Intelligent Systems



Rule Based Systems

[30077548@students.southwales.ac.uk](mailto:30077548@students.southwales.ac.uk?subject=Rule%20Based%20Systems%20Report)

Sebastian Ha

30077548

Table of Contents

[Introduction 3](#_Toc210668180)

[Evaluation of the System 3](#_Toc210668181)

[Architecture and encapsulation 3](#_Toc210668182)

[Inference method 3](#_Toc210668183)

[Conflict resolution 4](#_Toc210668184)

[Traceability and decision making 4](#_Toc210668185)

[Dialogue quality and relevance 4](#_Toc210668186)

[Usability and user experience 4](#_Toc210668187)

[Consultation examples 5](#_Toc210668188)

[Wait consultation path 5](#_Toc210668189)

[Leave consultation path 6](#_Toc210668190)

[Knowledge acquisition and learning 7](#_Toc210668191)

[AI context 7](#_Toc210668192)

[Limitations and future work 7](#_Toc210668193)

[Conclusion 8](#_Toc210668194)

[Appendices 8](#_Toc210668195)

[Problem tree diagram 8](#_Toc210668196)

[Expert System Code 8](#_Toc210668197)

[Panopto demonstration 32](#_Toc210668198)

[References 32](#_Toc210668199)

# Introduction

Negnevitsky, 2011 defined expert systems as computer programs capable of performing at human expert level within a narrow problem domain, with rule-based expert systems being the most popular. They are designed to utilise knowledge and inference mechanisms to provide advice, solutions or recommendations in specialised areas by capturing and emulating the expertise of human specialists to support decision making and solve presented problems.

The problem domain I was assigned was the Wait for Table decision problem. This covers the everyday decision of a restaurant customer of choosing whether they should wait for a table by considering many factors such as how busy the restaurant is, waiting times for a table, whether there is another restaurant they can go to and other types of factors that can affect their resulting decision. For this problem I was allocated a decision tree to work off to help create the rule base for the expert system.

The purpose of expert systems in this assignment is to encode the decision process behind the wait for table decision tree. By having rules applied to the facts provided by the customer the system will be able to advise whether a customer should wait for a table. Further to this, expert systems are designed to capture specialist knowledge and apply it systematically to individual cases, supporting decision making in a transparent way.

# Evaluation of the System

## Architecture and encapsulation

The expert system created for the wait for table problem domain follows the architecture described by Kaisler, 1986. The system is made up of a knowledge base, represented by the RuleBase class, an inference engine, implemented in the InferenceEngine class and the working memory, defined by the WorkingMemory class. The inference engine encapsulates the knowledge base and the working memory as private members. It manages their interactions so that user and other components only interact with the engine rather than other components directly. By making the implementation of this system align with this architecture, it satisfies the criteria to be classed as a rule-based expert system.

Although it was preferred to use CLIPS which has a dedicated expert system shell, C++ was chosen to implement the architecture of this system. C++ allows me to have low-level control over data structures and execution, while still being able to represent the components of the expert system by using the language’s object-oriented features.

## Inference method

The knowledge base stores the set of rules taken from the problem domain tree, each with a resulting conclusion. The inference engine applies forward chaining matching the facts provided by the user and storing them within the working memory. After each input it compares the working memory against the rules within the rule base. It will either fire a rule if the antecedents are satisfied or use the traceStep() function to ask the next relevant question, applying conflict resolution by specificity.

## Conflict resolution

When deciding what question to fire next, if multiple rules are consistent with the facts within the working memory, the system applies conflict resolution by specificity. This method chooses the rule with the largest number of antecedents to represent the most precise match to the current data in the working memory. The inference engine does this via the conflictResolution() function. This ensures that unnecessary questions are avoided while keeping the consultation efficient.

## Traceability and decision making

After all the components of the expert system work together, it can deduce a decision from all the information gathered, the presentDecision() function within the inference engine serves as the explanation facility. It justifies the conclusion by displaying all the facts that support the decision made. This is essential in expert systems, as users understand how the system reached its conclusion through the list of reasons provided when displaying the decision, increasing the reliability and trust in the expert system’s decision making.

