Final Project

Josh Seligman

joshua.seligman1@marist.edu

October 26, 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.

2.2 Asymptotic Analysis

3 Appendix

3.1 Original Algorithm

```
1  void generateStableMatches(ResidentArr* residents, HospitalArr* hospitals) {
2      Queue<Resident*> residentQueue;
3
4      // Add the residents to a queue for determining the next resident to propose
5      for (int i = 0; i < residents->length; i++) {
6          Node<Resident*>* n = new Node(&residents->arr[i]);
7          residentQueue.enqueue(n);
8      }
9
10     std::cout << std::endl;
11
12     while (!residentQueue.isEmpty()) {
13         residentQueue.printQueue();
14
15         Node<Resident*>* resident = residentQueue.dequeue();
16
17         while (resident->data->getAssignment() == nullptr && !resident->data->
               getHospitalPreferences()->isEmpty()) {
18             for (int j = 0; j < hospitals->length; j++) {
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
```

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:     for i ← 0, NUM_LEVELS − 1 do    // Have to check at all levels
7:         for j ← 0, len(hospitals) − 1 do    // Have to iterate through each hospital
8:             while hospitals[j].numAssigned(0, i) > hospitals[j].capacity do    // Continue until the hospital is no longer over capacity at the given level
9:                 res ← hospitals[j].assignments[i].dequeue()    // Get the resident being removed
10:                res.preferences.dequeue()    // Remove the hospital from the resident's preferences
11:                if i < NUM_LEVELS − 1 then
12:                    newTopHospital ← res.preferences.head
13:                    newTopHospital.assignments[i + 1].addres    // Add the resident to their next top choice
14:                end if
15:            end while
16:        end for
17:    end for
18:    for i ← 0, len(hospitals) − 1 do
19:        for j ← 0, NUM_LEVELS − 1 do    // Iterate through all the residents assigned to the hospital
20:            cur ← hospitals[i].assignments[j].head
21:            while cur != null do
22:                cur.assignment ← hospitals[i]    // Formally assign the resident to the hospital
23:                cur ← cur.next
24:            end while
25:        end for
26:    end for
27:    swaps ← −1    // Initialize to -1 to enter the loop
28:    while swaps != 0 do
29:        swaps ← 0    // Start off the iteration with 0 swaps
30:        for i ← 0, len(residents) − 1 do
31:            for j ← i + 1, len(residents) − 1 do    // Compare all residents to all residents
32:                iHosp ← residents[i].assignment    // Get the current assignments
33:                jHosp ← residents[j].assignment
34:                canSwap ← true    // Assume we can swap and end up with a stable pairing
35:                if residents[i].rankings[jHosp.index] != 0 then
36:                    canSwap ← false    // Cannot swap because resident i didn't rank jHosp
37:                else if residents[j].rankings[iHosp.index] != 0 then
38:                    canSwap ← false    // Same but for resident j
39:                end if
40:                if canSwap then
41:                    curHappiness ← avg(residents[i].rankings[iHosp.index], residents[j].rankings[jHosp.index])
42:                    swapHappiness ← avg(residents[i].rankings[jHosp.index], residents[j].rankings[iHosp.index])
43:                    if swapHappiness > curHappiness then    // Swapping would result in a better net happiness
44:                        iHosp.remove(residents[i])    // Remove the residents from the hopsitals
45:                        jHosp.remove(residents[j])
46:                        iHosp.add(residents[j])    // Swap the residents and the hospitals they are assigned to
47:                        residents[j].assignment ← iHosp
48:                        jHosp.add(residents[i])
49:                        residents[i].assignment ← jHosp
50:                        swaps ← swaps + 1    // Increment the number of swaps made
51:                    end if
52:                end if
53:            end for
54:        end for
55:    end while
56: end procedure
```

```
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      }
```

```
}
```

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

3.2 Variation Algorithm

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

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

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

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

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

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

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

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

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

```
57      std::cout << std::endl;
58
59      // Compute the happiness indices for both residents and hospitals
60      std::cout << "Resident Happiness: " << computeResidentHappiness(residents) << std::endl;
61      std::cout << "Hospital Happiness: " << computeHospitalHappiness(hospitals) << std::endl;
62
63      std::cout << std::endl;
64
65
66      // Make any needed adjustments to increase resident happiness
67      // Create a variable to keep track of the number of swaps made
68      int swaps = -1;
69      while (swaps != 0) {
70          swaps = 0;
71
72          for (int i = 0; i < residents->length; i++) {
73              for (int j = i + 1; j < residents->length; j++) {
74                  // Get the current assignments
75                  Hospital* iHosp = residents->arr[i].getAssignment();
76                  Hospital* jHosp = residents->arr[j].getAssignment();
77
78                  bool canSwap = true;
79                  // Make sure jHosp is a preference for resident i
80                  if (jHosp != nullptr && residents->arr[i].getPreferencesArr()[jHosp->
                        getIndex()] == 0) {
81                      // Cannot swap if resident i does not want to go to jHosp
82                      canSwap = false;
83                  } else if (iHosp != nullptr && residents->arr[j].getPreferencesArr()[iHosp->
                        getIndex()] == 0) {
84                      // Same but for resident j and iHosp
85                      canSwap = false;
86                  }
87
88                  if (canSwap) {
89                      // Compute the current average happiness among the 2 residents
90                      int iCurHappiness = 0;
91                      if (iHosp != nullptr) {
92                          iCurHappiness = residents->arr[i].getPreferencesArr()[iHosp->
                                getIndex()];
93                      }
94
95                      int jCurHappiness = 0;
96                      if (jHosp != nullptr) {
97                          jCurHappiness = residents->arr[j].getPreferencesArr()[jHosp->
                                getIndex()];
98                      }
99                      double curHappiness = (double) (iCurHappiness + jCurHappiness) / 2;
100
101                     // Compute the average happiness if the 2 residents swapped
102                     int iSwapHappiness = 0;
103                     if (jHosp != nullptr) {
104                         iSwapHappiness = residents->arr[i].getPreferencesArr()[jHosp->
                                getIndex()];
105                     }
106
107                     int jSwapHappiness = 0;
108                     if (iHosp != nullptr) {
109                         jSwapHappiness = residents->arr[j].getPreferencesArr()[iHosp->
                                getIndex()];
110                     }
111                     double swapHappiness = (double) (iSwapHappiness + jSwapHappiness) / 2;
112
113                     // Conduct a swap if needed
114                     if (swapHappiness > curHappiness) {
115                         // Remove the residents from the hospital lists
```

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

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

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

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

                        // Increment swaps
                        swaps++;

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

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

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

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