CS 4346- Project #1 Report

Submitted by:

Bigyan Bhandari

10/04/2021

Table of Contents

Problem Description:

The Expert System we have developed helps with diagnosing the underlying heart disease based on the list of symptoms present in a patient. It is meant to be used by a healthcare professional where a healthcare professional answers a series of yes/no questions asked by the program and based on the various variables initialized depending on the responses, the backward chaining system in the program comes up with the possible diagnosis based on those symptoms.

Furthermore, once the diagnosis has been made, a possible list of treatments is also recommended by the program based on forward chaining principle.

Domain of the problem:

The domain of the problem is medical diagnosis and treatment. More precisely, the problem is concerned with heart diseases. Therefore, the domain of the problem is cardiology.

Methodologies and rules:

The program uses backward chaining to come up with the conclusion or the diagnosis of the disease based on symptoms told to be present or absent by the user of the program.

At the core of this implementation of backward chaining, we have a decision tree that was used to implement the rules or the if conditions which when satisfied triggered the diagnosis of a particular heart disease. Knowing that diseases share symptoms and that two diseases could have shared symptoms, we set about to find the unique combination of symptoms that would trigger a diagnosis of a particular disease.

Our research began on the internet, and we used the website of the Mayo clinic as our expert to collect various facts about heart diseases and their symptoms. We focused on the following heart diseases:

1. Heart Failure
2. Cardiomyopathy
3. Angina
4. Coronary Artery Disease
5. Tachycardia
6. Ventricular Tachycardia

Once we listed down each disease and their corresponding symptoms, we began a way to matching the symptoms with the disease so that we can come up with unique combination of symptoms for each disease. Each one of these unique combinations would act as a rule for our rule based backward chaining process. Altogether we came up with 36 rules for our backward chaining process.

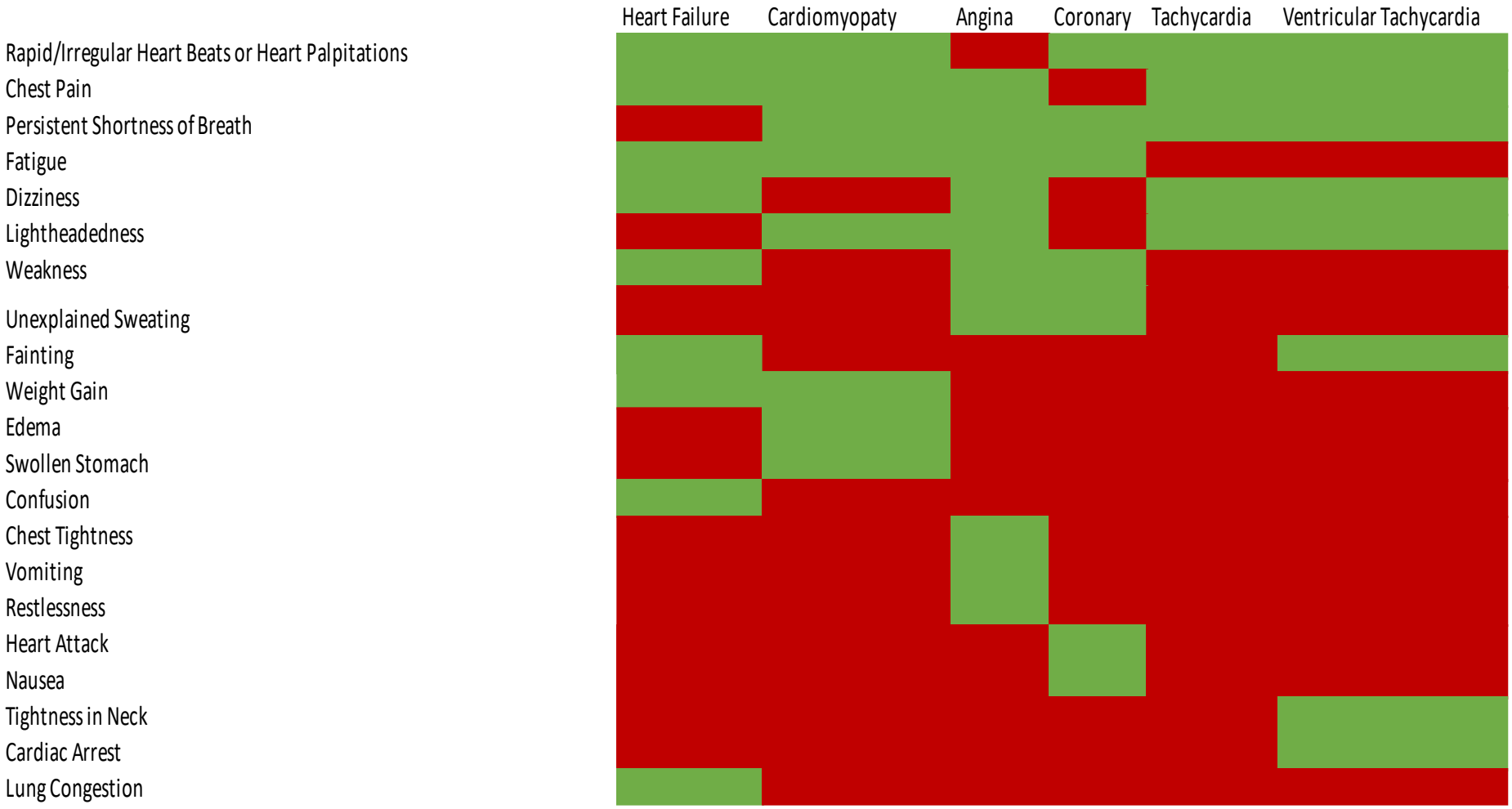


Fig 1. Developing a rule base for backward chaining to diagnose a heart disease. The rows represent the symptoms and the columns a particular diagnosis. Green rows in each column indicate the corresponding symptom to be present in the disease. We will develop our decision tree and rules for backward chaining based on this heatmap of symptoms. For example, absence of shortness of breath (marked by red in the shortness of breath row) along with the presence of fainting and confusion (both marked by green) would uniquely identify Heart Failure in our table.

Based on the heatmap in the previous page, and the indexing of symptoms done in the following manner we came up with the following list of if/then rules to diagnose the disease.

0	Rapid/Irregular Heart Beats or Heart Palpitations
1	Chest Pain
2	Persistent Shortness of Breath
3	Fatigue
4	Dizziness
5	Lightheadedness
6	Weakness
7	Unexplained Sweating
8	Fainting
9	Weight Gain
10	Edema
11	Swollen Stomach
12	Confusion
13	Chest Tightness
14	Vomiting
15	Restlessness
16	Heart Attack
17	Nausea
18	Tightness in Neck
19	Cardiac Arrest
20	Lung Congestion

Fig 2: The index table used for creating our if/then clauses

The 36 rules for diagnosis we came up with were as following:

		Should the patient be diagnosed?	
		RULES:	
10	IF	0	AND
	IF	1	AND
	IF	6	
	THEN	HEART FAILURE	
20	IF	0	AND
	IF	12	
	THEN	HEART FAILURE	
30	IF	0	AND
	IF	20	
	THEN	HEART FAILURE	
40	IF	0	AND
	IF	10	
	THEN	CARDIOMYOPATHY	
50	IF	0	AND
	IF	11	AND
	THEN	CARDIOMYOPATHY	
60	IF	0	AND
	IF	18	
	THEN	VENTRICULAR TACHYCARDIA	
70	IF	0	AND
	IF	19	
	THEN	VENTRICULAR TACHYCARDIA	
80	IF	1	AND
	IF	2	AND
	IF	6	
	THEN	ANGINA	
90	IF	1	AND
	IF	2	AND
	IF	7	
	THEN	ANGINA	
100	IF	1	AND
	IF	2	AND
	IF	8	
	THEN	VENTRICULAR TACHYCARDIA	

		Should the patient be diagnosed?	
		RULES:	
110	IF	1	AND
	IF	2	AND
	IF	9	
	THEN	CARDIOMYOPATHY	
120	IF	1	AND
	IF	10	
	THEN	CARDIOMYOPATHY	
130	IF	1	AND
	IF	11	
	THEN	CARDIOMYOPATHY	
140	IF	1	AND
	IF	13	
	THEN	ANGINA	
150	IF	1	AND
	IF	14	AND
	THEN	ANGINA	
160	IF	1	AND
	IF	15	
	THEN	ANGINA	
170	IF	1	AND
	IF	18	
	THEN	VENTRICULAR TACHYCARDIA	
180	IF	1	AND
	IF	19	
	THEN	VENTRICULAR TACHYCARDIA	
190	IF	1	AND
	IF	3	AND
	IF	7	
	THEN	ANGINA	
200	IF	1	AND
	IF	3	AND
	IF	8	
	THEN	HEART FAILURE	

