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

1: **procedure** STABLEMATCHORIGINAL(*residents*, *hospitals*)
2:     **for** *r of residents* **do**
3:         *r.assignment* ← *null*   // *Residents start off unassigned*
4:     **end for**
5:     **for** *h of hospitals* **do**
6:         *h.assignments* ← [ ]   // *Hospitals initially have no assignments*
7:     **end for**
8:     **while** !*residents.isEmpty*() **do**
9:         r ← *residents.dequeue*()   // *Get the next resident in line to be assigned*
10:         **while** *r.assignment* == *null* && !*r.preferences.isEmpty*() **do**
11:             h ← *r.preferences.dequeue*()   // *Try the resident's next top preference*
12:             **if** *h.isFull*() **then**
13:                 r' ← *h.getLeastPreferredAssignedResident*()
14:                 *r'.assignment* ← *null*   // *Set the least preferred assigned resident to be free*
15:                 *residents.enqueue*(r)   // *Add the resident back to the list to be reassigned*
16:             **end if**
17:             *r.assignment* ← h // *Provisionally assign r to h*
18:             **if** *h.isFull*() **then**
19:                 s ← *h.getLeastPreferredAssignedResident*()
20:                 **for** i ← *h.preferences.indexOf*(s) + 1, *len*(*h.preferences*) − 1 **do**
21:                     s' ← *h.preferences*[i]
22:                     *s'.preferences.remove*(h)   // *Remove h from preferences of s'*
23:                     *h.preferences.remove*(s')   // *Remove s' from preferences of h*
24:                 **end for**
25:             **end if**
26:         **end while**
27:     **end while**
28: **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 ← r.preferences.head*
4:         *r.assignment ← topHosp*   // *Residents start with their top choice regardless of capacity*
5:     **end for**
6:     *hasToCheck ← true*
7:     **while** *hasToCheck* **do**
8:         *residentsToReassign ← 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 ← h.assignments[h.numAssigned − 1]*   // *Get best one to reassign*
13:                *h.numAssigned ← h.numAssigned − 1*
14:                *residentToRemove.curPreferenceIndex ← 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 ← 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 ← 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 $O(hp^2r^2log_2r)$.

3 Appendix

3.1 Original Algorithm

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

```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
```

```cpp
            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++)

```cpp
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
```

```cpp
20          for (int i = 0; i < hospitals->length; i++) {
21              // Sort the residents by keeping the best one to leave at the end of the list
22              hospitals->arr[i].sortResidentAssignments();
23
24              // Continue until the hospital is no longer over capacity
25              while (hospitals->arr[i].getNumAssigned() - hospitals->arr[i].getCapacity() > 0)
                    {
26                  hospitals->arr[i].printAssignments();
27
28                  // The last one in the assignment array will get booted
29                  Resident* residentToRemove = hospitals->arr[i].getAssignments()[hospitals->
                        arr[i].getNumAssigned() - 1];
30                  hospitals->arr[i].setNumAssigned(hospitals->arr[i].getNumAssigned() - 1);
31
32                  // Update the resident preference to try the next hospital on its list and
                        add it to the linked list
33                  residentToRemove->setCurPreferenceIndex(residentToRemove->
                        getCurPreferenceIndex() + 1);
34                  Node<Resident*>* resNode = new Node<Resident*>(residentToRemove);
35                  residentsToReassign.enqueue(resNode);
36              }
37          }
38
39          // Check to make sure there are residents that have to be reassigned
40          if (residentsToReassign.getSize() > 0) {
41              while (!residentsToReassign.isEmpty()) {
42                  // Remove the residents one by one
43                  Node<Resident*>* res = residentsToReassign.dequeue();
44
45                  // Add the resident to its next choice if possible
46                  if (res->data->getCurPreferenceIndex() < Hospital::NUM_LEVELS) {
47                      res->data->getHospitalPreferences()[res->data->getCurPreferenceIndex()
                            ]->addResident(res->data);
48                  }
49
50                  delete res;
51              }
52          } else {
53              // If the linked list was empty, then we know all residents are in a stable
                    position
54              hasToCheck = false;
55          }
56      }
57
58      // Formally assign each resident to the hospitals
59      for (int i = 0; i < hospitals->length; i++) {
60          for (int j = 0; j < hospitals->arr[i].getNumAssigned(); j++) {
61              hospitals->arr[i].getAssignments()[j]->setAssignment(&hospitals->arr[i]);
62          }
63      }
64
65      std::cout << "Finished_main_algo" << std::endl << std::endl;
66
67      // Print the final results
68      std::cout << "Initial_results:" << std::endl;
69      for (int i = 0; i < residents->length; i++) {
70          if (residents->arr[i].getAssignment() != nullptr) {
71              std::cout << "(" << residents->arr[i].getName() << ",_" << residents->arr[i].
                    getAssignment()->getName() << ")" << std::endl;
72          } else {
73              std::cout << "(" << residents->arr[i].getName() << ",_nullptr)" << std::endl;
74          }
75      }
76      std::cout << std::endl;
77
```

