CMPT 435 - Fall 2022 - Dr. Labouseur

Final Project

Nicholas Fiore

Nicholas.Fiore@Marist.edu

December 3, 2022

1 Stable Marriage/Stable Matching

The idea of the stable marriage problem is having a set of men and a set of women (both sets being of equal size), in which all women and all men will rank the other set, and the goal is to create all stable pairings. A few assumptions made are that all pairings are heterosexual and monogamous. For all pairings to be stable, there can be no unmatched man and woman who prefer each other to their matched partner. For example, if man A likes woman C and women C likes woman A, while man B likes woman C and woman D likes man B but the pairings are (A, D) and (B, C), then these two are unstable as A and C prefer each other but are not matched. (A, C) and (B, D) are a stable pairing because although man B prefers woman C, woman D already prefers man B.

1.1 Hospitals and Residents Variation

There is a variation on this stable marriage/stable matching problem in which hospitals and residents are matched with each other. In this variation, hospitals have a certain capacity (so they can have more than one resident paired with them), but both hospitals and residents still rank each other.

There is an algorithm to achieve stable matches in this variation, which is what this lab describes, and can be found in the code listings (do note that the code listings for the Node and Queue classes are omitted, as their code only changed to function with Resident objects).

The program begins with a file parser, which just takes in a file and sets up the Residents and Hospitals into arrays, including the rankings for each. Then the algorithm begins. The algorithm starts by assuming all Residents are free (not assigned to a hospital) and all hospitals are unsubscribed (empty). Beginning with the first resident in the array, the resident's current top choice of hospital is checked if it is full. Because of how items were added to the arraylist of choices, the resident's choices will be in descending order, with the element at index 0 being their top *available* choice (more on that distinction later).

If the top choice of the resident is not full, the resident is added to that hospital, and they are marked as no longer being free. This is a provisional assignment, as it has the potential to change. After being added to a hospital, a post check is made to see if the hospital is now full after that resident is added. If it is in fact full, that hospital's resident ranking arraylist is checked in reverse. It will check every resident in ascending order until it finds a resident that is currently being considered (i.e. one that was provisionally assigned to the hospital). Once found, all residents after that last considered resident are removed from the hospital's ranking, and all residents that were removed have that hospital removed from their hospital ranking. The reason for this is that due to the hospital being full, the lowest ranked considered resident is the bar that needs to be passed to even be considered for the hospital, and no lower ranked residents will ever be considered for the hospital.

This process will continue to happen with every resident. A different branch will occur however if a resident has a currently full hospital as their top choice. Due to the post check described before, any resident that still has a full hospital as their top choice must be ranked higher than the currently lowest ranked resident being considered by that hospital. When this occurs, the hospital's ranking are checked until the lowest ranked resident being considered is found. As a double check, I also included an if statement that ensures that the resident that wants to be added is in fact higher ranked than the lowest ranked considered resident. If it passes, it moves on with bumping the lowest ranked resident from the hospital. A resident that has been bumped from a hospital is added to a queue of bumped residents and marked as free again. They also have the hospital which they were bumped from removed from their ranked choices. Due to how an arraylist works, this causes their second choice to shift into the index 0, which is why I described it always checking for their top *available* choice as sometimes their top choice is no longer available to them. The post-check then occurs as usual, before the process is repeated. It should be noted that the queue of bumped residents are processed first before the next resident within the array of residents.

This process will continue until every resident has been matched with a hospital. Residents are given their matched hospital in a $Hospital$ variable and the pairs of residents to hospitals are printed. These matches are stable due to the nature of limited capacity and ranking. While not every resident is able to get their first choice, they will at least be matched with a hospital that preferred them more over other candidates that tried to get into the hospital. It is the best possible fit for any given resident and hospital.

1.2 No Hospital Rank

An unsolved variation on the hospitals and residents ranking is one where hospitals no longer rank residents, but still have limited capacity, and residents still rank the hospitals. This version of the problem is difficult to solve due to how the definition of stability used in the original problem may not apply exactly and needs to be changed slightly.

