Final Project

Josh Seligman

joshua.seligman1@marist.edu

November 6, 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 operations. The assignment on line 17 sets the hospital 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. However, as shown in Listing 1 on lines 57-65, a space for time tradeoff was made as the index of each resident and hospital was stored to prevent having to iterate through the lists

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**

---

to get the appropriate position. This changes the runtime of the entire loop to be $O(r)$ with the condition and body running in constant time. Overall, when putting it all together, the runtime of the original stable matching problem for residents and hospitals is $r + h + r * h * (r + r + r) = r + h + rh * (3r) = r + h + 3r^2h$. This can be simplified to $O(r^2)$ because there are typically a lot more residents than hospitals, so r^2 becomes the dominant term in the expression.

1.3 EXAMPLE AND WALKTHROUGH

Listing 2 contains a sample test case for the algorithm using 4 residents and 3 hospitals. The initial state of the algorithm is shown below:

Entity Name	Preferences
r1	h1 h2 h3
r2	h1 h3 h2
r3	h1 h2
r4	h2 h3
h1 (1)	r2 r3
h2 (1)	r4 r3
h3 (1)	r1 r2 r4

The residents go in order and try to propose to their preferred hospitals. The first resident to propose is r1, which begins by proposing to h1. Although r1 is not on h1's preference list, it will be provisionally assigned there for now because h1 has no other residents to take in. The table will be updated as shown below.

Entity Name	Preferences
r1	**h1** h2 h3
r2	h1 h3 h2
r3	h1 h2
r4	h2 h3
h1 (1)	r2 r3
h2 (1)	r4 r3
h3 (1)	r1 r2 r4

Next, r2 will propose to its top choice of h1. Since h1 is full and r2 is still on the hospital's preference list, h1 will find its lowest preferred resident that is assigned to it (r1) and unassign it to make room to r2. Once r2 gets assigned, all residents that are below r2 on h1's preference list are removed and also remove h1 from their list.

Entity Name	Preferences
r1	~~h1~~ h2 h3
r2	**h1** h3 h2
r3	~~h1~~ h2
r4	h2 h3
h1 (1)	r2 ~~r3~~
h2 (1)	r4 r3
h3 (1)	r1 r2 r4

It is now r3's turn to propose and, since h1 was removed from its list in the last step, it will turn to h2 for an assignment. Since r3 is on h2's preference list and the hospital is not at capacity, r3 will be assigned to h2. The hospital is now full, so all residents below r3 on h2's list get removed and also remove h2 from their respective lists. The updated table is shown below.

Entity Name	Preferences
r1	~~h1~~ ~~h2~~ h3
r2	**h1** h3 ~~h2~~
r3	~~h1~~ **h2**
r4	h2 h3
h1 (1)	r2 ~~r3~~
h2 (1)	r4 r3
h3 (1)	r1 r2 r4

r4 will now propose to its top choice of h2, which still has r4 on its list. Since h2 is already full, r3 will have to be removed prior to handling r4's assignment. Once r4 gets assigned to h2, all residents below it on h2's list will be removed and will also remove h2 from their lists.

Entity Name	Preferences
r1	~~h1~~ ~~h2~~ h3
r2	**h1** h3 ~~h2~~
r3	~~h1~~ ~~h2~~
r4	**h2** h3
h1 (1)	r2 ~~r3~~
h2 (1)	r4 ~~r3~~
h3 (1)	r1 r2 r4

The resident queue goes back to r1, which only has h3 left in its preference list. Thus, r1 will propose to h3 and all resident and hospital lists will be adjusted accordingly as h3 as it will now be full with r1 being assigned to it.

Entity Name	Preferences
r1	~~h1~~ ~~h2~~ **h3**
r2	**h1** ~~h3~~ ~~h2~~
r3	~~h1~~ ~~h2~~
r4	**h2** ~~h3~~
h1 (1)	r2 ~~r3~~
h2 (1)	r4 ~~r3~~
h3 (1)	r1 ~~r2~~ ~~r4~~

Lastly, r3 is reached again, but none of its preferences are available, so it will remain unassigned. The algorithm will then terminate in the state presented above.

