Project1 Report

# The Problem

The problem assigned was to create a program that gets information from a user on the symptoms a person is experiencing and then diagnose said person based on those symptoms. The program was to use backward chaining and forward chaining. The backward chaining for collecting symptom data from the user and make a diagnosis, and the forward chaining to provide the user with suggested treatments based on the diagnosis.

# The Domain

The domain is a medical/hospital setting wherein a patient is expected to have been exposed to some sort of poison or toxic substance. The medical professional caring for the patient would answer questions regarding the symptoms of the patient and then would be presented with a diagnosis and suggested treatments to administer to the patient.

# Methodologies

## Backward Chaining Description

Backward chaining is an inference engine used in expert system artificial intelligence development. In backward chaining the goal or conclusion is known and the program must work to determine if the requirements are met for the goal/conclusion to be achieved.

In this program, backward chaining is used in the diagnosing of the patient. The program runs through a list of possible diagnoses and the user is asked to answer questions about the symptoms associated with the diagnosis. These answers are then compared to the values required for the diagnosis to be true. If all the values match a diagnosis is made, if not the next possible diagnosis is checked.

## Forward Chaining Description

Forward chaining is an inference engine used in expert system artificial intelligence development. As opposed to backward chaining, in forward chaining the goal or conclusion is unknown and the facts are known. The facts are evaluated and a goal/conclusion is determined based on them.

In this program, forward chaining is used for determining the list of treatments to be suggested to the user. The facts, that is the diagnosis made, are compared to a list of all possible diagnoses. If the diagnoses match, the associated conclusion, the list of treatments, is printed to the screen. If they do not match the next diagnosis is checked.

# The Program

## Decision Trees

### Backward Chaining Tree:

Diagram

Description automatically generated

A couple of notes about the “indicative symptom” questions:

The indicative symptom questions are for when, (considering the users previous answers) at that point, a particular symptom could only indicate a particular diagnosis. Note that any symptom listed as an indicative symptom is not seen any further down the flow chart.

Also, if the user answers yes to the initial question (is the user experiencing any of the following…?) and then does not select a symptom from the list, it is treated as if the user answered no to the initial question.

### Forward Chaining Tree:

A picture containing text, receipt, document

Description automatically generated

## Rules

### Backward Chaining Rules:

1. IF indicQuestion1 = TRUE

THEN indicSymp1 = TRUE

1. IF indicSymp1 = TRUE AND urinary retention = TRUE

THEN diagnosis = Antifreeze

1. IF indicSymp1 = TRUE AND paralysis = TRUE

THEN diagnosis = Food

1. IF indicSymp1 = TRUE AND red or swollen skin = TRUE

THEN diagnosis = Arsenic

1. IF indicSymp1 = TRUE AND warts or lesions = TRUE

THEN diagnosis = Arsenic

1. IF indicSymp1 = TRUE AND burning eyes, nose, or mouth = TRUE

THEN diagnosis = Formaldehyde

1. IF vomiting or nausea = FALSE AND chest pains = TRUE

THEN diagnosis = Asbestos

1. IF vomiting or nausea = FALSE AND chest pains = FALSE AND headaches = TRUE

THEN diagnosis = Mercury

1. IF vomiting or nausea = FALSE AND chest pains = FALSE AND fatigue = TRUE

THEN diagnosis = Mercury

1. IF vomiting or nausea = FALSE AND chest pains = FALSE AND convulsions = TRUE

THEN diagnosis = Mercury

1. IF vomiting or nausea = FALSE AND chest pains = FALSE AND insomnia = TRUE

THEN diagnosis = Mercury

1. IF vomiting or nausea = FALSE AND chest pains = FALSE AND headaches = FALSE AND fatigue =

FALSE AND convulsions = FALSE AND insomnia = FALSE

THEN diagnosis = Unknown

1. IF indicQuestion2 = TRUE

THEN indicSymp2 = TRUE

1. IF indicSymp2 = TRUE AND convulsions = TRUE

THEN diagnosis = Antifreeze

1. IF indicSymp2 = TRUE AND fever = TRUE

THEN diagnosis = Food

1. IF indicSymp2 = TRUE AND irritability = TRUE

THEN diagnosis = Lead

1. IF indicSymp2 = TRUE AND dehydration = TRUE

THEN diagnosis = Alcohol

1. IF irregular breathing = TRUE AND chest pains = TRUE

THEN diagnosis = Formaldehyde

1. IF irregular breathing = TRUE AND chest pains = FALSE AND seizures = TRUE

THEN diagnosis = Alcohol

1. IF irregular breathing = TRUE AND chest pains = FALSE AND confusion = TRUE

THEN diagnosis = Carbon-Monoxide

1. IF irregular breathing = TRUE AND chest pains = FALSE AND arrythmia = TRUE

THEN diagnosis = Antifreeze

1. IF irregular breathing = TRUE AND chest pains = FALSE AND seizures = FALSE AND

confusion = FALSE AND arrythmia = FALSE

THEN diagnosis = Unknown

1. IF indicQuestion3 = TRUE

THEN indicSymp3 = TRUE

1. IF indicSymp3 = TRUE AND confusion = TRUE

THEN diagnosis = Tylenol

1. IF indicSymp3 = TRUE AND seizures = TRUE

THEN diagnosis = Lead

1. IF indicSymp3 = TRUE and headaches = TRUE

THEN diagnosis = Food

1. IF indicSymp3 = TRUE and fatigue = TRUE

THEN diagnosis = Food

1. IF indicSymp3 = TRUE and arrythmia = TRUE

THEN diagnosis = Arsenic

1. IF tingling or numbness = TRUE AND muscle pains = TRUE

THEN diagnosis = Arsenic

1. IF tingling or numbness = TRUE AND muscle pains = FALSE

THEN diagnosis = “Food Poisoning”