The implementation of the algorithm for this problem is somewhat similar, is admittedly less complex than the original. The algorithm starts like usual, assigning residents to their top choice. What's different however is the post check. Since there are no ranks that the hospital can use to determine a better fit resident, when a hospital becomes full, ***all*** unmatched residents with that hospital in their rank will have that hospital removed from their rankings. This means that once a hospital is full, there is no longer any chance for any resident to be put in that hospital. The algorithm will then continue through the list, placing residents as needed and closing hospital choices as needed.

The definition of stability here is hard to describe. Technically, there is no way to tell how stable a pairing is due to hospitals not having a rank. If hospitals cannot "prefer" one resident over another resident, then *any* pairing could really be considered "stable". The largest problem that comes with my implementation of the algorithm to solve the problem is that it is inconsistent. Residents are basically assigned on a first come first serve basis, so if the order that the residents are accessed is changed, the pairings could end up completely different. This makes the pairings seemingly unstable, but if we subscribe to the idea that *any* pairing can technically be stable due to the lack of resident preference from hospitals, then it is, by that definition, a stable pairing. It is just that there are a lot of different pairing combinations that can be considered stable.

2 Appendix

2.1 MainProject.java

```java
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class MainProject {
    public static void main(String[] args) throws IOException {
        stableMatchingProblem("final-project-data.txt");
        System.out.println();
        stableMatchingProblemNoHospitalRank("final-project-data-no-hospital-rank.txt");
    }

    public static void stableMatchingProblem(String fileName) throws IOException {
        long totalLines = 0;
        File file = new File(fileName);
        BufferedReader input = null;
        String line;

        try {
            //gets the path of the current file in order to get the # of lines
            Path path = Paths.get(file.getName());

            input = new BufferedReader(new FileReader(fileName));

            totalLines = Files.lines(path).count();

            //instantiate lists for hospitals and residents
            Hospital[] hospitals = null;
            Resident[] residents = null;

            String[] lineArr = null; //variable to hold the word array of the current line

            //used when initializing resident/hospital objects
            int id;
            int capacity;

            //positions in the hospitals and residents arrays, respectively
            int hosPos = 0;
            int resPos = 0;

            //Command parsing. Goes through each line and determines what command is being used based on strings.
            for (int i = 0; i < totalLines; i++) {
                line = input.readLine();
                lineArr = line.split("␣");
                if (lineArr[0].compareTo("--") == 0) {
                    //do nothing; it is a comment
                } else if (lineArr[0].compareTo("Config:") == 0){ //initialize the array size of the total hospitals and residents
                    hospitals = new Hospital[Integer.parseInt(lineArr[2])];
                    residents = new Resident[Integer.parseInt(lineArr[1])];
                } else if (line.charAt(0) == ’r’) {
                    id = Integer.parseInt(lineArr[0].substring(1, lineArr[0].length() - 1));
                    residents[resPos] = new Resident(id);
                    for (int j = 1; j < lineArr.length; j++) {
                        residents[resPos].addHospital(Integer.parseInt(lineArr[j].substring(1)));
                    }

                    resPos++; //increments the position in the array
                } else if (line.charAt(0) == ’h’) { //initializes the rankings of each resident. First index is first choice, last is last choice.
```

```java
                    id = Integer.parseInt(lineArr[0].substring(1, lineArr[0].length() - 1));
                    capacity = Integer.parseInt(lineArr[1]);
                    hospitals[hosPos] = new Hospital(id, capacity);
                    for (int j = 3; j < lineArr.length; j++) { //initializes the rankings of each hospital. First index is first choice, last is last choice.
                        hospitals[hosPos].addResident(Integer.parseInt(lineArr[j].substring(1)));
                    }

                    hosPos++; //increments the position in the array
                }
            }

            //post file parsing method calls and other executions
            //immediate execution of the conversion methods for both lists
            for (int i = 0; i < hospitals.length; i++) {
                hospitals[i].idToObject(residents);
            }
            for (int i = 0; i < residents.length; i++) {
                residents[i].idToObject(hospitals);
            }

            ResQueue bumpedQueue = new ResQueue(); //queue for any residents that get bumped from a hospital
            for (int i = 0; i < residents.length; i++) {
                while (!bumpedQueue.isEmpty()) {
                    matchingAlgorithm(bumpedQueue, bumpedQueue.dequeue().getMyRes());
                }
                matchingAlgorithm(bumpedQueue, residents[i]);
            }

            System.out.println("Match:");
            for (int i = 0; i < hospitals.length; i++) { //adds all hospitals to their resident’s matchedHospital member
                Hospital currHos = hospitals[i];
                Resident currRes;
                for (int j = 0; j < currHos.getConsideredResidents().size(); j++) {
                    currRes = currHos.getConsideredResidents().get(j);
                    currRes.setMatchedHospital(currHos);
                }
            }

            for (int i = 0; i < residents.length; i++) {
                int resId = residents[i].getId();
                int hosId = residents[i].getMatchedHospital().getId();
                System.out.println("(r" + resId + ",␣h" + hosId + ")");
            }

        } catch(FileNotFoundException ex) {
            System.out.println("Failed␣to␣find␣file:␣" + file.getAbsolutePath());
        } catch(IOException ex) {
            System.out.println(ex.getMessage());
        } catch(NullPointerException ex) {
            System.out.println(ex.getMessage());
        } catch(Exception ex) {
            System.out.println("Something␣went␣wrong.");
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        } finally {
            if (input != null) {
                input.close();
            }
        }
    }

    //used in stableMatchingProblem()
    public static void matchingAlgorithm(ResQueue bumped, Resident res) {
        Hospital topChoice = res.getHospitalRank().get(0); //returns the current top available choice of the resident
        //initial addition
        if (!topChoice.isFull()) {
            topChoice.assignResident(res);
            res.setFree(false);
        } else {
            int i = topChoice.getResidentRank().size() - 1;
            Resident currRes = null;
            boolean resFound = false;
