

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 $r * h * (r + r * (h + r)) = rh * (2r + rh + r^2) = 2r^2h + r^2h^2 + r^3h = O(r^2h + r^2h^2 + r^3h)$.

Algorithm 1 Hospitals and Residents Stable Matching Algorithm

```
1: procedure STABLEMATCHORIGINAL(residents, hospitals)
2:   for r of residents do
3:     r.assignment  $\leftarrow$  null    // Residents start off unassigned
4:   end for
5:   for h of hospitals do
6:     h.assignments  $\leftarrow$  [ ]    // Hospitals initially have no assignments
7:   end for
8:   while !residents.isEmpty() do
9:     r  $\leftarrow$  residents.dequeue()    // Get the next resident in line to be assigned
10:    while r.assignment == null && !r.preferences.isEmpty() do
11:      h  $\leftarrow$  r.preferences.dequeue()    // Try the resident's next top preference
12:      if h.isFull() then
13:        r'  $\leftarrow$  h.getLeastPreferredAssignedResident()
14:        r'.assignment  $\leftarrow$  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  $\leftarrow$  h    // Provisionally assign r to h
18:      if h.isFull() then
19:        s  $\leftarrow$  h.getLeastPreferredAssignedResident()
20:        for i  $\leftarrow$  h.preferences.indexOf(s) + 1, len(h.preferences) - 1 do
21:          s'  $\leftarrow$  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->
18            getHospitalPreferences()->isEmpty()) {
19            for (int j = 0; j < hospitals->length; j++) {
20                std::cout << resident->data->getPreferencesArr()[j] << "\n";
21            }
22            std::cout << std::endl;
23
24            // Get the hospital at the front of the preference list
25            Node<Hospital*>* preference = resident->data->getHospitalPreferences()->dequeue
26                ();
27
28            // The hospital is full with more preferable residents, so the current resident
29            // is not getting in
30            if (resident->data->getPreferencesArr()[preference->data->getIndex()] == 0) {
31                // Clean up memory and try the next hospital
32                delete preference;
33                continue;
34            }
35
36            if (preference->data->isFull()) {
37                // 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
38         // the current resident
39         Resident* lowest = &residents->arr[preference->data->getAssignedResidents()[
40             preference->data->getLowestPreferredAssignedResidentIndex()].getIndex()
41         ];
42         lowest->setAssignment(nullptr);
43
44         preference->data->replaceLowest(resident->data);
45         // Since the lowest preferred resident was replaced, we need to come back to
46         // it later
47         residentQueue.enqueue(new Node<Resident*>(lowest));
48     } else {
49         // Add the new resident to the
50         preference->data->addResident(resident->data);
51     }
52
53     // Print the assignment as it happens
54     std::cout << "Assigned_" << resident->data->getName() << "_to_" << resident->
55     data->getAssignment()->getName() << std::endl;
56
57     if (preference->data->isFull()) {
58         // Get the lowest preferred assigned resident index
59         preference->data->setLowestPreferredAssignedResidentIndex();
60
61         // We need to iterate through all of the residents
62         for (int i = 0; i < residents->length; i++) {
63             // Check if the current resident is less preferred compared to the
64             // lowest preferred resident
65             if (preference->data->getResidentPreferences()[residents->arr[i].
66                 getIndex()] > preference->data->getResidentPreferences()[preference
67                 ->data->getAssignedResidents()[preference->data->
68                 getLowestPreferredAssignedResidentIndex()].getIndex()]) {
69
70                 // Take the resident out of the running if that is the case
71                 residents->arr[i].getPreferencesArr()[preference->data->getIndex()]
72                 = 0;
73                 preference->data->getResidentPreferences()[residents->arr[i].
74                 getIndex()] = INT_MAX;
75             }
76         }
77     }
78
79     // Clean up memory because the node is no longer needed
80     delete preference;
81 }
82
83 // The resident is done for now and a new node has already been created if needed
84 delete resident;
85 }
86
87 std::cout << "Finished_algo" << std::endl << std::endl;
88
89 // Print the final results
90 std::cout << "Final_results:" << std::endl;
91 for (int i = 0; i < residents->length; i++) {
92     if (residents->arr[i].getAssignment() != nullptr) {
93         std::cout << "(" << residents->arr[i].getName() << ",_" << residents->arr[i].
94         getAssignment()->getName() << ")" << std::endl;
95     } else {
96         std::cout << "(" << residents->arr[i].getName() << ",_nullptr)" << std::endl;
97     }
98 }