1. IF tingling or numbness = FALSE

THEN diagnosis = “Tylenol Poisoning”

### Forward Chaining Rules:

1. IF diagnosis = Antifreeze

THEN treatmentFile = antifreezeTreatment.txt

1. IF diagnosis = Carbon-Monoxide

THEN treatmentFile = cOTreatment.txt

1. IF diagnosis = Food

THEN treatmentFile = foodPoisoningTreatment.txt

1. IF diagnosis = Arsenic

THEN treatmentFile = arsenicTreatment.txt

1. IF diagnosis = Asbestos

THEN treatmentFile = asbestosisTreatment.txt

1. IF diagnosis = Formaldehyde

THEN treatmentFile = formaldehydeTreatment.txt

1. IF diagnosis = Lead

THEN treatmentFile = leadTreatment.txt

1. IF diagnosis = Mercury

THEN treatmentFile = mercuryTreatment.txt

1. IF diagnosis = Alcohol

THEN treatmentFile = alcoholTreatment.txt

1. IF diagnosis = Tylenol

THEN treatmentFile = tylenolTreatment.txt

1. IF diagnosis = Unknown

THEN treatmentFile = unkownTreatment.txt

## Source Code Implementation

The program starts off in main() by creating goalList, a vector that contains all the goal/conclusion names, their initialization state and their value. The goalList is created in main() because it is used in and passed to most other functions. In a do-loop goalList is cleared, so that the program can be restarted with a clean slate. The initial values of goalList are read into it from the goalVariList.txt. Then identify\_poison(), the backward chaining process, is called, followed by treatment(), the forward chaining process. The console screen is cleared of the questions asked of the user and then the .txt file found by treatment() is read to the console. This .txt file contains both the diagnosis and list of suggested treatments. Then the user is asked to enter “r” or “q” to restart or quit the program, respectively. The console is cleared again, if “q” was entered the program exits normally, if “r” the loop is restarted.

The function identify\_process() starts off the backward chaining process to make a diagnosis. All the .txt files in the backwardChaining folder are read into vectors to be used in the backward chaining process. Vector ifStates holds the symptom-value requirements for each rule, and thenStates holds the respective goal-value assignments to be made if the requirements are met for a particular rule. Then the conList vector contains a list of the goal/conclusions names, but not their values, of each rule in order of rule number. The variList vector contains, in no particular order, the variable names, their initialization state and their associated values. Finally, the clauseList vector holds variable names, but not the values, of each symptom in ifStates of a particular rule, in the order they appear in ifStates. The function finally calls processBC() with “diagnosis” as the desired goal.

processBC()’s only calls search\_con(), but it serves as the beginning of a loop which may be called up later in the program when variable’s value is dependent on a goal/conclusion’s value. In which case processBC() is called with the needed goal/conclusion name as the desired goal.

search\_con() first finds the location of the desired goal in the goalList using a while-loop. Then a second while-loop is used to find each rule number with a conclusion in conList that matches the desired goal. The loop is continued until the desired goal is initialized or the end of conList is reached. For any rule with a goal/conclusion in conList matching the desired goal, rule\_to\_clause() is called.

rule\_to\_clause() converts the rule number to the index of the first variable name in clauseList associated with it. It then calls upate\_VLBC().

update\_VLBC() uses a for-loop to move through the list of variable names in clauseList associated with the current rule number and sets and/or checks their value. Each rule has its own list with a max of 6 possible variable names, if a rule uses less variables then the other spaces in the list are filled with “ “. Two boolean variables are created called, found and match. Variable found indicates whether the variable name has been found in a certain list, match indicates if the currently initialized variable names in the clauseList match with the expected values shown in ifStates for the given rule. If the end of the clauseList for that rule is reached, by going through all 6 variable names or reaching a “ “, or if match is false then the for-loop is ended. After, if match is still true, validate\_Ri() is called to assign the given value in thenStates to goal in goalList. Inside the for-loop the variable is searched for with two inner for-loops. The first for-loop iterates through variList, if it is found and it is initialized validate\_Ri() is called to check if it’s value matches the expected value in ifStates, match is changed accordingly and the inner-loop is exited. If the variable is found in variList but is not initialized symptomQuestions() is called to ask the user to give a “y” or “n” answer to if the symptom being represented by the variable is being experienced. The variable is then set to initialized and is updated with the “y”/”n” value given by the user, validate\_Ri() is called to check if it’s value matches the expected value in ifStates, match is changed accordingly and the inner-loop is exited. If the variable is not found in variList it is then searched for in the second for-loop that iterates through goalList. If the variable is found in goalList and is initialized, validate\_Ri() is called on it, match is updated, the inner-loop is exited. If it is not initialized, processBC() with the variable as the new desired goal will be called to initialize it. When processBC() returns validate\_Ri() is called on the newly initialized variable, match is updated and the inner-loop is exited.

If validate\_Ri() is being called from inside the for-loop in update\_VLBC(), it finds the index of the ifStates the passed variable is being compared to, then returns true if the variable’s value in variList or goalList matches the expected value in ifStates, if it does not false is returned. if validate\_Ri() is being called from outside the for-loop, indicating that all variable values match all ifStates values, then the goal in goalList is updated with the value given by the corresponding thenStates rule, the goal is set to initialized and the function returns true.

