. The sort itself runs in $O(rlog_2r)$ time, which means the residents are compared about $rlog_2r$ times.

The comparison algorithm in Listing 4 presents how each resident is compared with one another. After doing 2 $O(p)$ operations by iterating through the hospitals in the preference lists for both residents on lines 7-21 and some assignments on lines 24 and 25, both the if-statement branch and else-if branch contain the same bodies, but for different residents. The main section is the nested for-loop on lines 37-53. The outer loop iterates through the resident's preferences and the middle loop goes through each resident provisionally assigned to the hospital from the outer loop. Overall, this will run at worst case $r - 1$ times as all other residents will be compared to each other. The inner-most loop considers the remaining preferences for the middle man being looked at, which can be $p - 1$ iterations at worst case. The body of the inner-most loop contains assignments and calculations and runs in $O(1)$ time. Thus, the nested loops as a whole run in $O(r*p)$ time because it would have to go through all preferences for all residents. Overall, the comparison algorithm runs in $O(2p+rp)$ time, which is $O(ph)$ as rp is the dominant term. Therefore the sorting algorithm runs in $O(rp * r * log_2r)$ is $O(pr^2log_2r)$.

Once the residents are sorted on line 22 of Listing 2, the while-loop on lines 25-36 iterates until the hospital is no longer over capacity, which will be around r times. The body of the while-loop has several assignments, which run in constant time. Thus, the overall loop on lines 20-37 runs in $O(h*(pr^2log_2r+r))$ is $O(hpr^2log_2r + hr)$, which simplifies to $O(hpr^2log_2r)$. Next, the body of the if-block defined on lines 40-52 contains a while-loop that goes through all of the residents that were removed from their hospital and reassigns them to their next preference, which is an $O(r)$ operation. However, the residents are only able to be reassigned if there is still a hospital in their preference list, which means that this part of the code will only be called about p times, which is the number of times the big while-loop runs and the value of x. Therefore, the while-loop on lines 15-56 runs in $O(p * (hpr^2log_2r + r))$, which is $O(hp^2r^2log_2r)$ as pr is not dominant and can be removed for simplification purposes. Next, the loops on lines 59-63 formally assign each resident to a hospital, which is another $O(r)$ operation.

The final part of the algorithm is on line 85, which calls the swapping function. This function can be found in Listing 3. Line 4 of the swapping function defines a loop that runs until no swaps are made in an iteration. THe body of the while-loop contains nested for-loops that each go through all residents, which means the inner body runs about r^2 times. The entire body, however, is only comprised of assignments, math operations, and comparisons, which are all constant time operations. Lastly, the if-else block on lines 74-92 also hos assignments in the body, which runs in constant time. Overall, the swapping algorithm runs in $O(r^2)$ time.

The runtime for the variation algorithm is $O(hp^2r^2log_2r + r + r^2)$, which is $O(hp^2r^2log_2r)$.

2.3 Example and Walkthrough

Listing 5 contains a sample data set for the variation algorithm that contains 8 residents, 7 hospitals, and 6 preferences per resident. The first step of the algorithm is to assign each resident to their top choice hospital. The initial assignments will be as follows:

Hospital	Capacity	Residents
h1	1	r1
h2	1	r2, r6
h3	1	r3
h4	2	r4
h5	1	r5
h6	1	r7
h7	1	r8

As shown in the table of initial results, h2 is over capacity as it has 2 residents, r2 and r6, provisionally assigned to the hospital even though h2 has a capacity of only 1. Therefore, when comparing r2 and r6 for the sort, r6 is initially favored to leave h2 because it has a preference of h4, which has availability, and r2 does not. However, through the loop structure on lines 37-53 in Listing 4, it is determined that r3 can act as a middle man and would be able to fill the open space in h4 with r2 taking the r3's spot in h3. This would create a better overall resident happiness as 2 residents being in their second choice is better than 1 resident being in their top choice and the other being in their sixth choice. Therefore, r2 will be added to the list of residents to be reassigned and r6 stays put in h2. None of the other hospitals are over capacity, so there are no changes other. r2 will then be assigned to h3, which is its second choice hospital, which will cause the assignments to be as shown in the table below.

Hospital	Capacity	Residents
h1	1	r1
h2	1	r6
h3	1	r3, r2
h4	2	r4
h5	1	r5
h6	1	r7
h7	1	r8

In this next iteration, h3 is now over capacity. When comparing r3 and r2, r3 has a preference for h4, which has an availability, and none of r2's preferences are open, so r3 is prioritized to move out of h3 and will be reassigned to h4. The assignment table will look as follows.