		Should the patient be diagnosed?	
		RULES:	
210	IF	1	AND
	IF	20	
	THEN	HEART FAILURE	
220	IF	1	AND
	IF	5	AND
	IF	6	
	THEN	ANGINA	
230	IF	1	AND
	IF	5	AND
	IF	7	
	THEN	ANGINA	
240	IF	1	AND
	IF	5	AND
	IF	9	
	THEN	CARDIOMYOPATHY	
250	IF	1	AND
	IF	12	
	THEN	HEART FAILURE	
260	IF	1	AND
	IF	6	AND
	IF	9	
	THEN	HEART FAILURE	
270	IF	1	AND
	IF	6	AND
	IF	7	
	THEN	ANGINA	
280	IF	1	AND
	IF	6	AND
	IF	8	
	THEN	HEART FAILURE	
290	IF	1	AND
	IF	7	
	THEN	ANGINA	
300	IF	1	AND
	IF	8	AND
	IF	9	
	THEN	HEART FAILURE	

	Should the patient be diagnosed?		
	RULES:		
310	IF	0	AND
	IF	16	
	THEN	CAD	
320	IF	0	AND
	IF	17	
	THEN	CAD	
330	IF	0	AND
	IF	7	
	THEN	CAD	
340	IF	0	AND
	IF	8	AND
	IF	9	
	THEN	CAD	
350	IF	1	AND
	IF	4	AND
	IF	7	
	THEN	ANGINA	
360	IF	1	AND
	IF	4	AND
	IF	9	
	THEN	HEART FAILURE	

The 36 rules are numbered as multiples of 10 and the first column in these tables correspond to the rule number. The second column for each rule has at least two and at most three ‘if’ clauses whose numbers in the third column correspond to the index of symptoms. The way to read these rules be, for example in rule 360:

If symptom index 1 - Chest Pain, symptom index 4 – Dizziness and symptom index 9 – Weight gain were initialized as true, our backward chaining program would then give a diagnosis of Heart Failure.

Similarly, for the forward chaining program to recommend a course of treatment, we referred the website of Mayo clinic as our expert to create our clause variable list which would be instantiated based on the diagnosis provided by the backward chaining program.

Once a disease is known, its effects on various organs/ parts of body / bodily functions are known or instantiated. Based on this instantiation of effect variables (effect variables because they are the result of having that heart disease) we can further instantiate treatment variables as particular effect variable would be effectively cured by a particular treatment variable.

With this line of reasoning, we set out to develop rules for forward chaining.

	Effect Variable	Situation	Solution	Treatment
HEART FAILURE	High Blood Pressure and Progression of Disease	HIGH	NORMALIZE	ACE INHIBITOR / ARB BLOCKERS
	Fluid Collection	COLLECTED	DRAIN	DIURETICS / ALDOSTERONE ANTAGONISTS
CARDIOMYOPATHY	Inflammation	INFLAMED	NORMALIZE	CORTICOSTEROIDS
	Poor Blood Circulation	REDUCED	INCREASE	ACE INHIBITOR/ ANGIOTENSIN II RECEPTOR BLOCKERS
ANGINA	Narrow blood vessels and chest pain	NARROWED	RELAX	NITRATES
	Elevated Cholesterol	ELEVATED	REDUCE	STATIN
	Blood Clotting	CLOT	THINNING	CLOT PREVENTING DRUGS (CLOPIDOGREL, TICAGRELOR)/ ASPIRIN
CORONARY ARTERY DISEASE	High Blood Pressure and Progression of Disease	HIGH	NORMALIZE	ACE INHIBITOR / ARB BLOCKERS
	Blood Clotting	CLOT	THINNING	CLOT PREVENTING DRUGS (CLOPIDOGREL, TICAGRELOR)/ ASPIRIN
TACHYCARDIA	Increased Heart Rate	INCREASED	NORMALIZE	VAGAL MANEUVER
	Blood Clotting	CLOT	THINNING	CLOT PREVENTING DRUGS (CLOPIDOGREL, TICAGRELOR)/ ASPIRIN
VENTRICULAR TACHYCARDIA	Increased Heart Rate	INCREASED	NORMALIZE	ANTI-ARRYTHMIC DRUGS
	Blockages in blood vessels	BLOCKAGE	CLEAR	NITRATES/ OPEN HEART SURGERY

Fig 4. Variables for forward chaining. Beginning with a diagnosis in column 1, which would be known once the backward chaining program has been run, each diagnosis will instantiate effect variable in column 2 with the value as presented in the situation column. Each effect variable once initiated will methodically instantiate the treatment variable. Our forward chaining program will then employ a specific algorithm to traverse through these lists of effect and solution variables and recommend a particular course of treatment for a given diagnosis.