Once the goal, “diagnosis”, is assigned a value identify\_poison() returns and treatment() is called in main(). treatment() has it’s own set of ifStates, thenStates, clauseList, and variList vectors to be read in from the .txt files in the forwardChaining folder and initialized to. Once that is done processFC() is called with “diagnosis” as the fact being searched for.

process\_FC() only calls search\_cvl(), but like process\_BC() works as the beginning of a loop that can be called later in the program.

search\_cvl() searches through the clauseList for variable names that match the fact variable name. If a match is found clause\_to\_rule() is called, which calculates and returns the index of the rule that the clause belongs to, then update\_VLFC() is called.

update\_VLFC() works nearly the same as update\_VLBC() with a few minor changes. First, the max size of the list of variables belonging to a rule in clauseList is only 1 since “diagnosis” is the only fact to be checked due to the program’s current capabilities. Second, goalList is checked before variList because “diagnosis” is only found in goalList. Fnally, if the fact is in goalList but is not initialized processFC() is called instead of processBC().

Seemingly unnecessary things like, checking variList if “diagnosis” is always in goalList, and calling processFC() if “diagnosis” must already be initialized for treatment() to run, were left in with the expectation that the program might be expanded on in the future. Allowing for more personalized treatment plans to be produced depending on specific factors. For example, a treatment plan might not include the administration of morphine for pain if the patient regularly uses opioids, since opioid use builds up a resistance to the effects of morphine.

Also validate\_Ri() and symptomQuestions() are used in both update\_VLBC() and update\_VLFC() as their functions do not change between the backward and forward chaining processes.

## Source Code

#include <stdlib.h>

#include <fstream>

#include <iostream>

#include <sstream>

#include <string>

#include <vector>

using namespace std;

void identify\_poison(vector<vector<string>>&);

void processBC(string, vector<vector<string>>&, vector<vector<vector<string>>>,

vector<vector<vector<string>>>, vector<string>, vector<vector<string>>&, vector<string>);

void search\_con(string, vector<vector<string>>&, vector<vector<vector<string>>>, vector<vector<vector<string>>>, vector<string>, vector<vector<string>>&, vector<string>);

void rule\_to\_clause(int, string, vector<vector<string>>&,

vector<vector<vector<string>>>,

vector<vector<vector<string>>>, vector<string>,

vector<vector<string>>&, vector<string>);

void update\_VLBC(int, int, string, vector<vector<string>>&,

vector<vector<vector<string>>>, vector<vector<vector<string>>>,

vector<string>, vector<vector<string>>&, vector<string>);

bool validate\_Ri(int, int, int, string, string, string,

vector<vector<vector<string>>>, vector<vector<vector<string>>>, vector<vector<string>>, vector<vector<string>>&);

string symptomQuestions(string);

void treatment(vector<vector<string>>&);

void processFC(string, vector<vector<string>>&, vector<vector<vector<string>>>,

vector<vector<vector<string>>>, vector<vector<string>>&,

vector<string>);

int search\_cvl(int, string,  vector<vector<string>>&,

vector<vector<vector<string>>>, vector<vector<vector<string>>>,

vector<vector<string>>&, vector<string>);

bool update\_VLFC(int, int, string, vector<vector<string>>&,

vector<vector<vector<string>>>, vector<vector<vector<string>>>, vector<vector<string>>&, vector<string>);

int clause\_to\_rule(int, string,  vector<vector<string>>&,

vector<vector<vector<string>>>, vector<vector<vector<string>>>, vector<vector<string>>&, vector<string>);

//create goalList containing intializations state and values of the goals, calls the backward and forward chaining functions, then prints the final results.

int main(){

    vector<vector<string>> goalList;    //holds the values names, initialization

state, and values of the

goal/conclusion variables.

    vector<string> tokens;              //the following variables are needed to

read .txt data into vectors

    stringstream ss;

    string token;

    string str;

    ifstream inFile;

    string input = "r";                //user input, accepts values: "r" =

restart program and "q" = quit program

    do{

    goalList.clear();                  //clears names, initialization states, and

values of the goal/conclusion variables

for clean restart

    inFile.open("data/goalVariList.txt");   //reads .txt data into vector

    if(!inFile){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (goalList.txt) failed to open"<< endl;

            return 1;

    }

    else{

        while(getline(inFile, str)){

            ss.clear();

            ss.str(str);

            while(getline(ss, token, ' ')){

                tokens.push\_back(token);

            }

            goalList.push\_back(tokens);

            tokens.clear();

        }

    }

    inFile.close();

    string input;

    identify\_poison(goalList);       //Start backward chaining to find diagnosis

    treatment(goalList);             //Start forward chaining to find treatment

    system("cls");                   //clear questions asked in identify\_poison()

from screen

    string path = "data/treatmentPlans/" + goalList[1][2];        //Prints the

result of

treatment()

    inFile.open(path);

    if(!inFile){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (treatment.txt) failed to open"<<endl;

            return 1;

    }

    else{

        while(getline(inFile,str)){

            cout << str << endl;

        }

    }

    inFile.close();

    while((input != "r") && (input != "q")){        //validates user input to

restart or quit the program

        cout << "Enter 'r' to restart the program or 'q' to exit." << endl;

        cin >> input;

    }

    system("cls");              //clear the result of treatment() from the screen

    if(input == "q"){           //exit program if input = "q"

        return 0;

    }

    }while(input == "r");       //restart program if input = "r"

}

                         //BACKWARD CHAINING METHODS

//Initializes all required data structures (using info from related files) to find a diagnosis, calls processBC() to begin the search.

void identify\_poison(vector<vector<string>>& goalList){

    vector<string> tokens;                 //the following variables are needed

to read .txt data into vectors

    vector<vector<string>> statement;

    stringstream ss;

    string str;

    string token;

    ifstream inFile;

    vector<vector<vector<string>>> ifStates;  //Create ifStates containing

symptoms required values for

diagnosis to be made.