Hospital	Capacity	Residents
h1	1	r1
h2	1	r6
h3	1	r2
h4	2	r4, r3
h5	1	r5
h6	1	r7
h7	1	r8

In the current state, all hospitals are at capacity and all residents have matched the definition of stability by being assigned only to hospitals on their preference list. Additionally, there is no swap between 2 residents that will make the overall resident happiness increase and, therefore, the algorithm terminates in this state.

3 Appendix

3.1 Original Algorithm

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

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

```cpp

    // 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++) {
                    // 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;
        }
    }
}
```

3.2 Variation Algorithm

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

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

    // Iterate through all residents
    for (int i = 0; i < residents->length; i++) {
        Hospital* cur = residents->arr[i].getHospitalPreferences()[residents->arr[i].
            getCurPreferenceIndex()];

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

    // We have to make sure that the loop iterates at least once, so initialize the variable
        to true
    bool hasToCheck = true;

    while (hasToCheck) {
        // Create a linked list of the residents that get booted from their current choice
            hospital
        List<Resident*> residentsToReassign;

        // Iterate through all of the hospitals
        for (int i = 0; i < hospitals->length; i++) {
            // Sort the residents by keeping the best one to leave at the end of the list
            hospitals->arr[i].sortResidentAssignments();
```

```cpp

            // Continue until the hospital is no longer over capacity
            while (hospitals->arr[i].getNumAssigned() - hospitals->arr[i].getCapacity() > 0) {
                hospitals->arr[i].printAssignments();

                // The last one in the assignment array will get booted
                Resident* residentToRemove = hospitals->arr[i].getAssignments()[hospitals->arr[i].getNumAssigned() - 1];
                hospitals->arr[i].setNumAssigned(hospitals->arr[i].getNumAssigned() - 1);

                // Update the resident preference to try the next hospital on its list and add it to the linked list
                residentToRemove->setCurPreferenceIndex(residentToRemove->getCurPreferenceIndex() + 1);
                Node<Resident*>* resNode = new Node<Resident*>(residentToRemove);
                residentsToReassign.enqueue(resNode);
            }
        }

        // Check to make sure there are residents that have to be reassigned
        if (residentsToReassign.getSize() > 0) {
            while (!residentsToReassign.isEmpty()) {
                // Remove the residents one by one
                Node<Resident*>* res = residentsToReassign.dequeue();

                // Add the resident to its next choice if possible
                if (res->data->getCurPreferenceIndex() < Hospital::NUM_LEVELS) {
                    res->data->getHospitalPreferences()[res->data->getCurPreferenceIndex()]->addResident(res->data);
                }

                delete res;
            }
        } else {
            // If the linked list was empty, then we know all residents are in a stable position
            hasToCheck = false;
        }
    }

    // Formally assign each resident to the hospitals
    for (int i = 0; i < hospitals->length; i++) {
        for (int j = 0; j < hospitals->arr[i].getNumAssigned(); j++) {
            hospitals->arr[i].getAssignments()[j]->setAssignment(&hospitals->arr[i]);
        }
    }

    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;

    // Try to do swaps to boost the performance of the algorithm
    performSwaps(residents);

    // 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 3: Swapping Function (C++)

```
void performSwaps(ResidentArr* residents) {
    // Make any needed adjustments to increase resident happiness
    bool swapped = true;
    while (swapped) {
        // Initialize the residents to swap to be nothing
        Resident* res1 = nullptr;
        Resident* res2 = nullptr;
        double bestDiff = 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()];
```

```cpp
                    }
                    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;

                    // Make sure the new happiness is better than it was before
                    if (swapHappiness > curHappiness) {
                        // Get the gained happiness
                        double diff = swapHappiness - curHappiness;

                        // Check to see if it is better than the current best
                        if (diff > bestDiff) {
                            // Update the variables
                            bestDiff = diff;
                            res1 = &residents->arr[i];
                            res2 = &residents->arr[j];
                        } else if (diff == bestDiff && rand() % 2 == 0) {
                            // If equal, randomly pick one or the other to use as the best difference
                            bestDiff = diff;
                            res1 = &residents->arr[i];
                            res2 = &residents->arr[j];
                        }
                    }
                }
            }
        }

        // Make the swap if a possible swap was found
        if (res1 != nullptr && res2 != nullptr) {
            // Get the hospitals being used
            Hospital* h1 = res1->getAssignment();
            Hospital* h2 = res2->getAssignment();

            // Replace the residents
            h1->replace(res1, res2);
            h2->replace(res2, res1);

            // Update the assignments
            res2->setAssignment(h1);
            res1->setAssignment(h2);

            std::cout << "Swapped␣" << res1->getName() << "␣and␣" << res2->getName() << std::endl;
        } else {
            // No swaps were made, so can set the flag to false and end the loop
            std::cout << "No␣swaps␣made" << std::endl;
            swapped = false;
        }
    }
}
```