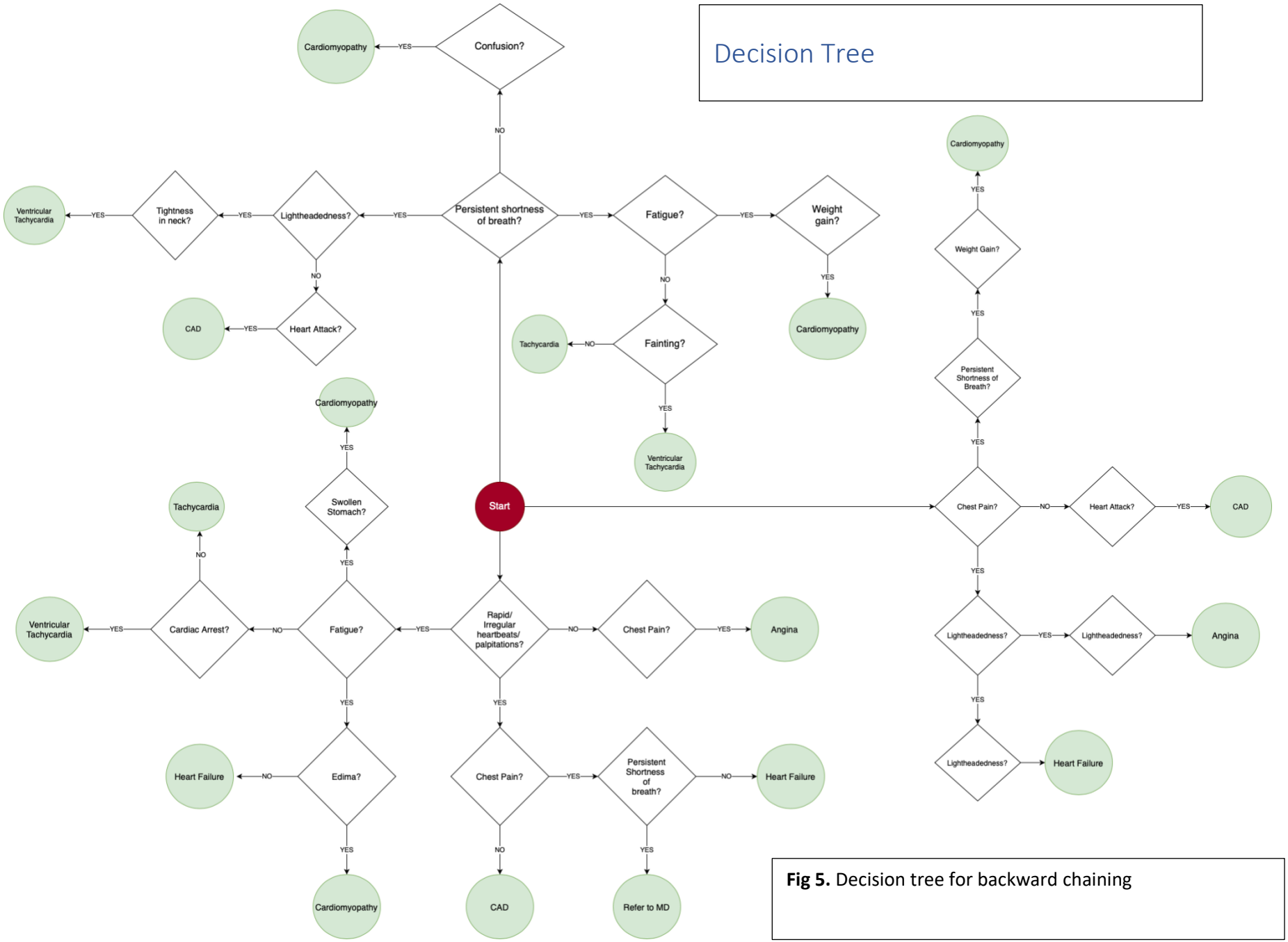


Fig 5. Decision tree for backward chaining

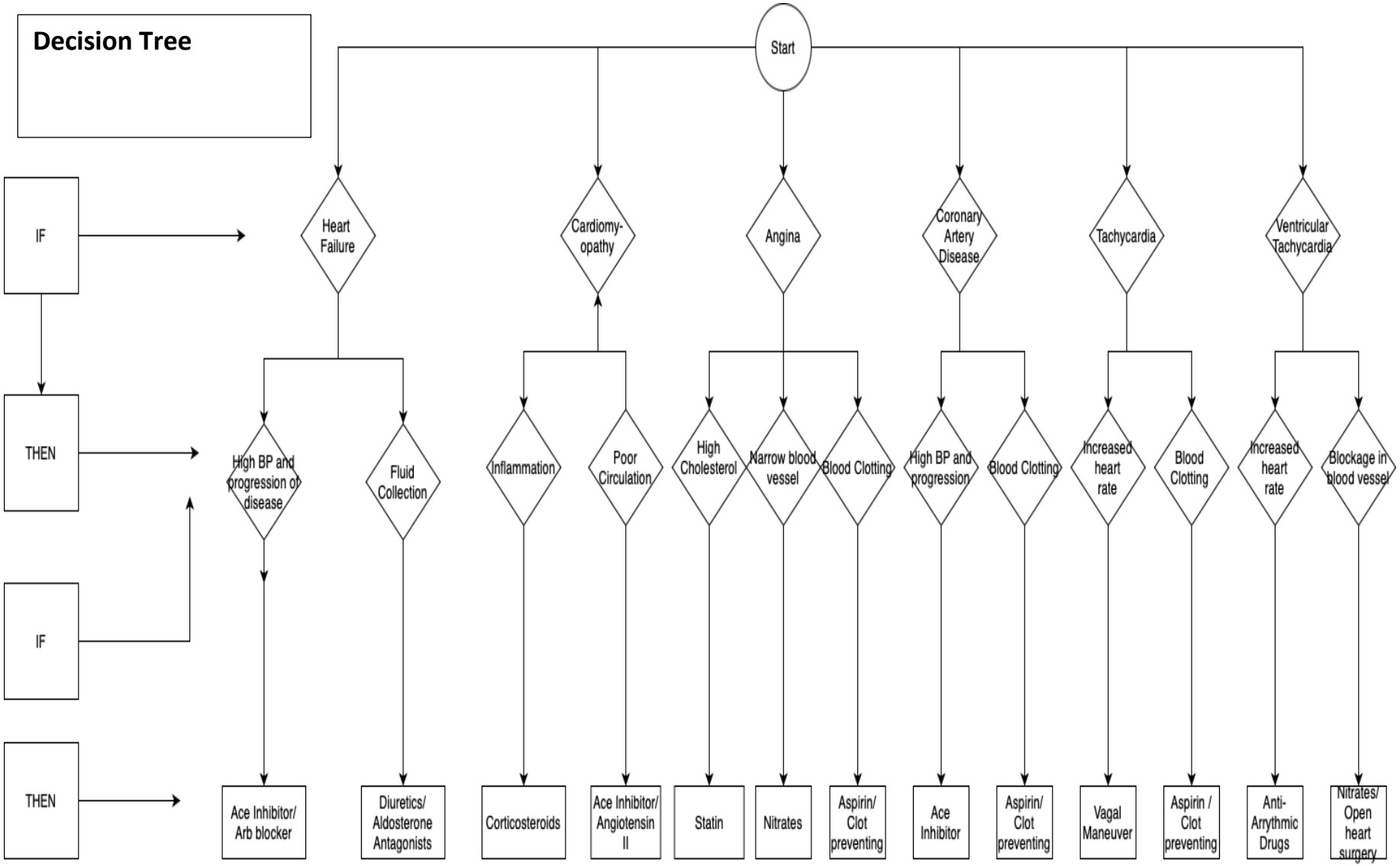


Fig 6. Decision tree for forward chaining rules

The rules for forward chaining can easily be traced from the decision tree unlike the backward chaining decision tree. As soon as a disease in instantiated, the if condition for effect variables, level two in our decision tree immediately below the disease names are instantiated as true. The instantiation of effect variable in turn instantiates the 'If' rules which lead to 'then' conditions to be instantiate – in our diagram the last level of conditions. Since there are no more 'then' conditions to be triggered, our forward chaining rule outputs these instantiated treatment variables as suggested course of treatment. We began with the instantiation of a disease that led to a domino effect of instantiation of effect which further led to the instantiation of treatment and when this domino effect ends, our program outputs these variables as course of treatment for the given disease.

Program Implementation

The implementation of the program is done in two parts. There are specific data structures used depending on the needs of the program and the choice of the algorithm.

Let us begin by a detailed description of the implementation of backward chaining.

Backward Chaining Implementation

Data Structures

The knowledge base for backward chaining implementation is initialized and stored in a *knowledge_base* class. Separate vectors of strings are used to store possible diagnosis as well symptoms *variables_list*.

The rules associated with particular diagnosis, the 'if' clauses of 'if-then' rules are stored in a *symptoms_combination* vector of integers where each integer refers to the index of symptom name in the variables list. Then a *clause_variable_list* is initialized where for each possible disease, only those variables/ symptoms which are needed to diagnose particular disease are stored but not instantiated. The *clause_variable_list* is initialized during the program run based on the responses of the user.

Initially, all variables in the *clause_variable_list* are initialized as -1 – which is interpreted as uninitialized state by the program. When the user answers yes if that clause/symptom is present in a patient, that variable in the *clause_variable_list* as well as the *variables_initialized* list is instantiated as 1 and 0 if the answer is no.

Algorithm for Backward chain

The *rules* class is the main driver of the backward chain program. Using the *knowledge_base* object, it has the access to the knowledge_base class in order for it to fetch what questions to ask to the user, to check if instatiated variables based on the responses of the user lead to the diagnosis or if it needs to ask further questions until all possibilities are exhausted or a diagnosis is reached.

The start_iteration method initiates the program. It begins by checking if the conclusion stack has any rules and variables that need to be instantiated next. If

the stack is empty, which happens at the beginning or end of the program – if at the beginning of the program it is hardwired to push the first rule and the first symptom on the top of the stack which is fed back to the start_iteration method.

Based on what is at the top of the conclusion stack, the user is asked if that symptom is present in the patient. The program can only accept yes or no as an answer and depending on the response it initializes that symptom or variable accordingly. After getting such response each time, the program calls *process_response* method and checks if all the if conditions of the current rule are fulfilled. Process_response method is provided with the rule number currently being worked on. It checks the rules_symptoms map and figures whether all symptoms variables have been initialized. If all symptoms variables are initialized, a diagnosis can be made or ruled out. If diagnosis is made, we have reached the end of our program. If the diagnosis is ruled out, because one of the if clause turns out to be false, we go the next rule in the rules list, update the stack and call *start_Iteration* method one more time.

Forward Chaining Implementation

Data Structures

The major difference between the forward chain and backward chain program is the used of queue over stack in the forward chain program. The forward_knowledge class has a list of all clauses which could potentially be initialized depending on the diagnosis provided by the backward chain program.

Once the disease has been initialized, it then initializes other variables in the clause_variable list, these are the part of the if then rule base where ‘if’ is the

diagnosis and ‘then’ follow the possible effects of that diagnosis. These initialized variables are placed in a variable_initialized queue.

In contrast to the backward chaining where next rule and symptom to be initialized were placed in a Last-In-First-Out stack data structure, the initialized variables are placed in a First-In-First-Out queue data structure.

Algorithm for forward chain

Initialize_forward_rule is called by the backward chain program once the diagnosis has been made using backward chain. Based on the diagnosis passed to the initialize method it calls an initialize_ailment method which initializes clause variables list based on if-then logic hardwired according to the previous describe decision tree. Every time a clause variable is initialized, it is added to the back of the variables_initialized queue. When there are no more clause_variables that need to be initialized, the initialize ailment calls the apply_forward_chain method passing the queue as an argument.

The apply_forward_chain method then methodically pops each variable from the queue and checks the ailment_treatment list where another ‘if-then’ rule is hardwired where if part represents the clause_variables in the queue and the then part represents the treatment. This brings to the end of the forward chain part of our program but for a more complex forward chaining we could further enque these treatment in the variables initialized queue and these treatments themselves could form an if part of an if-then clause where the then clause could possibly represent the potential sideeffects of the given treatment and instruct the user of the program to check for adverse effects of that prescribed treatment and what to do in such case.