    inFile.open("data/backwardChaining/ifStatements\_BC.txt");

    if(!inFile){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (ifStatements\_BC.txt) failed to open"<< endl;

            return;

    }

    else{

        while(getline(inFile, str)){

            while(!str.empty()){

                for(int i = 0; i < 2; i++){

                    tokens.push\_back(str);

                    getline(inFile, str);

                }

                statement.push\_back(tokens);

                tokens.clear();

            }

            ifStates.push\_back(statement);

            statement.clear();

        }

    }

    inFile.close();

    vector<vector<vector<string>>> thenStates;  //Create thenStates containing

conclusions made if

if-statements are met.

    inFile.open("data/backwardChaining/thenStatements\_BC.txt");

    if(!inFile){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (thenStatements\_BC.txt) failed to open"<< endl;

            return;

    }

    else{

        while(getline(inFile, str)){

            ss.clear();

            ss.str(str);

            if(!str.empty()){

                while(getline(ss, token, ' ')){

                    tokens.push\_back(token);

                }

                statement.push\_back(tokens);

                tokens.clear();

            }

            else{

                thenStates.push\_back(statement);

                statement.clear();

            }

        }

    }

    thenStates.push\_back(statement);

    statement.clear();

    inFile.close();

    vector<string> conList;       //Create conList containg the conclusions of

each then-statement in each rule.

    inFile.open("data/backwardChaining/conclusionList\_BC.txt");

    if(!inFile){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (conclusionList.txt) failed to open"<< endl;

            return;

    }

    else{

        while(getline(inFile, str)){

            conList.push\_back(str);

        }

    }

    inFile.close();

    vector<vector<string>> variList;  //Creates variList contianing the

initialization states and values of the

variables.

    inFile.open("data/backwardChaining/variableList\_BC.txt");

    if(!inFile){

        cout << endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (variableList\_BC.txt) failed to open"<< endl;

            return;

    }

    else{

        while(getline(inFile, str)){

            tokens.push\_back(str);

            getline(inFile, str);

            ss.clear();

            ss.str(str);

            while(getline(ss, token, ' ')){

                tokens.push\_back(token);

            }

            variList.push\_back(tokens);

            tokens.clear();

        }

    }

    inFile.close();

    vector<string> clauseList;   //Creates clauseList containg the variables of

each if-statements in each rule.

    inFile.open("data/backwardChaining/clauseList\_BC.txt");

    if(!inFile){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (clauseList\_BC.txt) failed to open"<< endl;

            return;

    }

    else{

        while(getline(inFile, str)){

            clauseList.push\_back(str);

        }

    }

    inFile.close();

    processBC("diagnosis", goalList, ifStates, thenStates, conList, variList,

clauseList);      //Call process to begin searching for

diagnosis.

}

//Begins backward chaining process

void processBC(string goal, vector<vector<string>>& goalList,

vector<vector<vector<string>>> ifStates,

vector<vector<vector<string>>> thenStates,

vector<string> conList, vector<vector<string>>& variList,

vector<string> clauseList){

    search\_con(goal, goalList, ifStates, thenStates, conList, variList,

clauseList);

}

//Searches for the first/next rule number with the desired goal in the conclusion.

void search\_con(string goal, vector<vector<string>>& goalList,

vector<vector<vector<string>>> ifStates,

vector<vector<vector<string>>> thenStates,

vector<string> conList, vector<vector<string>>& variList,

vector<string> clauseList){

    int i = 0;

    int Ri = 0;

    while(goalList[i][0] != goal){    //find the goal in goalList

        i++;

    }

    while(goalList[i][1] == "n" && Ri < conList.size()){  //while goal is not

initialized and not

all rules have been

checked

        if(conList[Ri] == goal){          //if goal is the conclusion of rule Ri

            rule\_to\_clause(Ri, goal, goalList, ifStates, thenStates, conList,

variList, clauseList);

        }

        Ri++;

    }

    if(Ri > conList.size()){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            << "End of conclusion list reached without initializing" << goalList[i][0] << endl;

        return;

    }

}

//Finds the intiail clause number for the rule number

void rule\_to\_clause(int Ri, string goal, vector<vector<string>>& goalList,

vector<vector<vector<string>>> ifStates,

vector<vector<vector<string>>> thenStates,

vector<string> conList, vector<vector<string>>& variList,

vector<string> clauseList){

    int Ci = 6\*Ri;                  //Calculate the first clause of rule Ri

    //the given formula was not used because the

Ri and Ci are given by the index in the

vectors.

    //As a result for Ri = 0, Ci = 6\*((Ri)-1) +1 would result in Ci = -5

    //to adjust for this the formula would have to be altered to Ci-1 = ((Ri+1)-

1)+1 to get the

    //the correct clause number, which is equal to Ci = 6\*Ri

    update\_VLBC(Ci, Ri, goal, goalList, ifStates, thenStates, conList, variList,

clauseList);

}

//Updates the variables in the variable list that are relevant to Rule Ri.