```

87 }

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

3.2 VARIATION ALGORITHM

```
1 void generateStableMatches(ResidentArr* residents, HospitalArr* hospitals) {
2     std::cout << std::endl;
3
4     for (int i = 0; i < residents->length; i++) {
5         Node<Hospital*>* cur = residents->arr[i].getHospitalPreferences()->getHead();
6
7         // Add each resident to their top choice hospital
8         cur->data->addResident(&residents->arr[i], 0);
9     }
10
11     for (int i = 0; i < hospitals->length; i++) {
12         hospitals->arr[i].getAssignments()[0].printList();
13     }
14
15     // Go through each level of the hospitals
16     for (int i = 0; i < Hospital::NUM_LEVELS; i++) {
17         for (int j = 0; j < hospitals->length; j++) {
18             // Make sure the hospital is not over capacity
19             while (hospitals->arr[j].getNumAssignedRange(i) > hospitals->arr[j].getCapacity
20                 ()) {
21                 // Get the resident and remove the hospital from its list
22                 Node<Resident*>* res = hospitals->arr[j].getAssignments()[i].dequeue();
23                 Node<Hospital*>* h = res->data->getHospitalPreferences()->dequeue();
24                 delete h;
25
26                 // Add the resident to its next preferred hospital if possible
27                 if (i < Hospital::NUM_LEVELS - 1) {
28                     res->data->getHospitalPreferences()->getHead()->data->getAssignments()[i
29                         + 1].priorityAdd(res, i + 1);
30                 } else {
31                     delete res;
32                 }
33             }
34         }
35     }
36
37     // Formally assign each resident to the hospitals
38     for (int i = 0; i < hospitals->length; i++) {
39         for (int j = 0; j < Hospital::NUM_LEVELS; j++) {
40             Node<Resident*>* cur = hospitals->arr[i].getAssignments()[j].getHead();
41             while (cur != nullptr) {
42                 cur->data->setAssignment(&hospitals->arr[i]);
43                 cur = cur->next;
44             }
45         }
46     }
47
48     std::cout << "Finished_main_algo" << std::endl << std::endl;
49
50     // Print the final results
51     std::cout << "Initial_results:" << std::endl;
52     for (int i = 0; i < residents->length; i++) {
53         if (residents->arr[i].getAssignment() != nullptr) {
54             std::cout << "(" << residents->arr[i].getName() << ",_" << residents->arr[i].
55                 getAssignment()->getName() << ")" << std::endl;
56         } else {
57             std::cout << "(" << residents->arr[i].getName() << ",_nullptr)" << std::endl;
58         }
59     }
60 }
```

```

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                 // Compute the current average happiness among the 2 residents
79                 int iCurHappiness = 0;
80                 if (iHosp != nullptr) {
81                     iCurHappiness = residents->arr[i].getPreferencesArr()[iHosp->getIndex()];
82                 }
83
84                 int jCurHappiness = 0;
85                 if (jHosp != nullptr) {
86                     jCurHappiness = residents->arr[j].getPreferencesArr()[jHosp->getIndex()];
87                 }
88                 double curHappiness = (double) (iCurHappiness + jCurHappiness) / 2;
89
90                 // Compute the average happiness if the 2 residents swapped
91                 int iSwapHappiness = 0;
92                 if (jHosp != nullptr) {
93                     iSwapHappiness = residents->arr[i].getPreferencesArr()[jHosp->getIndex()];
94                 }
95
96                 int jSwapHappiness = 0;
97                 if (iHosp != nullptr) {
98                     jSwapHappiness = residents->arr[j].getPreferencesArr()[iHosp->getIndex()];
99                 }
100                 double swapHappiness = (double) (iSwapHappiness + jSwapHappiness) / 2;
101
102                 // Conduct a swap if needed
103                 if (swapHappiness > curHappiness) {
104                     // Remove the residents from the hospital lists
105                     if (iHosp != nullptr) {
106                         iHosp->removeResident(&residents->arr[i], Hospital::NUM_LEVELS -
107                             residents->arr[i].getPreferencesArr()[iHosp->getIndex()]);
108                     }
109                     if (jHosp != nullptr) {
110                         jHosp->removeResident(&residents->arr[j], Hospital::NUM_LEVELS -
111                             residents->arr[j].getPreferencesArr()[jHosp->getIndex()]);
112                     }
113                     // Add the j resident to the i hospital
114                     if (iHosp != nullptr) {
115                         iHosp->addResident(&residents->arr[j], Hospital::NUM_LEVELS -

```

```

116         residents->arr[j].getPreferencesArr()[iHosp->getIndex()]);
117     }
118     residents->arr[j].setAssignment(iHosp);
119     // Add the i resident to the j hospital
120     if (jHosp != nullptr) {
121         jHosp->addResident(&residents->arr[i], Hospital::NUM_LEVELS -
122             residents->arr[i].getPreferencesArr()[jHosp->getIndex()]);
123     }
124     residents->arr[i].setAssignment(jHosp);
125     // Increment swaps
126     swaps++;
127     std::cout << "Swapped_" << residents->arr[i].getName() << "_and_" <<
128         residents->arr[j].getName() << std::endl;
129     }
130     }
131 }
132 std::cout << "Swaps_made:" << swaps << std::endl;
133 }
134
135 // Print the final results
136 std::cout << std::endl;
137 std::cout << "Final_results:" << std::endl;
138 for (int i = 0; i < residents->length; i++) {
139     if (residents->arr[i].getAssignment() != nullptr) {
140         std::cout << "(" << residents->arr[i].getName() << ",_" << residents->arr[i].
141             getAssignment()->getName() << ")" << std::endl;
142     } else {
143         std::cout << "(" << residents->arr[i].getName() << ",_nullptr)" << std::endl;
144     }
145 }
146 std::cout << std::endl;
147
148 // Compute the happiness indices for both residents and hospitals
149 std::cout << "Resident_Happiness:" << computeResidentHappiness(residents) << std::endl;
150 std::cout << "Hospital_Happiness:" << computeHospitalHappiness(hospitals) << std::endl;
151 }

```

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