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 3 provides the C++ implementation of the variation algorithm to the stable matching problem in addition to the swapping algorithm in Listing 5 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 $O(rp)$ 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 5. 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 has 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 6 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 other changes. 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++)

```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++) {
```

```cpp
19                    std::cout << resident->data->getPreferencesArr()[j] << "␣";
20                }
21                std::cout << std::endl;
22
23                // Get the hospital at the front of the preference list
24                Node<Hospital*>* preference = resident->data->getHospitalPreferences()->dequeue
                    ();
25
26                // The hospital is full with more preferable residents, so the current resident
                    is not getting in
27                if (resident->data->getPreferencesArr()[preference->data->getIndex()] == 0) {
28                    // Clean up memory and try the next hospital
29                    delete preference;
30                    continue;
31                }
32
33                if (preference->data->isFull()) {
34                    // Get the lowest preferred assigned resident index
35                    preference->data->setLowestPreferredAssignedResidentIndex();
36
37                    // If the hospital is full, then replace the lowest preferred resident with
                        the current resident
38                    Resident* lowest = &residents->arr[preference->data->getAssignedResidents()[
                        preference->data->getLowestPreferredAssignedResidentIndex()].getIndex()
                        ];
39                    lowest->setAssignment(nullptr);
40
41                    preference->data->replaceLowest(resident->data);
42                    // Since the lowest preferred resident was replaced, we need to come back to
                        it later
43                    residentQueue.enqueue(new Node<Resident*>(lowest));
44                } else {
45                    // Add the new resident to the
46                    preference->data->addResident(resident->data);
47                }
48
49                // Print the assignment as it happens
50                std::cout << "Assigned␣" << resident->data->getName() << "␣to␣" << resident->
                    data->getAssignment()->getName() << std::endl;
51
52                if (preference->data->isFull()) {
53                    // Get the lowest preferred assigned resident index
54                    preference->data->setLowestPreferredAssignedResidentIndex();
55
56                    // We need to iterate through all of the residents
57                    for (int i = 0; i < residents->length; i++) {
58                        // Check if the current resident is less preferred compared to the
                            lowest preferred resident
59                        if (preference->data->getResidentPreferences()[residents->arr[i].
                            getIndex()] > preference->data->getResidentPreferences()[preference
                            ->data->getAssignedResidents()[preference->data->
                            getLowestPreferredAssignedResidentIndex()].getIndex()]) {
60
61                            // Take the resident out of the running if that is the case
62                            residents->arr[i].getPreferencesArr()[preference->data->getIndex()]
                                = 0;
63                            preference->data->getResidentPreferences()[residents->arr[i].
                                getIndex()] = INT_MAX;
64                        }
65                    }
66                }
67
68                // Clean up memory because the node is no longer needed
69                delete preference;
70            }
```

```
71
72          // The resident is done for now and a new node has already been created if needed
73          delete resident;
74      }
75
76      std::cout << "Finished algo" << std::endl << std::endl;
77
78      // Print the final results
79      std::cout << "Final results:" << std::endl;
80      for (int i = 0; i < residents->length; i++) {
81          if (residents->arr[i].getAssignment() != nullptr) {
82              std::cout << "(" << residents->arr[i].getName() << ", " << residents->arr[i].
                      getAssignment()->getName() << ")" << std::endl;
83          } else {
84              std::cout << "(" << residents->arr[i].getName() << ", nullptr)" << std::endl;
85          }
86      }
87 }
```

Listing 2: Original Algorithm Sample Test Data

```
1 r1: h1 h2 h3
2 r2: h1 h3 h2
3 r3: h1 h2
4 r4: h2 h3
5 h1: 1 - r2 r3
6 h2: 1 - r4 r3
7 h3: 1 - r1 r2 r4
```

3.2 VARIATION ALGORITHM

Listing 3: Variation Stable Matching Algorithm (C++)

```
1 void generateStableMatches(ResidentArr* residents, HospitalArr* hospitals) {
2      std::cout << std::endl;
3
4      // Iterate through all residents
5      for (int i = 0; i < residents->length; i++) {
6          Hospital* cur = residents->arr[i].getHospitalPreferences()[residents->arr[i].
                  getCurPreferenceIndex()];
7
8          // Add each resident to their top choice hospital
9          cur->addResident(&residents->arr[i]);
10     }
11
12     // We have to make sure that the loop iterates at least once, so initialize the variable
              to true
13     bool hasToCheck = true;
14
15     while (hasToCheck) {
16         // Create a linked list of the residents that get booted from their current choice
                  hospital
17         List<Resident*> residentsToReassign;
18
19         // Iterate through all of the hospitals
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
```

```cpp
                    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() < NUM_PREFERENCES) {
                        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
```

```cpp
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 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 < NUM_PREFERENCES) {
8          if (hospitalPreferences[thisCur]->isFull()) {
9              thisCur++;
10         } else {
11             break;
12         }
13     }
14
15     while (otherCur < NUM_PREFERENCES) {
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 < NUM_PREFERENCES && !
                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 < NUM_PREFERENCES &&
                        !found; k++) {
42                     if (middleMan->getHospitalPreferences()[k] == target) {
```

```cpp
                            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 < NUM_PREFERENCES && !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 < NUM_PREFERENCES &&
                             !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: Swapping Function (C++)

```cpp
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();
```

```cpp
15
                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;

                    // 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];
                        }
                    }
                }
            }
        }
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      }
94  }
```

Listing 6: Variation Algorithm Sample Test Data

```
1  r1: h1 h3 h2 h4 h5 h7
2  r2: h2 h3 h6 h5 h1 h7
3  r3: h3 h4 h1 h2 h5 h7
4  r4: h4 h5 h1 h2 h3 h7
5  r5: h5 h4 h1 h2 h3 h7
6  r6: h2 h1 h3 h5 h7 h4
7  r7: h6 h1 h2 h3 h4 h7
8  r8: h7 h1 h2 h3 h4 h5
9  h1: 1
10 h2: 1
11 h3: 1
12 h4: 2
13 h5: 1
14 h6: 1
15 h7: 1
```