void update\_VLBC(int Ci, int Ri, string goal, vector<vector<string>>& goalList,

vector<vector<vector<string>>> ifStates,

vector<vector<vector<string>>> thenStates,

vector<string> conList, vector<vector<string>>& variList,

vector<string> clauseList){

    string vari;                        //variable name of clause Ci

    int j = 0;

    bool found = false;      //indicates if vari is found in variList or goalList

    bool match = true;       //indicates if the value of vari is matching with

what is required of rule Ri

    int CiEnd = Ci + 6;                 //the end of the clause list for rule Ri

    for(Ci; Ci < CiEnd; Ci++){          //while the end of the clause list has

not been reached

        vari = clauseList[Ci];          //read the first/next variable into vari

        found = false;                  //reinitialize found to false for 'next'

variable

        if( (vari == " ") || (match == false) ){   //if there are no more

variables or the values of

the variables do not match

with rule Ri

            Ci = CiEnd;                            //exit the loop

        }

        else{

            for(int i = 0; i < variList.size(); i++){ //iterate through

variList

                if(vari == variList[i][0]){                   //if vari is found

in variList

                    found = true;                             //set found to true

                    if(variList[i][1] == "y"){           //if vari is initialized

                        match = validate\_Ri(Ri, i, Ci, "i", "v", vari, ifStates,

thenStates, variList, goalList); //check if the

value of vari

matches with

rule Ri

                        i = variList.size();                    //exit the loop

                    }

                    else{                           //if vari is not initialized

                        string value = symptomQuestions(vari);    //ask user for

value of

vari

                        variList[i][1] = "y";         //set vari to initialized

                        variList[i][2] = value;         //set value of vari

                        match = validate\_Ri(Ri, i, Ci, "i", "v", vari, ifStates,

thenStates, variList, goalList);

//check if the value of vari matches

with rule Ri

                        i = variList.size();                      //exit the loop

                    }

                }

            }

            if(found == false){               //if vari is not found in variList

                for(int i = 0; i < goalList.size(); i++){ //iterate through

goalList

                    if(vari == goalList[i][0]){    //if vari is found in goalList

                        found == true;                      //set found to true

                        if(goalList[i][1] == "y"){      //if vari is initialized

                            match = validate\_Ri(Ri, i, Ci, "i", "g", vari,

ifStates, thenStates, variList,

goalList); //check if the value

of vari matches with

rule Ri

                            i = goalList.size();                  //exit the loop

                        }

                        else{                        //if vari is not initialized

                            processBC(vari, goalList, ifStates, thenStates,

conList, variList, clauseList); //restart process

with vari as the

goal

                            match = validate\_Ri(Ri, i, Ci, "i", "g", vari,

ifStates, thenStates, variList, goalList);

//check if the value of vari matches with rule Ri

                            i = goalList.size();                  //exit the loop

                        }

                    }

                }

            }

        }

    }

    int i = 0;

    while (goal != goalList[i][0]){      //find the index of the goal in goalList

        i++;

    }

    if(match == true){              //if the vari values match with the rule Ri

        match = validate\_Ri(Ri, i, j, "t", "g", goal, ifStates, thenStates,

variList, goalList); //set the value of the goal

        return;

    }

    else{               //if the vari values did not match with the rule Ri

        return;

    }

}

//Compares the values of the variables in the variable list to the required values in the if-statements forn the then-statement to be true, then updates the goalList with the correct value.

//NOTE: Works for both backward and forward chaining!

bool validate\_Ri(int Ri, int i, int j, string ifthen, string goalvari,

string vari, vector<vector<vector<string>>> ifStates,

vector<vector<vector<string>>> thenStates,

vector<vector<string>> variList,

vector<vector<string>>& goalList){

    if(ifthen == "i"){                    //if an if statement is being validated

        j = 0;

        while(vari != ifStates[Ri][j][0]){      //find the index of the if

statement corresponding to vari

            j++;

        }

        if(goalvari == "v"){                        //if vari is in the variList

            if(variList[i][2] == ifStates[Ri][j][1]){    //if the value of vari

and the if statement

match return true

                return true;

            }

            else{ //if the value of vari and the if statement

do not match return false

                return false;

            }

        }

        if(goalvari == "g"){                        //if vari is in the goalList

            if(goalList[i][2] == ifStates[Ri][j][1]){    //if the value of vari

and the if statement

match return true

                return true;

            }

            else{                   //if the value of vari and the if statement

do not match return false

                return false;

            }

        }

    }

    if(ifthen == "t"){                            //if a then statement validated

        goalList[i][2] = thenStates[Ri][0][1];    //set the value of vari to the

conclusion value of rule Ri

        goalList[i][1] = "y";                     //set vari to initalized

        return true;

    }

}

//Asks the user to provide a value for a variable

//NOTE: Works for both backward and forward chaining!

string symptomQuestions(string symptom){

    string input;

    cout << "Is the patient experiencing ";

    if(symptom == "indicQuestion1"){

        cout << "urinary retention, paralysis, red/swollen skin, " << endl

             << "warts/lesions, swollen digits, or burning eyes, nose, or

mouth?"<< endl;

    }

    else if(symptom == "indicQuestion2"){

        cout << "convulsions, fever, irritability, or dehydration?" << endl;

    }

    else if(symptom == "indicQuestion3"){

        cout << "confusion, seizures, headaches, fatigue or arrhythmia?" << endl;

    }

    else{

        cout << symptom << "?" << endl;

    }

    do{

    cout << "Enter 'y' for yes or 'n' for no." << endl;

    cin >> input;

    }while(input != "y" && input != "n");

    return input;

}

                          //FORWARD CHAINING METHODS

//Initializes all required data structures (using info from related files) to find a diagnosis, calls processFC() to begin the search.

void treatment(vector<vector<string>>& goalList){

    vector<string> tokens;   //the following variables are needed to read .txt

data into vectors

    vector<vector<string>> statement;

    stringstream ss;

    string str;

    string token;

    ifstream inFile;

    vector<vector<vector<string>>> ifStates;   //Create ifStates containing

symptoms required values for

diagnosis to be made.

    inFile.open("data/forwardChaining/ifStatements\_FC.txt");

    if(!inFile){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (ifStatements.txt) failed to open"<< endl;

            return;

    }

    else{

        while(getline(inFile, str)){

             ss.clear();

            ss.str(str);

            if(!str.empty()){

                while(getline(ss, token, ' ')){

                    tokens.push\_back(token);