            while (i >= 0 && !resFound) {
                currRes = topChoice.getResidentRank().get(i);
                if (topChoice.isConsidering(currRes))
                    resFound = true;
                else
                    i--;
            }
            if (topChoice.getResidentRank().indexOf(res) < topChoice.getResidentRank().indexOf(currRes)) { //ensures that the new resident is ranked higher before bumping the old resident
                //bump lower ranked resident
                topChoice.bumpResident(currRes);
                bumped.enqueue(new ResNode(currRes));

                //add new resident
                topChoice.assignResident(res);
                res.setFree(false);
            }
        }

        //post check. If a hospital was just filled up, all residents after its last ranked resident being considered are dropped from the running.
        if(topChoice.isFull()) {
            int i = topChoice.getResidentRank().size() - 1;
            Resident currRes = null;
            boolean resFound = false;
            while (i >= 0 && !resFound) {
                currRes = topChoice.getResidentRank().get(i);
                if (topChoice.isConsidering(currRes))
                    resFound = true;
                else
                    i--;
            }
            Resident removedRes = null;
            int removedResHospitalIndex;
            while (topChoice.getResidentRank().size() > i + 1) { //removes all elements in the list after index i
                removedRes = topChoice.getResidentRank().remove(i + 1);
                removedResHospitalIndex = removedRes.getHospitalRank().indexOf(topChoice);
                if (removedResHospitalIndex != -1) //if the removed resident has the hospital in its rankings, remove it
                    removedRes.getHospitalRank().remove(removedResHospitalIndex);
            }
        }
    }

    public static void stableMatchingProblemNoHospitalRank(String fileName) throws IOException {
        long totalLines = 0;
        File file = new File(fileName);
        BufferedReader input = null;
        String line;


        try {
```

```java
        //gets the path of the current file in order to get the # of lines
        Path path = Paths.get(file.getName());

        input = new BufferedReader(new FileReader(fileName));

        totalLines = Files.lines(path).count();

        //instantiate lists for hospitals and residents
        Hospital[] hospitals = null;
        Resident[] residents = null;

        String[] lineArr = null; //variable to hold the word array of the current line

        //used when initializing resident/hospital objects
        int id;
        int capacity;

        //positions in the hospitals and residents arrays, respectively
        int hosPos = 0;
        int resPos = 0;

        //Command parsing. Goes through each line and determines what command is being used based on strings.
        for (int i = 0; i < totalLines; i++) {
            line = input.readLine();
            lineArr = line.split("␣");
            if (lineArr[0].compareTo("--") == 0) {
                //do nothing; it is a comment
            } else if (lineArr[0].compareTo("Config:") == 0){ //initialize the array size of the total hospitals and residents
                hospitals = new Hospital[Integer.parseInt(lineArr[2])];
                residents = new Resident[Integer.parseInt(lineArr[1])];
            } else if (line.charAt(0) == 'r') {
                id = Integer.parseInt(lineArr[0].substring(1, lineArr[0].length() - 1));
                residents[resPos] = new Resident(id);
                for (int j = 1; j < lineArr.length; j++) {
                    residents[resPos].addHospital(Integer.parseInt(lineArr[j].substring(1)));
                }

                resPos++; //increments the position in the array
            } else if (line.charAt(0) == 'h') { //initializes the rankings of each resident. First index is first choice, last is last choice.
                id = Integer.parseInt(lineArr[0].substring(1, lineArr[0].length() - 1));
                capacity = Integer.parseInt(lineArr[1]);
                hospitals[hosPos] = new Hospital(id, capacity);

                hosPos++; //increments the position in the array
            }
        }

        //post file parsing method calls and other executions
        //immediate execution of the conversion methods for both lists
        for (int i = 0; i < hospitals.length; i++) {
            hospitals[i].idToObject(residents);
        }
        for (int i = 0; i < residents.length; i++) {
            residents[i].idToObject(hospitals);
        }

        ResQueue bumpedQueue = new ResQueue(); //queue for any residents that get bumped from a hospital
        for (int i = 0; i < residents.length; i++) {
            while (!bumpedQueue.isEmpty()) {
                matchingAlgorithmNoHospitalRank(bumpedQueue, bumpedQueue.dequeue().getMyRes(), residents);
                }
                matchingAlgorithmNoHospitalRank(bumpedQueue, residents[i], residents);
            }

            System.out.println("Match:");
            for (int i = 0; i < hospitals.length; i++) { //adds all hospitals to their resident's matchedHospital member
                Hospital currHos = hospitals[i];
                Resident currRes;
                for (int j = 0; j < currHos.getConsideredResidents().size(); j++) {
                    currRes = currHos.getConsideredResidents().get(j);
                    currRes.setMatchedHospital(currHos);
                }
            }

            for (int i = 0; i < residents.length; i++) {
                int resId = residents[i].getId();
                int hosId = residents[i].getMatchedHospital().getId();
                System.out.println("(r" + resId + ",␣h" + hosId + ")");
            }

        } catch(FileNotFoundException ex) {
            System.out.println("Failed␣to␣find␣file:␣" + file.getAbsolutePath());
        } catch(IOException ex) {
            System.out.println(ex.getMessage());
        } catch(NullPointerException ex) {
            System.out.println(ex.getMessage());
        } catch(Exception ex) {
            System.out.println("Something␣went␣wrong.");
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        } finally {
            if (input != null) {
                input.close();
            }
        }
    }

    public static void matchingAlgorithmNoHospitalRank(ResQueue bumped, Resident res, Resident[] residents) {
        Hospital topChoice = res.getHospitalRank().get(0); //returns the current top available choice of the resident
        //initial addition
        if (!topChoice.isFull()) {
            topChoice.assignResident(res);
            res.setFree(false);
        } else {
            bumped.enqueue(new ResNode(res));
        }

        //post check. If a hospital was just filled up, all residents after its last ranked resident being considered are dropped from the running.
        if(topChoice.isFull()) {

            int hosIndex;
            for (int i = 0; i < residents.length; i++){ //removes the hospital from all of the residents that have it ranked
                hosIndex = residents[i].getHospitalRank().indexOf(topChoice);
                if (hosIndex != -1)
                    residents[i].getHospitalRank().remove(hosIndex);
            }
        }
    }
}
```