Source Code:

Main Program:

```
#include <iostream>
#include "rules.h"



int main(){


  std::cout << "Welcome to the Diagnosis-Treatment program for Heart
Diseases!\n\n\n";

  Rules heart_disease_diagnosis = Rules();


  heart_disease_diagnosis.start_iteration();



  return 0;
};
```

Knowledge base for backward chaining:

```
#include <vector>
#include <stack>
#include <map>
#include <string>
#include <unordered_map>




class Knowledge_base{

  public:


  // all possible conclusions for our program

  std::vector<std::string>conclusions = {"Heart Failure", "Cardiomyopathy",
"Angina",   "Coronary   Aretry   Disease",   "Tachycardia",   "Ventricular
Tachycardia"};

  // variables_list is the list of all symptoms

  std::vector<std::string>variables_list = {"Rapid/Irregular Heart Beats or
Heart  Palpitations",  "Chest  Pain",  "Persistent  Shortness  of  Breath",
"Fatigue",    "Dizziness",    "Lightheadedness","Weakness",    "Unexplained
Sweating",   "Fainting","Weight   Gain",   "Edema",   "Swollen   Stomach",
"Confusion", "Chest Tightness", "Vomiting","Restlessness", "Heart Attack",
"Nausea", "Tightness in Neck", "Cardiac Arrest", "Lung Congestion"};

  /*
 list of all possible symptoms combination that leads to a diagnosis

  */

  std::vector<std::vector<int>>symptoms_combination = {
  {0,1,6},
  {0,12},
  {0,20},
  {0,10},
  {0,11},
  {0,18},
  {0,19},
  {1,2,6},
  {1,2,7},
  {1,2,8},
  {1,2,9},
  {1,10},
  {1,11},
  {1,13},
  {1,14},
```

```
  {1,15},
  {1,18},
  {1,19},
  {1,3,7},
  {1,3,8},
  {1,20},
  {1,5,6},
  {1,5,7},
  {1,5,9},
  {1,12},
  {1,6,9},
  {1,6,7},
  {1,6,8},
  {1,7},
  {1,8,9},
  {0,16},
  {0,17},
  {0,7},
  {0,8,9},
  {1,4,7},
  {1,4,9}
};

// clauses for diseases, numerical value represents
  // index in the variables list

  std::vector<int> heart_failure_clause = {0,1,3,4,6,8,9,12,20};

  std::vector<int> cardiomyopathy_clause = {0,1,2,3,5,9,10,11};

  std::vector<int> angina_clause = {1,2,3,4,5,6,7,13,14,15};

  std::vector<int> coronary_clause = {0,2,3,6,7,16,17};

  std::vector<int> tachycardia_clause = {0,1,2,4,5};

  std::vector<int> ventricular_tachycardia_clause = {0,1,2,4,5,8,18,19};

  //pushing all clauses in the clause index vector for easy retrieval later

  std::vector<std::vector<int>>  clause_index  =  {  heart_failure_clause,
cardiomyopathy_clause, angina_clause, coronary_clause, tachycardia_clause,
ventricular_tachycardia_clause};




  std::map<std::string,             int>variable_initialized             =
variable_list_initializer(variables_list);


  std::vector<int>clause_variable_list  =  initialize_clause_variable_list(
20, clause_index);

```

```
/*
 the variable list initializer function returns a map of variable and their
initialization status, -1 means the variable has not been initialized, 0
means the variable is false and 1 means the variable is true

*/


std::map<std::string,            int>            variable_list_initializer(
std::vector<std::string> variables_list ){
  std::map<std::string, int> initialized;

  for (std::string variable : variables_list){
    std::pair<std::string, int> key_value;
    key_value = std::make_pair(variable, -1);
    initialized.insert(key_value);
  }

  return initialized;
}



std::vector<int>     initialize_clause_variable_list     (int     numRules,
std::vector<std::vector<int>>clause_index){

  std::vector<int>clause_variable_list(numRules * clause_index.size());

  for (int i = 0; i < clause_index.size(); i++ ){
    int begin_at = i * numRules;
    for ( auto variable_index : clause_index[i]){
      clause_variable_list[begin_at + variable_index] = 1;
    }
  }
  return clause_variable_list;

}


};


```

Rule class for backward-chaining program:

```
#include <stdio.h>
#include <vector>
#include <stack>
#include <map>
#include <unordered_map>
#include "knowledge_base.h"
#include "forward_rules.h"
/*
Index used for Diagnoses ( our goals):
0: Heart Failure
1: Cardiomyopathy
2: Angina
3: Coronary
4: Tachycardia
5: Ventricular Tachycardia


Index used for Clause / Symptons ( our states) :

0: "Rapid/Irregular Heart Beats or Heart Palpitations"
1: "Chest Pain"
2: "Persistent Shortness of Breath"
3: "Fatigue"
4: "Dizziness"
5: "Lightheadedness"
6: "Weakness"
7: "Unexplained Sweating"
8: "Fainting"
9: "Weight Gain"
10: "Edema"
11: "Swollen Stomach"
12: "Confusion"
13: "Chest Tightness"
14: "Vomiting"
15: "Restlessness"
16: "Heart Attack"
17: "Nausea"
18: "Tightness in Neck"
19: "Cardiac Arrest"
20: "Lung Congestion"
*/




class Rules{

public:
  Rules() {};

```

```
private:

    std::string final_diagnosis;

/*
The rule-symptom map initialized has three parts to it.
The first int refers to the rule number in our knowledge base and the pair
on the second part has a vector and an int. First vector references the list
of symptoms that have to have been initialized and the int references the
diagnosis. eg.  {10, ({2, 4, 5}, 5) } means that the corresponding rule
number is 10; the index of symptoms that need to be true is {2, 4, 5} and
the diagnosis is index 5 of the diagnoses list.
*/

std::vector<int>rule_number = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110,
120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 260,
270, 280, 290, 300, 310, 320, 330, 340, 350, 360};

// for the diagnosis_index, the value at index i corresponnds to the
diagnosis that would be given by the rule at index i of vector<int>
rule_number

std::vector<int>                    diagnosis_index                    =
{0,0,0,1,1,5,5,2,2,5,1,1,1,2,2,2,5,5,2,0,0,2,2,1,0,0,2,0,2,0,3,3,3,3,2,0};


// for the symptoms_combination the vector at index i is the combination of
symptoms that need to be true to trigger a diagnosis at index i of the
diagnosis_index


std::vector<std::vector<int>>symptoms_combination = {
  {0,1,6},
  {0,12},
  {0,20},
  {0,10},
  {0,11},
  {0,18},
  {0,19},
  {1,2,6},
  {1,2,7},
  {1,2,8},
  {1,2,9},
  {1,10},
  {1,11},
  {1,13},
  {1,14},
  {1,15},
  {1,18},
  {1,19},
  {1,3,7},
  {1,3,8},
  {1,20},
  {1,5,6},
```

```
  {1,5,7},
  {1,5,9},
  {1,12},
  {1,6,9},
  {1,6,7},
  {1,6,8},
  {1,7},
  {1,8,9},
  {0,16},
  {0,17},
  {0,7},
  {0,8,9},
  {1,4,7},
  {1,4,9}
};

//rule_symptoms map contains rule as key and associate symptoms as value


std::map<      int,      std::vector<int>      >      rule_symptoms      =
initialize_rule_symptoms(rule_number, symptoms_combination);

// conclusion list is the objective/conclusion of each rule. In our case,
we are looking to diagnose certain heart condition in each rule, so
'diagnosis' would be a conclusion for each of our rules and we initialize
our conclusion list accordingly


std::vector<std::pair<int,std::string>>conclusion_list                    =
initialize_conclusion_list(36);

//visited conclusion list keeps track to what rules we have visited already
and ruled out the possibility of diagnosis. It gets updated when we visit
different conclusion list elements

std::vector<std::pair<int,std::string>>visited_conclusion_list;

// conclusion stack to keep track of what rule we need to go to next


std::stack<std::pair<int, int>>conclusion_stack = initialize_stack();

std::stack<std::pair<int, int>> initialize_stack () {
  std::stack<std::pair<int, int>>conclusion_stack;
  conclusion_stack.push(std::make_pair(10,0));
  return conclusion_stack;
}



// initialize_conclusion_list

std::vector<   std::pair<int,std::string>   >initialize_conclusion_list(int
numRules){
```

```

  std::vector<std::pair<int, std::string > > rules_conclusions;

  for ( int i = 1; i <= numRules; i++){
    std::pair< int, std::string> tempPair;
    tempPair = std::make_pair(i, "diagnosis");
    rules_conclusions.push_back(tempPair);
  }

  return rules_conclusions;
}

// initialize rule_symptoms map

std::map<     int,     std::vector<int>     >     initialize_rule_symptoms(
std::vector<int> rules, std::vector<std::vector<int>> symptoms){
    std::map< int, std::vector< int> > rule_symptoms;

    if ( rules.size() != symptoms.size()){
        printf("Rules and symptoms size mismatch. Recheck data.\n");
        return rule_symptoms;
    }

    for ( int i = 0; i < rules.size(); i++){
        rule_symptoms[rules[i]] = symptoms[i];
    }

        return rule_symptoms;
}

// Initialize a knowledge_base object

Knowledge_base current_knowledge = Knowledge_base();


public:

/*
Main logic-control area of the program here. After all data structures and
variables have been initialized we are ready to implement the rules.


*/

std::pair<int, int> check_conclusion_stack(){

  if (conclusion_stack.empty()){
    conclusion_stack.push( std::make_pair(10, 0));
  }

  return conclusion_stack.top();
}

/*
```

```

Start iteration checks the top of the stack and returns the rule number and
the conclusion index number that needs to be processed by the program. It
means the check_conclusion_stack function tells our program which rule and
which variable are we working on next to get response from or to check if
diagnosis is possible.

*/


void start_iteration(){
    std::string response;
    int response_value;

    // check_conclusion_stack to determine which rule and variable to get
response for


    std::pair<int, int> rule_to_process = check_conclusion_stack();


    //int rule_number = rule_to_process.first;
    int variable_index = rule_to_process.second;



    while ( (response != "Y") && (response != "N") && (response != "y") &&
(response != "n") ){
        std::string                      condition                      =
current_knowledge.variables_list[variable_index];
        printf("Does   the   patient   have   %s?      enter:   Y/N\n",
condition.c_str());
      std::cin >> response;
    }

    /*
     a function to read in an input from gui

    */
    if ( response == "Y" || response == "y"){
      response_value = 1;
    } else {
      response_value = 0;
    }

    updateResponse(response_value, variable_index);

}

/*
 startResponse has gotten a response whether a symptom can be initialized
as 0 if it is not present in a patient
 and 1 if a patient shows the symptom. updateResponse now updats that
response and calls process response for the
```

```
 rule that is at the top of the stack

*/

void updateResponse( int variable_value, int variable_position){


current_knowledge.variable_initialized[current_knowledge.variables_list[va
riable_position]] = variable_value;


    std::pair<int, int> rule_to_process = check_conclusion_stack();

    int rule_num_to_process = rule_to_process.first;

    processResponse(rule_num_to_process);
}

/*

 process response is provide with rule number. It checks the rules_symptoms
map and figures whether all symptoms variable have been
 initialized. If all symptoms variables are initialized, a dignosis can be
made or ruled out. If diagnosis is made, we have reached
 the end of our program. If the ddiagnosis is ruled out, we go the next rule
in the rules list, update the stack and call startIteration.
 If the decision can't be reached, it means some variable still need to be
initialized. If such is the case, we call start iteration for
 the same rule number and new symptom index. The stack is updated.

*/



void processResponse( int rule_num_to_process){

        int next_rule = -1;
        std::vector<int>symptoms_for_rule                                =
rule_symptoms[rule_num_to_process];



        for ( int symptom : symptoms_for_rule){


             if
(current_knowledge.variable_initialized[current_knowledge.variables_list[s
ymptom]] == 0){

           // the diagnosis cannot be true
           // check next set of rules
           // remove top item from conclusion stack
           // break from the for loop
```

```
                    visited_conclusion_list.push_back({rule_num_to_process,
"diagnosis"});
                    conclusion_stack.pop();

                    next_rule = ( rule_num_to_process + 10 );

                   if ( next_rule > rule_number[rule_number.size() - 1] ){
                       // We have reached the end without finding a diagnosis
--  print out a message and return
                          printf("Diagnosis not possible at this time. Please
refer to an MD.\n");
                          endProgram();
                       }

                   std::vector<int>          next_set_of_symptom          =
rule_symptoms[next_rule];

                   conclusion_stack.push({next_rule,
next_set_of_symptom[0]});
                   processResponse(next_rule);
             }



         /*
         if  symptom  is  -1  that  means  current  rule  has  not  been  fully
processed. We roll back the next rule variable as the code below
         processes  the  same  rule-symptom  block  until  we  initialize  all
variables for the rule and can reach to the conclusion

         */

          else                              if                              (
current_knowledge.variable_initialized[current_knowledge.variables_list[sy
mptom]] == -1) {
             next_rule = rule_num_to_process;
              conclusion_stack.pop();
              conclusion_stack.push({next_rule, symptom});
              start_iteration();
         }

     }



     if ( next_rule == -1){

         printf("\n\nThe patient is showing the following symptoms:\n\n");

         for ( auto each_symptom : rule_symptoms[rule_num_to_process]){
             std::string               symptom_description               =
current_knowledge.variables_list[each_symptom];
             printf("%s\n", symptom_description.c_str());
         }
```

```
        final_diagnosis                                                 =
current_knowledge.conclusions[diagnosis_index[(rule_num_to_process/  10)  -
1]];

        printf("\n\nThe patient might be suffering from %s. Please perform
appropriate  tests  to  confirm  and  treat  for  the  condition.\n\n",
final_diagnosis.c_str());
        endProgram();

    }


}

void endProgram(){
    Forward_Rules treatment = Forward_Rules( getDiagnosis());

    treatment.initialize_forward_rule();

}


// get diagnosis will be called by the forward chain program to recommend
the course of treatment
std::string getDiagnosis(){
        return this->final_diagnosis;
    }

};
```