                }

                statement.push\_back(tokens);

                tokens.clear();

            }

            else{

                ifStates.push\_back(statement);

                statement.clear();

            }

        }

    }

    ifStates.push\_back(statement);

    statement.clear();

    inFile.close();

    vector<vector<vector<string>>> thenStates;    //Create thenStates containing

conclusions made if if-

statements are met.

    inFile.open("data/forwardChaining/thenStatements\_FC.txt");

    if(!inFile){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (thenStatements\_FC.txt) failed to open"<< endl;

            return;

    }

    else{

        while(getline(inFile, str)){

             ss.clear();

            ss.str(str);

            if(!str.empty()){

                while(getline(ss, token, ' ')){

                    tokens.push\_back(token);

                }

                statement.push\_back(tokens);

                tokens.clear();

            }

            else{

                thenStates.push\_back(statement);

                statement.clear();

            }

        }

    }

    thenStates.push\_back(statement);

    statement.clear();

    inFile.close();

    vector<string> clauseList;         //Create clauseList containg the variables

of each if-statements in each rule.

    inFile.open("data/forwardChaining/clauseList\_FC.txt");

    if(!inFile){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (clauseList\_FC.txt) failed to open"<< endl;

            return;

    }

    else{

        while(getline(inFile, str)){

            clauseList.push\_back(str);

        }

    }

    inFile.close();

    vector<vector<string>> variList;    //Creates variList contiaing the

initialization states and values of the

variables.

    inFile.open("data/forwardChaining/variableList\_FC.txt");

    if(!inFile){

        cout<<endl

            <<"\*PROGRAM TERMINATED\*"<<endl

            <<"inFile (variableList\_FC.txt) failed to open"<< endl;

            return;

    }

    else{

        while(getline(inFile, str)){

            ss.clear();

            ss.str(str);

            while(getline(ss, token, ' ')){

                tokens.push\_back(token);

            }

            variList.push\_back(tokens);

            tokens.clear();

        }

    }

    inFile.close();

    processFC("diagnosis", goalList, ifStates, thenStates, variList, clauseList);               //Call process to begin searching for diagnosis.

}

//Controls the flow of control

void processFC(string fact, vector<vector<string>>& goalList,

vector<vector<vector<string>>> ifStates,

vector<vector<vector<string>>> thenStates,

vector<vector<string>>& variList, vector<string> clauseList){

    search\_cvl(0, fact, goalList, ifStates, thenStates, variList, clauseList);

}

//Iterates through the variables of the clause variable list are returns a Ci

int search\_cvl(int Ci, string fact,  vector<vector<string>>& goalList,

vector<vector<vector<string>>> ifStates,

vector<vector<vector<string>>> thenStates,

vector<vector<string>>& variList, vector<string> clauseList){

    int Ri;

    bool match = false;       //indicates whether a match to a rule has been made

    while((Ci < clauseList.size()) && (match == false)){   //while not at the end

of the clauseList,

and while a match

has not been made

        if(clauseList[Ci] == fact){                 //If clause Ci = the goal

            Ri = clause\_to\_rule(Ci, fact, goalList, ifStates, thenStates,

variList, clauseList);    //calculate the rule

number corresponding

to the clause

            match = update\_VLFC(Ci, Ri, fact, goalList, ifStates, thenStates,

variList, clauseList); //update the values of the

variables and check if a

match is made

        }

        Ci++;

    }

}

//updates the variables in the variable list that are relevant to rule Ri

bool update\_VLFC(int Ci, int Ri, string fact, vector<vector<string>>& goalList,

vector<vector<vector<string>>> ifStates,

vector<vector<vector<string>>> thenStates,

vector<vector<string>>& variList, vector<string> clauseList){

    string vari;                    //variable name of clause Ci

    int j = 0;

    bool found = false;      //indicates if vari is found in goalList or variList

    bool match = true;              //indicates if the value of vari is matching

with what is required of rule Ri

    int CiEnd = Ci + 1;             //the end of the clause list for rule Ri

    for(Ci; Ci < CiEnd; Ci++){      //while the end of the clause list has not

been reached

        vari = clauseList[Ci];      //read the first/next variable into vari

        found = false;              //reinitialize found to false for 'next'

variable

        if( (vari == " ") || (match == false) ){    //if there are no more

variables or the values of

the variables do not match

with rule Ri

           Ci = CiEnd;                                  //exit the loop

        }

        else{

            for(int i = 0; i < goalList.size(); i++){   //iterate through the

goalList

                if(vari == goalList[i][0]){  //if vari is found in goalList

                    found = true;                       //set found to true

                    if(goalList[i][1] == "y"){          //if vari is initialized

                            match = validate\_Ri(Ri, i, Ci, "i", "g", vari,

ifStates, thenStates, variList,

goalList); //check if the value

of vari matches

with rule Ri

                        i = goalList.size();            //exit the loop

                    }

                    else{                    //if vari is not initialized

                        processFC(vari, goalList, ifStates, thenStates, variList,

clauseList);  //restart process with vari as

the goal

                        match = validate\_Ri(Ri, i, Ci, "i", "g", vari, ifStates,

thenStates, variList, goalList);