Listing 4: Resident Comparison Function (C++)

```cpp
int Resident::compare(Resident* compResident) {
    // Get the current hospitals being compared
    int thisCur = curPreferenceIndex + 1;
    int otherCur = compResident->getCurPreferenceIndex() + 1;

    // Get the first hospital in the preference list that is still available
    while (thisCur < Hospital::NUM_LEVELS) {
        if (hospitalPreferences[thisCur]->isFull()) {
            thisCur++;
        } else {
            break;
        }
    }

    while (otherCur < Hospital::NUM_LEVELS) {
        if (compResident->getHospitalPreferences()[otherCur]->isFull()) {
            otherCur++;
        } else {
            break;
        }
    }

    // Compute the difference if each resident went to their next available choice
    int thisDiff = thisCur - curPreferenceIndex;
    int otherDiff = otherCur - compResident->getCurPreferenceIndex();

    // Assume they are the same
    int out = 0;

    if (thisDiff < otherDiff) {
        // Lower difference should be put at the end of the list
        out = 1;

        Hospital* target = hospitalPreferences[thisCur];
        bool found = false;
        // Iterate back through all possible moves to go down for the other resident
        for (int i = compResident->getCurPreferenceIndex() + 1; i < Hospital::NUM_LEVELS &&
            !found; i++) {
            // Iterate through all the residents assigned to the hospital
            for (int j = 0; j < compResident->getHospitalPreferences()[i]->getNumAssigned()
                && !found; j++) {
                Resident* middleMan = compResident->getHospitalPreferences()[i]->
                    getAssignments()[j];
                for (int k = middleMan->getCurPreferenceIndex() + 1; k < Hospital::
                    NUM_LEVELS && !found; k++) {
                    if (middleMan->getHospitalPreferences()[k] == target) {
                        int middleDiff = k - middleMan->getCurPreferenceIndex();
                        int newOtherDiff = i - compResident->getCurPreferenceIndex();
                        double avg = (double) (middleDiff + newOtherDiff) / 2;
                        if (avg < thisDiff) {
                            out = -1;
                            found = true;
                        }
                    }
                }
            }
        }
    } else if (thisDiff > otherDiff) {
        // Higher difference should be kept near the front
        out = -1;

        Hospital* target = compResident->getHospitalPreferences()[otherCur];
        bool found = false;
        // Iterate back through all possible moves to go down for the other resident
        for (int i = curPreferenceIndex + 1; i < Hospital::NUM_LEVELS && !found; i++) {
            // Iterate through all the residents assigned to the hospital
            for (int j = 0; j < hospitalPreferences[i]->getNumAssigned() && !found; j++) {
                Resident* middleMan = hospitalPreferences[i]->getAssignments()[j];
                for (int k = middleMan->getCurPreferenceIndex() + 1; k < Hospital::
                    NUM_LEVELS && !found; k++) {
                    if (middleMan->getHospitalPreferences()[k] == target) {
                        int middleDiff = k - middleMan->getCurPreferenceIndex();
                        int newThisDiff = i - curPreferenceIndex;
                        double avg = (double) (middleDiff + newThisDiff) / 2;
                        if (avg < thisDiff) {
                            out = -1;
                            found = true;
                        }
                    }
                }
            }
        }
    } else {
        // If the differences are the same, prioritize keeping the one that is happier as it
            is
        if (curPreferenceIndex < compResident->getCurPreferenceIndex()) {
            out = -1;
        } else if (curPreferenceIndex > compResident->getCurPreferenceIndex()) {
            out = 1;
        }
    }

    return out;
}
```

Listing 5: Variation Algorithm Sample Test Data

```
r1: h1 h3 h2 h4 h5 h7
r2: h2 h3 h6 h5 h1 h7
r3: h3 h4 h1 h2 h5 h7
r4: h4 h5 h1 h2 h3 h7
r5: h5 h4 h1 h2 h3 h7
r6: h2 h1 h3 h5 h7 h4
r7: h6 h1 h2 h3 h4 h7
r8: h7 h1 h2 h3 h4 h5
h1: 1
h2: 1
h3: 1
h4: 2
h5: 1
h6: 1
h7: 1
```