Knowledge-base for forward-chaining program

```
//
//  forward_chain.h
//  BackwardChain
//
//  Created by Bigyan Bhandari on 9/18/21.
//

#ifndef forward_chain_h
#define forward_chain_h

#endif /* forward_chain_h */


class ForwardChain {

    public:



    std::vector<std::string> clause_variable_list = {"High blood pressure
and progression of disease",
                                                "Fluid Collection",
                                                "Inflammation",
                                                "Poor blood circulation",
                                                "Narrow blood vessels and
chest pain",
                                                "Elevated Cholesterol",
                                                "Blood Clotting",
                                                "Increased Heart Rate",
                                                "Blockage in blood
vessels"};

    std::vector<std::string> treatment_list = { "ACE INHIBITOR / ARB
BLOCKERS" ,
                                            "DIURETICS / ALDOSTERONE
ANTAGONISTS"
                                            "CORTICOSTEROIDS",
                                            "ACE INHIBITOR/ ANGIOTENSIN II
RECEPTOR BLOCKERS",
                                            "NITRATES",
                                            "STATIN",
                                            "VAGAL MANEUVER",
                                            "CLOT PREVENTING DRUGS (
CLOPIDOGREL, TICAGRELOR)/ ASPIRIN",
                                            "ANTI-ARRYTHMIC DRUGS",
                                            "NITRATES/ OPEN HEART SURGERY"
};
```

```
    std::vector<std::string>    diagnosis_list    =    {"Heart    Failure",
"Cardiomyopathy",  "Angina",  "Coronary  Artery  Disease",  "Tachycardia",
"Ventricular Tachycardia"};

    std::vector<std::pair<std::string,                        std::string>>
variable_initialized_list = variable_initializer(clause_variable_list);


    std::vector<std::pair<std::string,
std::string>>variable_initializer(std::vector<std::string> variable_list){
        std::vector<      std::pair<std::string,      std::string>      >
temp_var_initializer;

        for ( std::string variable : variable_list){
            std::pair<std::string,      std::string>      temp_pair      =
std::make_pair(variable, "");
            temp_var_initializer.push_back(temp_pair);
        }

        return temp_var_initializer;
    }

    //

    std::vector< std::pair< int, std::vector<int>> >diagnosis_ailment = {
        {0, {0,1}}, {1, {2, 3}}, {2, {4, 5, 6}}, {3, {0,6}}, {4, {6,7}},
{7, {7,8}}   };

    std::vector < std::vector<std::string> > ailment_condition = { {"HIGH",
"COLLECTED"}, {"INFLAMED", "REDUCED"}, {"NARROWED", "ELEVATED", "CLOT"},
{"HIGH", "CLOT"}, {"INCREASED", "CLOT"}, {"INCREASED", "BLOCKAGE"} };

    std::vector<std::pair<  std::vector<std::string>,  std::string  >  >
ailment_condition_treatment = {
        {{clause_variable_list[0],   "HIGH"},   "ACE-INHIBITOR   /   ARB
BLOCKERS"},   {{clause_variable_list[1],   "COLLECTED"},   "DIURETICS   /
ALDOSTERONE   ANTAGONISTS"},   {{clause_variable_list[2],   "INFLAMED"},
"CORTICOSTEROIDS"}, {{clause_variable_list[3], "REDUCED"}, "ACE INHIBITOR /
ANGIOTENSIN II RECEPTOR BLOCKERS"}, {{clause_variable_list[4], "NARROWED"},
"NITRATES"},    {{clause_variable_list[5],    "ELEVATED"},    "STATIN"},
{{clause_variable_list[6], "CLOT"}, "CLOT PREVENTING DRUGS (CLOPIDOGREL,
TICAGRELOR) OR ASPIRIN"}, {{ clause_variable_list[7], "INCREASED"}, "VAGAL
MANEUVER"} };


};
```