//check if the value of vari matches

with rule Ri

                        i = goalList.size();            //exit the loop

                    }

                }

            }

            if(found == false){          //if vari is not found in goalList

                for(int i = 0; i < variList.size(); i++){   //iterate through

variList

                    if(vari == variList[i][0]){   //if vari is found in variList

                        found = true;                       //set found to true

                        if(variList[i][1] == "y"){        //if vari is initalized

                            match = validate\_Ri(Ri, i, Ci, "i", "v", vari,

ifStates, thenStates, variList,

goalList); //check if the value

of vari matches

with rule Ri

                            i = variList.size();            //exit the loop

                        }

                        else{                        //if vari is not initialized

                            string value = symptomQuestions(vari);  //ask user

for value

                            variList[i][1] = "y";       //set vari to initalized

                            variList[i][2] = value;         //set value of vari

                            match = validate\_Ri(Ri, i, Ci, "i", "v", vari,

ifStates, thenStates, variList,

goalList); //check if the value

of vari matches

with rule Ri

                            i = variList.size();            //exit the loop

                        }

                    }

                }

            }

        }

    }

    int i = 0;

    while (fact != goalList[i][0]){      //find the index of the goal in goalList

        i++;

    }

    if(match == true){               //if the vari values match with the rule Ri

        match = validate\_Ri(Ri, 1, j, "t", "g", fact, ifStates, thenStates,

variList, goalList); //set the value of the goal

        return match;

    }

    else{                 //if the vari values did not match with the rule Ri

        return match;

    }

}

//Finds the rule number for the corresponding clause number

int clause\_to\_rule(int Ci, string fact,  vector<vector<string>>& goalList,

vector<vector<vector<string>>> ifStates,

vector<vector<vector<string>>> thenStates,

vector<vector<string>>& variList, vector<string> clauseList){

    int Ri = Ci;                //Calculate the rule related to clause Ci

//the given formula was not used because the Ri and Ci are given by the index in the vectors, and the clause to rule ratio are 1 to 1.

//As a result for Ci = 0, Ri = ((Ci/1)+1) would result in Ri = 1

//to adjust for this the formula would have to be altered to Ri-1 = ((Ci/1)+1) to get the

//the correct clause number, which is equal to Ri = Ci

    return Ri;

}

## Program Run Copy

Is the patient experiencing urinary retention, paralysis, red/swollen skin,

warts/lesions, swollen digits, or burning eyes, nose, or mouth?

Enter 'y' for yes or 'n' for no.

y

Is the patient experiencing urinary retention?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing paralysis?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing red or swollen skin?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing warts or lesions?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing swollen digits?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing burning eyes, nose, or mouth?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing vomiting or nausea?

Enter 'y' for yes or 'n' for no.

y

Is the patient experiencing convulsions, fever, irritability, or dehydration?

Enter 'y' for yes or 'n' for no.

y

Is the patient experiencing convulsions?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing fever?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing irritability?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing dehydration?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing irregular breathing?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing confusion, seizures, headaches, fatigue or arrhythmia?

Enter 'y' for yes or 'n' for no.

y

Is the patient experiencing confusion?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing seizures?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing headaches?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing fatigue?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing arrythmia?

Enter 'y' for yes or 'n' for no.

n

Is the patient experiencing tingling or numbness?

Enter 'y' for yes or 'n' for no.

y

Is the patient experiencing muscle pains?

Enter 'y' for yes or 'n' for no.

n

----------------------------------------

| Diagnosis: Food Poisoning |

|======================================|

| Treatments: |

| Fluid replacement |

| Antibiotics |

| Antiparasitics |

| Probiotics |

----------------------------------------

Enter 'r' to restart the program or 'q' to exit.

q

# Program Analysis

## Analysis of the Program

The program was written based on the backward and forward chaining algorithm .docx files as opposed to the C language examples.

To start, all data structures such as goalList, ifStates, thenStates, etc… are all stored as vectors instead of arrays so that no more memory is used than necessary. This also allows for easily adding new rules for new poisons/toxins in the future.

The next modifications was the calculation of the clause number in rule\_to\_clause() and the rule number in clause\_to\_rule(). These changes were made for a couple of reasons, for both backward and forward chaining the vector indices were used instead of individually labeling the clauses and rules, so both Ci and Ri start with 0 instead of 1. For rule\_to\_clause() in the backward chaining process, Ci was instead calculated using the formula: Ci = 6\*Ri. Because the original formula, Ci = 6\*((Ri)-1)+1, would result in Ci = -5 instead of Ci = 0 when Ri = 0. For clause\_to\_rule() in the forward chaining process, Ri was calculated using the formula: Ri = Ci. Because of the before state reason, but also because the clause to rule ratio is 1:1.

Next, is the relationship between update\_VLBC()/update\_VLFC() and validate\_Ri(). It was decided to have update\_VLBC()/update\_VLFC() call validate\_Ri() for each iteration of the for-loop iterating through the clauseList for a particular rule. So that instead of checking if all variable matche the required values of ifStates after they’ve all been initialized, a rule is abandoned when the first variable doesn’t match. It was also decided to pass string variables ifthen and varigoal to validate\_Ri() so that validate\_Ri() quickly knows if the goal/fact is located in variList or goalList and if the goal/fact is being checked against the ifStates or if a goal is being assigned a value by the thenStates. If ifthen = “i” then a goal/fact is being checked against the ifStates, if ifthen = “t” then the goal is being assigned a value from the thenStates. If varigoal = “v” then the variable is in variList, if varigoal = “g” then the variable is in goalList.

## Results

The program produces the correct diagnosis and treatment for all possible input combinations.

## Memory Usage

Before a diagnosis is made 6.2 MB of data are being used in total, the goal list which exists throughout the program takes up 12 bytes, the data structures of the backward chaining portion of the program take up 60 bytes and the data structures of the forward chaining portion take up 48 bytes. After the diagnosis is made all the data structures from the backward and forward chaining proceses no longer exist and the program only takes up about 6.1 MB of data in total.

## Speed

The speed of the program is dependent on user input, not just the speed at which the user provides input, but also how many times the user is asked for input. To analyze the speed an average was taken for the shortest and longest paths that can be taken to complete a single run of the program.