```cpp
78        // Compute the happiness indices for both residents and hospitals
79        std::cout << "Resident_Happiness:_" << computeResidentHappiness(residents) << std::endl;
80        std::cout << "Hospital_Happiness:_" << computeHospitalHappiness(hospitals) << std::endl;
81
82        std::cout << std::endl;
83
84        // Try to do swaps to boost the performance of the algorithm
85        performSwaps(residents);
86
87        // Print the final results
88        std::cout << std::endl;
89        std::cout << "Final_results:" << std::endl;
90        for (int i = 0; i < residents->length; i++) {
91            if (residents->arr[i].getAssignment() != nullptr) {
92                std::cout << "(" << residents->arr[i].getName() << ",_" << residents->arr[i].
                      getAssignment()->getName() << ")" << std::endl;
93            } else {
94                std::cout << "(" << residents->arr[i].getName() << ",_nullptr)" << std::endl;
95            }
96        }
97        std::cout << std::endl;
98
99        // Compute the happiness indices for both residents and hospitals
100       std::cout << "Resident_Happiness:_" << computeResidentHappiness(residents) << std::endl;
101       std::cout << "Hospital_Happiness:_" << computeHospitalHappiness(hospitals) << std::endl;
102 }
```

Listing 3: Swapping Function (C++)

```cpp
1 void performSwaps(ResidentArr* residents) {
2     // Make any needed adjustments to increase resident happiness
3     bool swapped = true;
4     while (swapped) {
5         // Initialize the residents to swap to be nothing
6         Resident* res1 = nullptr;
7         Resident* res2 = nullptr;
8         double bestDiff = 0;
9
10        for (int i = 0; i < residents->length; i++) {
11            for (int j = i + 1; j < residents->length; j++) {
12                // Get the current assignments
13                Hospital* iHosp = residents->arr[i].getAssignment();
14                Hospital* jHosp = residents->arr[j].getAssignment();
15
16                bool canSwap = true;
17                // Make sure jHosp is a preference for resident i
18                if (jHosp != nullptr && residents->arr[i].getPreferencesArr()[jHosp->
                      getIndex()] == 0) {
19                    // Cannot swap if resident i does not want to go to jHosp
20                    canSwap = false;
21                } else if (iHosp != nullptr && residents->arr[j].getPreferencesArr()[iHosp->
                      getIndex()] == 0) {
22                    // Same but for resident j and iHosp
23                    canSwap = false;
24                }
25
26                if (canSwap) {
27                    // Compute the current average happiness among the 2 residents
28                    int iCurHappiness = 0;
29                    if (iHosp != nullptr) {
30                        iCurHappiness = residents->arr[i].getPreferencesArr()[iHosp->
                              getIndex()];
31                    }
32
33                    int jCurHappiness = 0;
```

```cpp
34                    if (jHosp != nullptr) {
35                        jCurHappiness = residents->arr[j].getPreferencesArr()[jHosp->
                              getIndex()];
36                    }
37                    double curHappiness = (double) (iCurHappiness + jCurHappiness) / 2;
38
39                    // Compute the average happiness if the 2 residents swapped
40                    int iSwapHappiness = 0;
41                    if (jHosp != nullptr) {
42                        iSwapHappiness = residents->arr[i].getPreferencesArr()[jHosp->
                              getIndex()];
43                    }
44
45                    int jSwapHappiness = 0;
46                    if (iHosp != nullptr) {
47                        jSwapHappiness = residents->arr[j].getPreferencesArr()[iHosp->
                              getIndex()];
48                    }
49                    double swapHappiness = (double) (iSwapHappiness + jSwapHappiness) / 2;
50
51                    // Make sure the new happiness is better than it was before
52                    if (swapHappiness > curHappiness) {
53                        // Get the gained happiness
54                        double diff = swapHappiness - curHappiness;
55
56                        // Check to see if it is better than the current best
57                        if (diff > bestDiff) {
58                            // Update the variables
59                            bestDiff = diff;
60                            res1 = &residents->arr[i];
61                            res2 = &residents->arr[j];
62                        } else if (diff == bestDiff && rand() % 2 == 0) {
63                            // If equal, randomly pick one or the other to use as the best
                                  difference
64                            bestDiff = diff;
65                            res1 = &residents->arr[i];
66                            res2 = &residents->arr[j];
67                        }
68                    }
69                }
70            }
71        }
72
73        // Make the swap if a possible swap was found
74        if (res1 != nullptr && res2 != nullptr) {
75            // Get the hospitals being used
76            Hospital* h1 = res1->getAssignment();
77            Hospital* h2 = res2->getAssignment();
78
79            // Replace the residents
80            h1->replace(res1, res2);
81            h2->replace(res2, res1);
82
83            // Update the assignments
84            res2->setAssignment(h1);
85            res1->setAssignment(h2);
86
87            std::cout << "Swapped_" << res1->getName() << "_and_" << res2->getName() << std
                  ::endl;
88        } else {
89            // No swaps were made, so can set the flag to false and end the loop
90            std::cout << "No_swaps_made" << std::endl;
91            swapped = false;
92        }
93    }
```