2.2 Resident.java

```java
import java.util.ArrayList;

public class Resident {
    private int id;
    private ArrayList<Integer> hosRankInt = null;
    private ArrayList<Hospital> hospitalRank = null;
    private Hospital matchedHospital = null; //allows the matches to be printed in order based on residents rather than hospital. Only initialized after stability is reached.
    private boolean isFree;

    public Resident(int id) {
        this.id = id;
        hosRankInt = new ArrayList<>();
        hospitalRank = new ArrayList<>();
        this.isFree = true;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setFree(boolean isFree) {
        this.isFree = isFree;
    }

    public boolean isFree() {
        return isFree;
    }

    public ArrayList<Hospital> getHospitalRank() {
        return hospitalRank;
    }

    public void addHospital(int hospitalId) {
        hosRankInt.add(hospitalId);
    }

    public Hospital getMatchedHospital() {
        return matchedHospital;
    }

    public void setMatchedHospital(Hospital matchedHospital) {
        this.matchedHospital = matchedHospital;
    }

    //converts the arraylist of integer ids to a list of their corresponding Resident objects, given a list of residen objects. Uses a linear search to do so.
    public void idToObject(Hospital[] hosList) {
        hospitalRank = new ArrayList<>();
        for (int i = 0; i < hosRankInt.size(); i++) {
            hospitalRank.add(linearSearch(hosList, hosRankInt.get(i)));
        }
    }

    //modified linear search designed to find a Hospital based on an ID, and return the object itself
    public Hospital linearSearch(Hospital[] arr, int key) {
        int i = 0;
        while (i < arr.length && arr[i].getId() != key) {
            i++;
        }
        if (i >= arr.length)
            i = -1;
        return arr[i];
    }
}
```