Rule-class for forward chaining

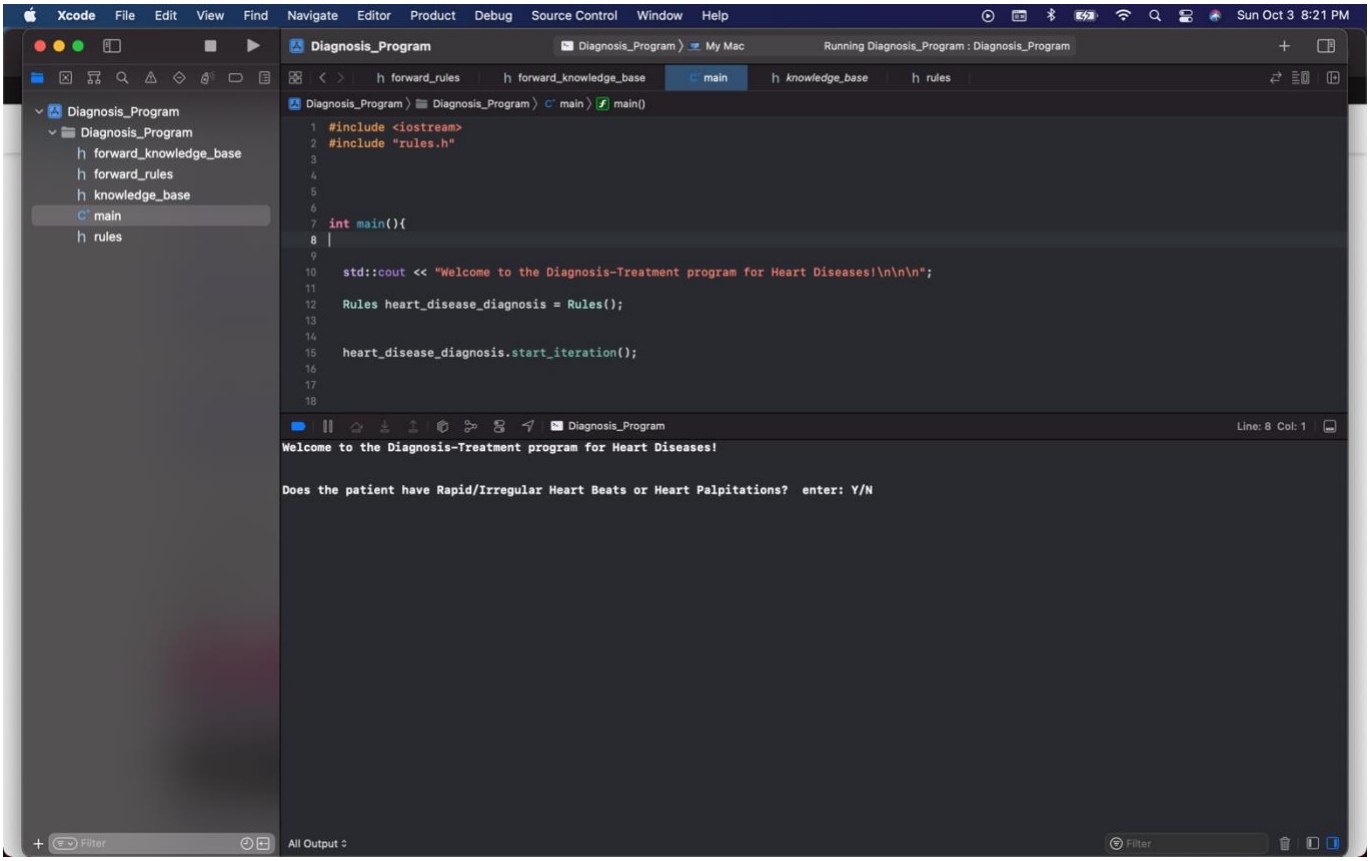
```
    //
//  forward_rules.h
//  BackwardChain
//
//  Created by Bigyan Bhandari on 9/21/21.
//

#ifndef forward_rules_h
#define forward_rules_h


#endif /* forward_rules_h */

#include <queue>
#include "forward_knowledge_base.h"

class Forward_Rules {

private:
    std::string diagnosis;




    std::queue        <std::pair        <std::string,        std::string>
>variables_initialized_queue;

    // constructor with diagnosis passed here
    // our default diagnosis is "Heart Failure" otherwise




    // initiate an object of ForwardChain to use knowledge base for forward
chain

    ForwardChain chain_forward = ForwardChain();




    //getting the index of the disease diagnosed to initiate our forward
chain




    // we got the index of the disease, now we can initiate the effects of
the disease
    // we can also initialize the conclusion variable queue
```

```
    // on our knowledge base diagnosis-ailment gives us the effects of the
disease
    // the ailment condition then initializes the variable initialized list
with the effect of the disease


    void initialize_ailment( int diagnosis_index) {

        for         (         auto         diagnosis_ailment_pair         :
chain_forward.diagnosis_ailment) {
            if ( diagnosis_index == diagnosis_ailment_pair.first){
                std::vector<int>              ailments_index              =
diagnosis_ailment_pair.second;


                // we are initializing the ailments or clause_variable based
on diagnosis


                for ( auto ailment_number : ailments_index){

chain_forward.variable_initialized_list[ailment_number].second           =
chain_forward.ailment_condition[diagnosis_index][ailment_number];



variables_initialized_queue.push(chain_forward.variable_initialized_list[a
ilment_number]);
                    }

            }
        }

        // this needs to call a function that goes thru the queue we just
enqueued
        apply_forward_chain(variables_initialized_queue);
    }


    // now that ailments/effects have been initialized, we can work our way
to the treatment as well
    // at this point all our effects of disease have been initialized, now
for the next part:

    void apply_forward_chain(std::queue< std::pair<std::string, std::string
> >variables_initialized_queue){


        while ( ! variables_initialized_queue.empty() ){
        std::pair<      std::string,      std::string>      variable      =
variables_initialized_queue.front();
        variables_initialized_queue.pop();

```

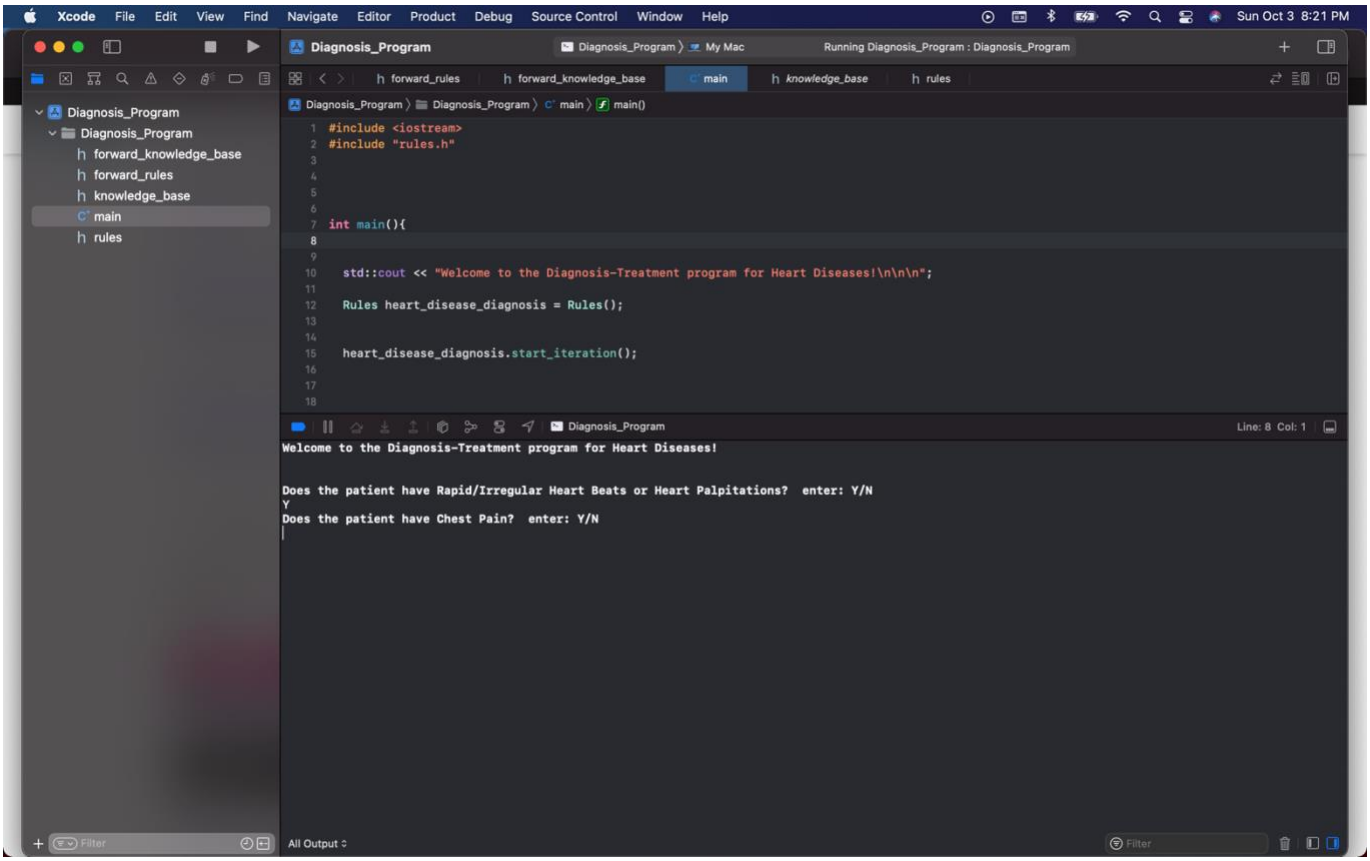
```
        //check the value of pair in the knowlege base
ailment_conditions_treatment to suggest a treatment

            for (auto ailment_condition_treat :
chain_forward.ailment_condition_treatment){
                std::vector<std::string> ailment_condition =
ailment_condition_treat.first;


                if ( variable.first == ailment_condition[0] ) {
                    //if (variable.second == ailment_condition[1]){  // &&
                    std::string output_print = "\n\nThe patient might have
" + variable.first;
                    std::string treatment_print = "\nTreat " +
variable.first + " with " + ailment_condition_treat.second;

                    std:: cout << output_print << std::endl;
                    std:: cout << treatment_print << std::endl;
                    }
                }
            }
        }


        //end_program();



    int end_program() {
        std::cout << "\n\n\n\nThank you for using this program!\n";
        exit(0);
    }




public:

    Forward_Rules( std::string diagnosis){
        this->diagnosis = diagnosis;
    };


    void initialize_forward_rule (){
        int diagnosis_index = 0;
        for ( int i = 0; i < chain_forward.diagnosis_list.size(); i++ ){
            if ( chain_forward.diagnosis_list[i] == this->diagnosis){
                diagnosis_index = i;
                break;
            }
        }

        initialize_ailment(diagnosis_index);
        end_program();
```

```
        }
    };
```