```cpp
94 }
```

Listing 4: Resident Comparison Function (C++)

```cpp
1  int Resident::compare(Resident* compResident) {
2      // Get the current hospitals being compared
3      int thisCur = curPreferenceIndex + 1;
4      int otherCur = compResident->getCurPreferenceIndex() + 1;
5
6      // Get the first hospital in the preference list that is still available
7      while (thisCur < Hospital::NUM_LEVELS) {
8          if (hospitalPreferences[thisCur]->isFull()) {
9              thisCur++;
10         } else {
11             break;
12         }
13     }
14
15     while (otherCur < Hospital::NUM_LEVELS) {
16         if (compResident->getHospitalPreferences()[otherCur]->isFull()) {
17             otherCur++;
18         } else {
19             break;
20         }
21     }
22
23     // Compute the difference if each resident went to their next available choice
24     int thisDiff = thisCur - curPreferenceIndex;
25     int otherDiff = otherCur - compResident->getCurPreferenceIndex();
26
27     // Assume they are the same
28     int out = 0;
29
30     if (thisDiff < otherDiff) {
31         // Lower difference should be put at the end of the list
32         out = 1;
33
34         Hospital* target = hospitalPreferences[thisCur];
35         bool found = false;
36         // Iterate back through all possible moves to go down for the other resident
37         for (int i = compResident->getCurPreferenceIndex() + 1; i < Hospital::NUM_LEVELS &&
                !found; i++) {
38             // Iterate through all the residents assigned to the hospital
39             for (int j = 0; j < compResident->getHospitalPreferences()[i]->getNumAssigned()
                    && !found; j++) {
40                 Resident* middleMan = compResident->getHospitalPreferences()[i]->
                        getAssignments()[j];
41                 for (int k = middleMan->getCurPreferenceIndex() + 1; k < Hospital::
                        NUM_LEVELS && !found; k++) {
42                     if (middleMan->getHospitalPreferences()[k] == target) {
43                         int middleDiff = k - middleMan->getCurPreferenceIndex();
44                         int newOtherDiff = i - compResident->getCurPreferenceIndex();
45                         double avg = (double) (middleDiff + newOtherDiff) / 2;
46                         if (avg < thisDiff) {
47                             out = -1;
48                             found = true;
49                         }
50                     }
51                 }
52             }
53         }
54     } else if (thisDiff > otherDiff) {
55         // Higher difference should be kept near the front
56         out = -1;
57
```

```cpp
58            Hospital* target = compResident->getHospitalPreferences()[otherCur];
59            bool found = false;
60            // Iterate back through all possible moves to go down for the other resident
61            for (int i = curPreferenceIndex + 1; i < Hospital::NUM_LEVELS && !found; i++) {
62                // Iterate through all the residents assigned to the hospital
63                for (int j = 0; j < hospitalPreferences[i]->getNumAssigned() && !found; j++) {
64                    Resident* middleMan = hospitalPreferences[i]->getAssignments()[j];
65                    for (int k = middleMan->getCurPreferenceIndex() + 1; k < Hospital::
                            NUM_LEVELS && !found; k++) {
66                        if (middleMan->getHospitalPreferences()[k] == target) {
67                            int middleDiff = k - middleMan->getCurPreferenceIndex();
68                            int newThisDiff = i - curPreferenceIndex;
69                            double avg = (double) (middleDiff + newThisDiff) / 2;
70                            if (avg < thisDiff) {
71                                out = -1;
72                                found = true;
73                            }
74                        }
75                    }
76                }
77            }
78        } else {
79            // If the differences are the same, prioritize keeping the one that is happier as it
                    is
80            if (curPreferenceIndex < compResident->getCurPreferenceIndex()) {
81                out = -1;
82            } else if (curPreferenceIndex > compResident->getCurPreferenceIndex()) {
83                out = 1;
84            }
85        }
86
87        return out;
88 }
```