2.3 Hospital.java

```
import java.lang.reflect.Array;
import java.util.ArrayList;

public class Hospital {
    private int id;
    private ArrayList<Integer> resRankInt = null; //specifically used while parsing. After the file is fully parsed, converted into the Resident ArrayList
    private ArrayList<Resident> residentRank = null;
    private ArrayList<Resident> consideredResidents = null;
    private int capacity;


    public Hospital(int id, int capacity) {
        this.id = id;
        this.capacity = capacity;
        resRankInt = new ArrayList<>();
        consideredResidents = new ArrayList<>();
    }

    public ArrayList<Resident> getResidentRank() {
        return residentRank;
    }

    public ArrayList<Resident> getConsideredResidents() {
        return consideredResidents;
    }

    public int getId() {
        return id;
    }

    public int getCapacity() {
        return capacity;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }

    public void addResident(int resident) {
        resRankInt.add(resident);
    }

    //assigns a resident tentatively to be considered by the hospital
    public void assignResident(Resident resident) {
        consideredResidents.add(resident);
    }

    //bumps a considered resident from consideration
```

```java
    public void bumpResident(Resident resident) {
        consideredResidents.remove(resident);
    }

    public boolean isFull() {
        boolean retVal = false;
        if (consideredResidents.size() >= capacity)
            retVal = true;
        return retVal;
    }

    //converts the arraylist of integer ids to a list of their corresponding Resident objects, given a list of residen objects. Uses a linear search to do so.
    public void idToObject(Resident[] resList) {
        residentRank = new ArrayList<>();
        for (int i = 0; i < resRankInt.size(); i++) {
            residentRank.add(linearSearch(resList, resRankInt.get(i)));
        }
    }

    //checks to see if a particular resident is being considered by linearaly searching the consideredResidents list
    public boolean isConsidering(Resident res) {
        boolean retVal = false;
        int i = 0;
        while (i < consideredResidents.size() && consideredResidents.get(i) != res) {
            i++;
        }
        if (i != consideredResidents.size() && consideredResidents.get(i) == res)
            retVal = true;
        return retVal;
    }

    //modified linear search designed to find a Resident based on an ID, and return the object itself
    public Resident linearSearch(Resident[] arr, int key) {
        int i = 0;
        while (i < arr.length && arr[i].getId() != key) {
            i++;
        }
        if (i >= arr.length)
            i = -1;
        return arr[i];
    }
}
```