The shortest path goes to rule 2, which only requires two "y" inputs and a visit to two rules in total.

The longest is an indirect path to rule 31, which includes three detours through the “indicative symptom” rules. Whereby the user inputs “y” for each question asking if the patient is experiencing any of the listed indicative symptoms, but “no” to each individual symptom resulting in a return to the main path. For a total of 31 rules visited, some considered more thoroughly than others.

The following are the ten times recorded for the shortest and longest paths as well as their averages in seconds.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | avg |
| Shortest | .028 | .030 | .031 | .028 | .033 | .034 | .030 | .029 | .029 | .032 | .0304 |
| Longest | .065 | .069 | .062 | .052 | .059 | .059 | .061 | .061 | .056 | .061 | .0605 |

Not factoring in wait time for user input, the program will take about .0304 to .0605 seconds to complete a single run.

## Effects of the Modifications

To start, using vectors instead of arrays for the data structures saves on space by not allocating any more space than is required.

The different formulas for calculating Ci in rule\_to\_clause() and Ri in clause\_to\_rule() saves time as fewer operations are made.

Using the vector indices instead of labeling the rules and clauses saves on memory usage.

The modification with the expected largest effect is the modification to the relationship between the update\_VLBC()/update\_VLFC() functions and validate\_Ri(). Abandoning a rule as soon as the first clause does not match the corresponding ifStates value saves time because then no time is wasted on comparing the following clauses to ifStates.

# Conclusion

Outside of learning how to implement backward and forward chaining. I’ve gained a better understanding of vectors, dynamic memory allocation, as well as setting up input files to be read into data structures more complex than a list or array.

I also learned how a variable can be represented in an unfamiliar way. Without the instructions, my initial thought would have been to represent the symptoms as typical string variables. Instead, I represented the symptoms in a vector of vectors of strings. Each inner vector represented a particular symptom with the first string being the variable name, the second being it’s initialization state, and the third being it’s value.

# Member Contributions

Karim Cisse:

Did not make any contributions to the project. He did not even communicate with the rest of the team except for once, where he apologized for not helping, told us he would help try to understand the provided C code examples, and then never communicated with us again. Several attempts were made to contact him in person as well as through text, discord, and an email from the professor.

Jayla Ajire:

Helped choose poisons and find their symptoms and treatments. Wrote the treatment .txt files in data/treatmentPlans/. Fully translated some functions and partially translated others from pseudocode to C++ code. Aided in debugging the final code.

Abigail De Rousselle:

Helped choose poisons and find their symptoms and treatments. Created backward and forward chaining flowcharts. Wrote out the knowledge base based on the flowcharts. Created the .txt files in data/backwardChaining/ and data/forwardChaining/. Created pseudocode for each function so that team mates could translate them to C++. Translated pseudocode that was not translated, or not fully translated by Jayla to C++. Aided in debugging the final code.

To be fair, I think some of the disproportion with the workload between Jayla and I was due to lack of communication and trust on my part. There was code written and questions asked that made me feel like I couldn’t rely on her to write working code on her own or debug in a timely manner. As a result, I chose to work on and finish the code without involving her much.

# References

“Alcohol poisoning,” *Mayo Clinic*, 19-Jan-2018. [Online]. Available: https://www.mayoclinic.org/diseases-conditions/alcohol-poisoning/symptoms-causes/syc-20354386. [Accessed: 26-Feb-2023].

“Asbestosis,” *Mayo Clinic*, 11-Feb-2022. [Online]. Available: https://www.mayoclinic.org/diseases-conditions/asbestosis/symptoms-causes/syc-20354637. [Accessed: 26-Feb-2023].

“Carbon monoxide poisoning,” *Mayo Clinic*, 16-Oct-2019. [Online]. Available: https://www.mayoclinic.org/diseases-conditions/carbon-monoxide/symptoms-causes/syc-20370642. [Accessed: 26-Feb-2023].

Department of Health & Human Services, “Mercury exposure and poisoning,” *Better Health Channel*, 21-Aug-2013. [Online]. Available: https://www.betterhealth.vic.gov.au/health/healthyliving/mercury-exposure-and-poisoning. [Accessed: 26-Feb-2023].

“Food poisoning,” *Mayo Clinic*, 30-Dec-2022. [Online]. Available: https://www.mayoclinic.org/diseases-conditions/food-poisoning/symptoms-causes/syc-20356230. [Accessed: 26-Feb-2023].

“Formaldehyde,” *Centers for Disease Control and Prevention*, 21-Oct-2014. [Online]. Available: https://wwwn.cdc.gov/TSP/MMG/MMGDetails.aspx?mmgid=216&toxid=39. [Accessed: 26-Feb-2023].

K. Cherney, “Arsenic poisoning: Symptoms, causes, and treatment,” *Healthline*, 24-Apr-2018. [Online]. Available: https://www.healthline.com/health/arsenic-poisoning. [Accessed: 26-Feb-2023].

“Lead poisoning,” *Mayo Clinic*, 21-Jan-2022. [Online]. Available: https://www.mayoclinic.org/diseases-conditions/lead-poisoning/symptoms-causes/syc-20354717. [Accessed: 26-Feb-2023].

“Tylenol poisoning (acetometophen overdose),” *WebMD*. [Online]. Available: https://www.webmd.com/a-to-z-guides/tylenol-acetaminophen-poisoning. [Accessed: 26-Feb-2023].

V. Higuera, “Antifreeze poisoning: Symptoms and treatment,” *Healthline*, 18-Sep-2018. [Online]. Available: https://www.healthline.com/health/antifreeze-poisoning. [Accessed: 26-Feb-2023].

My own past notes.