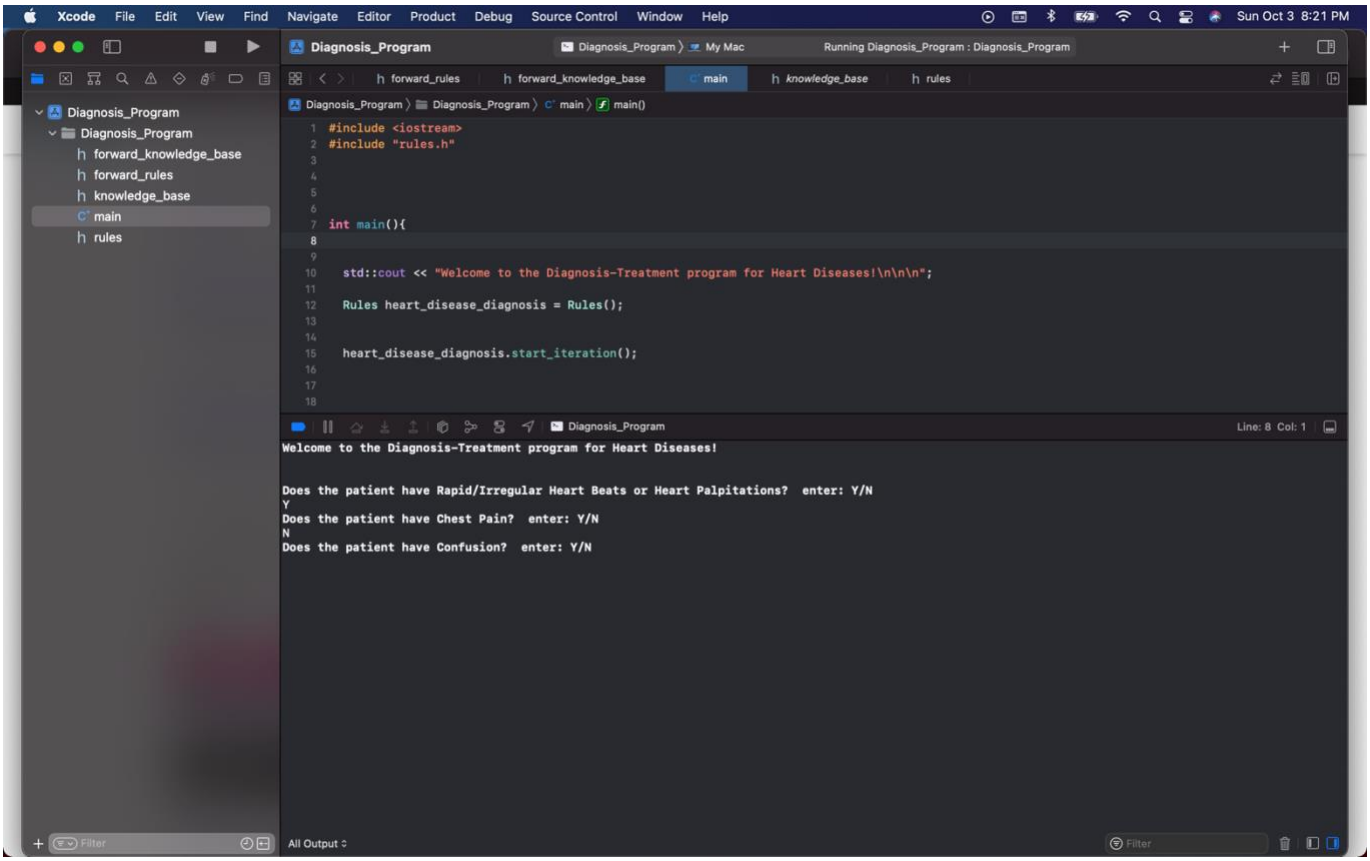
A Copy of the program Run

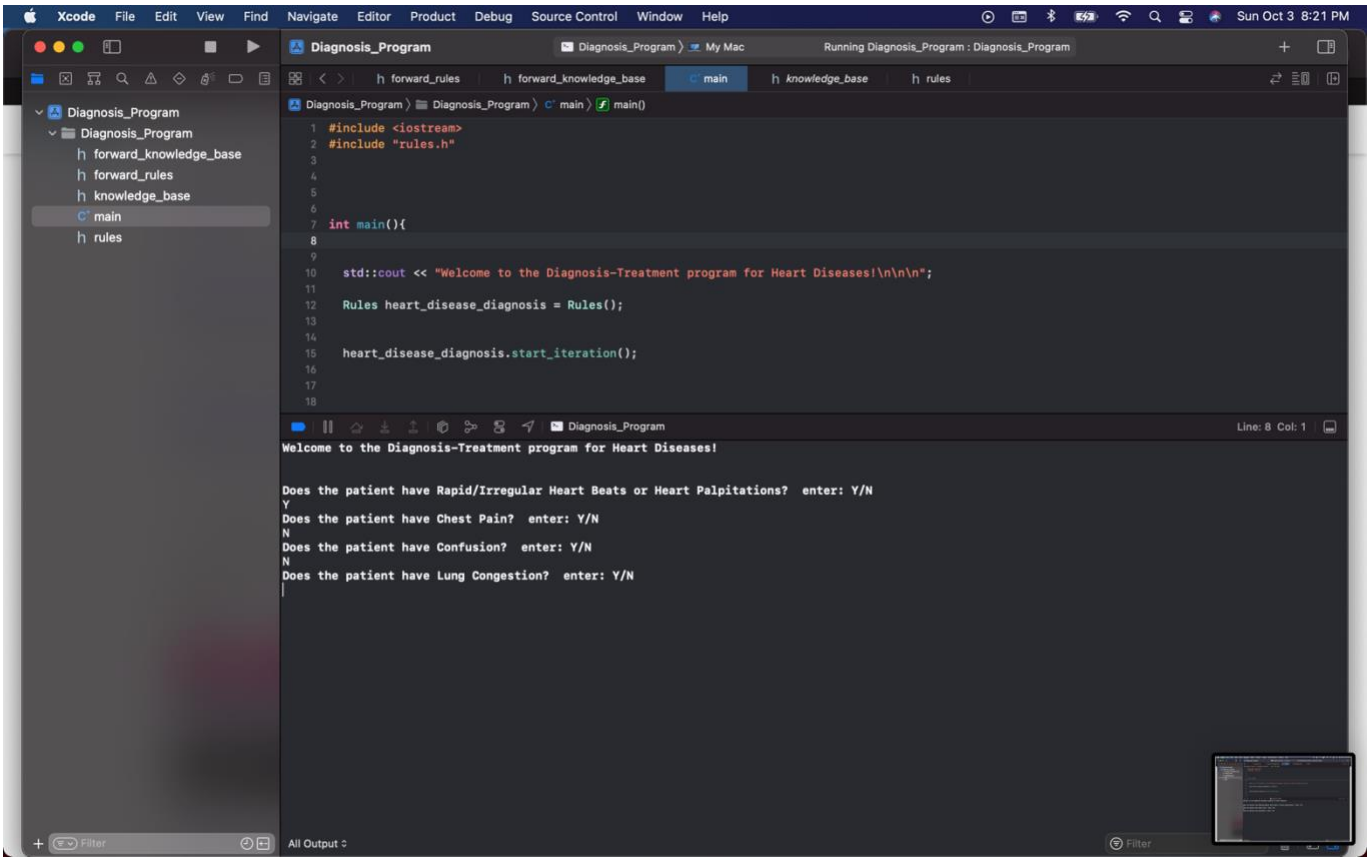


The program begins when an object of Rules class has been instantiated and a start_iteration() method called on that object. It begins by asking whether a particular symptom is being shown by the patient. The response of the user can only be yes or no and the prompt is repeated until the valid answer has been received.

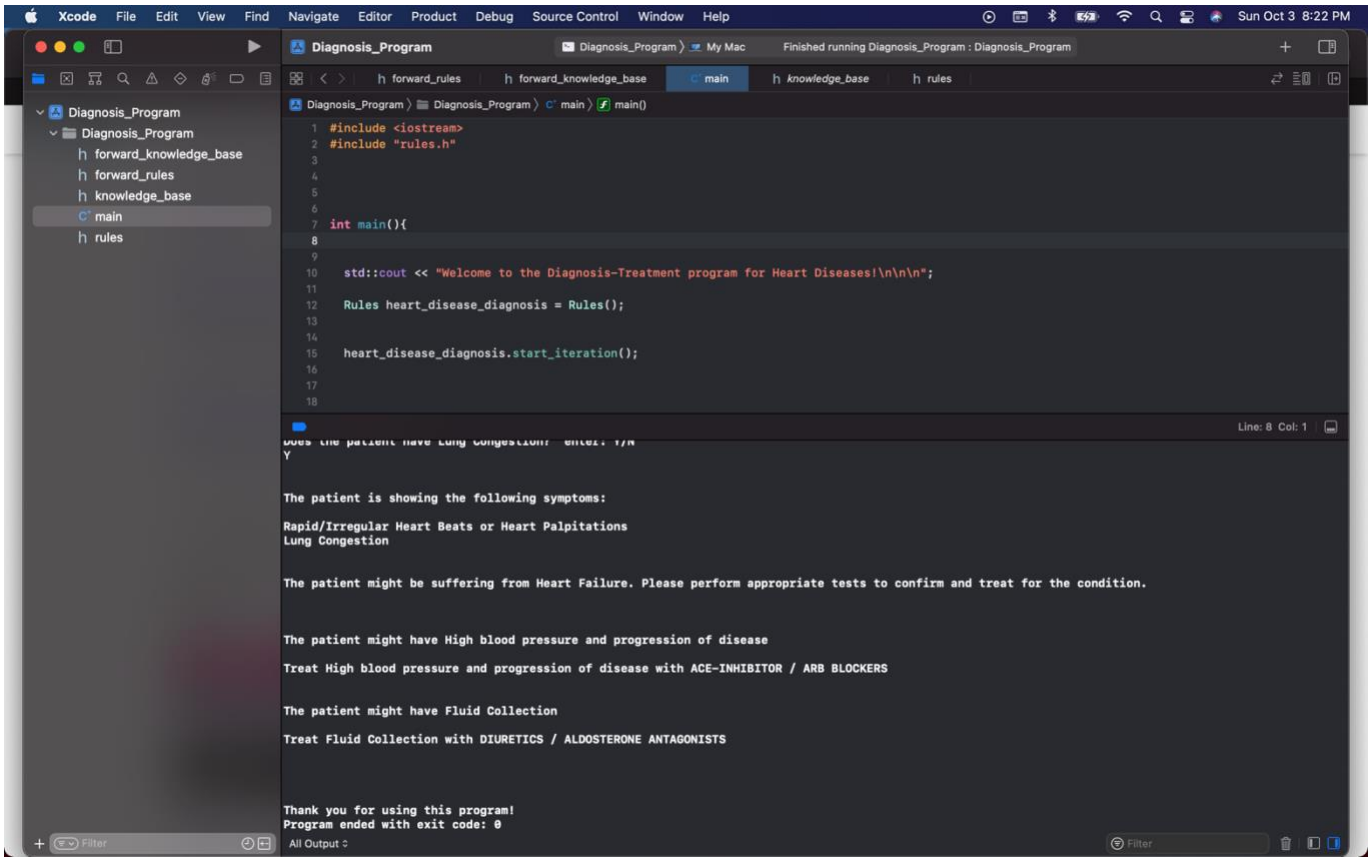


While enough variables have not been instantiated or a diagnosis cannot be reached based on the current rules in the knowledge base of the backward chain program, the user is repeatedly asked questions about further symptoms present in a patient.


Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help
Sun Oct 3 8:21 PM
Diagnosis_Program
Diagnosis_Program 〉 My Mac
Running Diagnosis_Program : Diagnosis_Program
forward_rules
forward_knowledge_base
main
knowledge_base
rules
Diagnosis_Program 〉 Diagnosis_Program 〉 main 〉 main()
Diagnosis_Program
Diagnosis_Program
forward_knowledge_base
forward_rules
knowledge_base
main
rules
#include <iostream>
#include "rules.h"
int main(){
std::cout << "Welcome to the Diagnosis-Treatment program for Heart Diseases!\n\n\n";
Rules heart_disease_diagnosis = Rules();
heart_disease_diagnosis.start_iteration();
Diagnosis_Program
Line: 8 Col: 1
Welcome to the Diagnosis-Treatment program for Heart Diseases!
Does the patient have Rapid/Irregular Heart Beats or Heart Palpitations? enter: Y/N
Y
Does the patient have Chest Pain? enter: Y/N
N
Does the patient have Confusion? enter: Y/N
All Output



While a diagnosis cannot be reached the user is repeatedly asked new questions that instantiate variables that have yet to be instantiated. As soon as enough variables have been instantiated for the program to satisfy all ‘if’ statements of a particular rule, the diagnosis, represented in the ‘then’ part of that rule is printed out by the program.



When the diagnosis is made, the program prints out the diagnosis in the screen. The backward chain program automatically calls the forward chain program as soon as the diagnosis is made. At this point, we do not need further user inputs. The forward chain program instantiates further variables based on the diagnosis passed to it which further instantiate possible course of treatment that have been coded as part of if-then structure in the knowledge base of forward chaining program.

Analysis of the program

There are a few differences in this program implementation in comparison to the methods we learned that constitute backward chaining process. In particular, when a rule has been discarded or deemed unable to be accomplished during the program run, backward chain program usually has the ability to recall what decision it is looking for and pick the next set of rules that have that particular decision as the 'then' part of the rules. But in our program implementation, we have generalized all decisions as 'diagnosis', so every instance of rule in our program gives a decision about diagnosis.

This would be different in other scenarios. A more complex program for diagnosing illnesses could have heart disease diagnosis as one possible decision among other disease diagnosis such as kidney disease or liver disease. In such event it would not be wise to sequentially go through each rule as we do in our program, the get_diagnosis() method in such program could check beforehand if the set of rules it is about to work on would be applicable to give diagnosis for the kind of diagnosis it is looking for. For example, if the backward chain has established it needs to diagnose a heart disease based on prior instantiation of variables, it should avoid such rules that seek to provide decision about kidney disease or liver disease and only add such rules in the stack that lead to conclusion about the heart disease.

We could easily check for such condition in our program but it seems unnecessary in the current scope of our program and we have avoided it.

Our data structures are small, our rules were few and conclusions were only a tiny subset of all possible permutation of conclusions and symptoms. As such, we

primarily used vectors as our data structures wherever possible. As the program becomes larger and datasets become several hundred or thousands size big, we could use more efficient data structures that allow for fast lookup time. Unordered maps in C++ use hashing techniques and provide O(1) lookup time and seem more suitable data structure if we were to scale up our program to be production ready.

Analysis of the results

Since our program is a relatively small project and almost implemented as a proof of concept rather than a production ready project, the data structures we have implemented and the features we have decided to omit in our program do not greatly affect the program implementation. The modifications do not present a challenge for us to prove that the program would run as expected if strict guidelines about the forward and backward chaining principles were followed. There are cases where the diagnosis is not possible (if the user answers all prompts with 'No' as an answer) and our program handles that by suggesting the user consult an M.D. for proper diagnosis of the disease.

Since we ended up writing the code for the program from scratch and not making use of the provided C code as a starting guide for the program, we tried as much as possible to implement the same logic and exact data structures that are the driving factors of backward chaining and forward chaining principles. We implemented stack data structure for backward chaining and queue data structure for forward chaining. Backward chaining initialized variables based on user input whereas forward chain followed a domino effect where once the diagnosis had been

provided it led to the possible effects of that diagnosis on the patient and recommended course of treatment to treat and cure those effects.

Conclusion

In conclusion the project provided us with a hand-on opportunity to understand and implement the forward-chaining and backward-chaining principles in an expert-system. It allowed us to question and understand the logic and the necessity behind the use of specific algorithm and data structures in implementing forward chaining versus implementing backward chaining. I have a better understanding of why forward chaining requires queue data structure and why backward chaining program requires the implementation of stack data structures.

It was also important opportunity for us as a team to practice working in groups after not being able to get involved in any group activities in the previous few semesters.

References

1. https://www.mayoclinic.org , Mayo Foundation for Medical Education and Research (MFMER)
2. https://www.webmd.com, WebMD LLC

Contributions

One of our teammates Robert Balthrop dropped the course and this project was completed by me and Jake Stuart. Jake provided important contributions during the research phase of the project taking initiatives to develop the knowledge base to implement the program. We both worked collaboratively to pick these particular diseases we have included in our knowledge base, looking up their symptoms, effects and course of treatment. Jake devised the heatmap included earlier in the methodology about coming up with requisite 30+ rules for backward chaining program. The heatmap was vital idea that provided a great visual aid for us to come up with unique 'if-then' rules that led to specific diagnosis. This effectively helped our rules avoid being redundant, circular, or incomplete.

Both of us worked together to come up with decision trees that we included in this report.

Jake provided me with constant feedback during the coding phase of the program which I took charge of. He was involved in going through the codes, testing various edge cases to make sure we eliminated as many bugs and errors as we could. I designed the data structures, the driving logic, and algorithm for forward chaining and backward chaining. Jake and I are also looking forward to developing a GUI for this project but that would be beyond the scope of this project at this moment.