## Dialogue quality and relevance

The system avoids unnecessary questions being asked through the matchRules() and traceStep() functions. These functions ensure only rules with conditions relevant to the facts gathered in the working memory are processed in matchRules() so that traceStep() can make the decision of which question to ask next. An example of this is when asked if the user has an alternative restaurant they can go to and the response is no, all rules that have yes as the response are excluded so that questions that branch off that answer are never asked. This improves the efficiency of the system by keeping the consultation focused on the relevant information and makes the dialogue feel natural rather than repetitive.

## Usability and user experience

In addition to the reasoning logic, the system implements features that enhance user experience and usability. The input validation in place ensures a user can never provide an invalid input. The system will continue to prompt for user input with hints until the user enters a valid input. This stops errors from happening and ensures the system’s reliability. A replay option was included so that users could go through another consultation without needing to reboot the system. Furthermore, small touches such as a delay function and messages to alert users to what was happening in the system were added. For example, when the user chose to replay, a message appeared with a countdown prompt before the screen was cleared and the consultation restarted.

# Consultation examples

## Wait consultation path

A screenshot of a computer

AI-generated content may be incorrect.

A blue screen with white text

AI-generated content may be incorrect.

## Leave consultation path

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Both examples demonstrate how forward chaining, conflict resolution and traceability are applied. The consultations were efficient, only relevant questions were asked, and each conclusion was justified at the end. Overall, this shows the reliability of the system to provide appropriate advice while meeting the design goals.

# Knowledge acquisition and learning

Using the problem domain tree provided, the rule base was constructed manually into a set of IF-THEN pseudocode rules. From there, these rules were broken down into condition names with associated answers based on the branches observed from the tree. Each branch that led to a leaf became a rule, with the conditions forming the antecedents and the outcome becoming the rule’s decision. The knowledge acquisition was straightforward, as the tree was already structured in a way where answers didn’t need to be dynamically obtained from the user. However, in more real-world applications of expert systems, knowledge acquisition becomes a bottleneck. Hua, 2008 explains that traditional techniques such as observations or interviews are resource-intensive and prone to error, highlighting the simplicity of the approach taken for this system. To contrast, inductive learning techniques could help automate more complex applications by reviewing data provided and generating rules. This would reduce the reliance on manually acquiring knowledge minimising the bottleneck within expert systems.

# AI context

Early expert systems, including the type developed for this project, rely on clearly defined IF-THEN rules provided by human experts to form the rule base. These were among the earliest practical artificial intelligence technologies which became popular especially in the 1980s. Marsland, 2014 contrasted this historical approach with machine learning methods, which adapt automatically from data. Although rule-based systems offer transparency by showing how it reached a decision and traceability for their reasoning methods, they are difficult to scale and are not flexible. In contrast, machine learning is used to handle complex data in areas such as voice recognition, malware detection and even predictive diagnostics in healthcare. This comparison shows that these systems are effective in narrow domains with clear rules but struggle to deal with complexity and uncertainty compared to adaptive machine-learning approaches.

Forward chaining was chosen due to it being fact-driven and suited to the problem domain I was given. The questions needed to be asked step by step until a decision was made suiting the interactive consultation needed for this system. When put into a wider context, ethical and social considerations can arise if too much trust is placed in the system’s advice without understanding of the limitations. Fortunately, the decisions made are minor, they only show if the customer should wait or leave a restaurant. Furthermore, results are constrained using input validation methods, making sure progression of the consultation is linear within one of the paths in the problem tree. However, if the problem domain was within healthcare or law, outputs could be misleading and carry serious consequences that could affect someone’s life.

# Limitations and future work

Although the system works well with the wait for table problem domain, there are still limitations. The rule base is very specific to this problem domain and cannot be used for other decision-making contexts. The reliability of the system depends on how accurate and complete the rules are. If a condition were to be missing or a result provided by the user be unknown, it could lead to an invalid conclusion. Input validation was solidified to reduce the risk of these errors occurring. Finally, the system is likely to face scalability issues as the rule set grows and the number of conditions increase, leading to more difficult maintenance as time goes on.

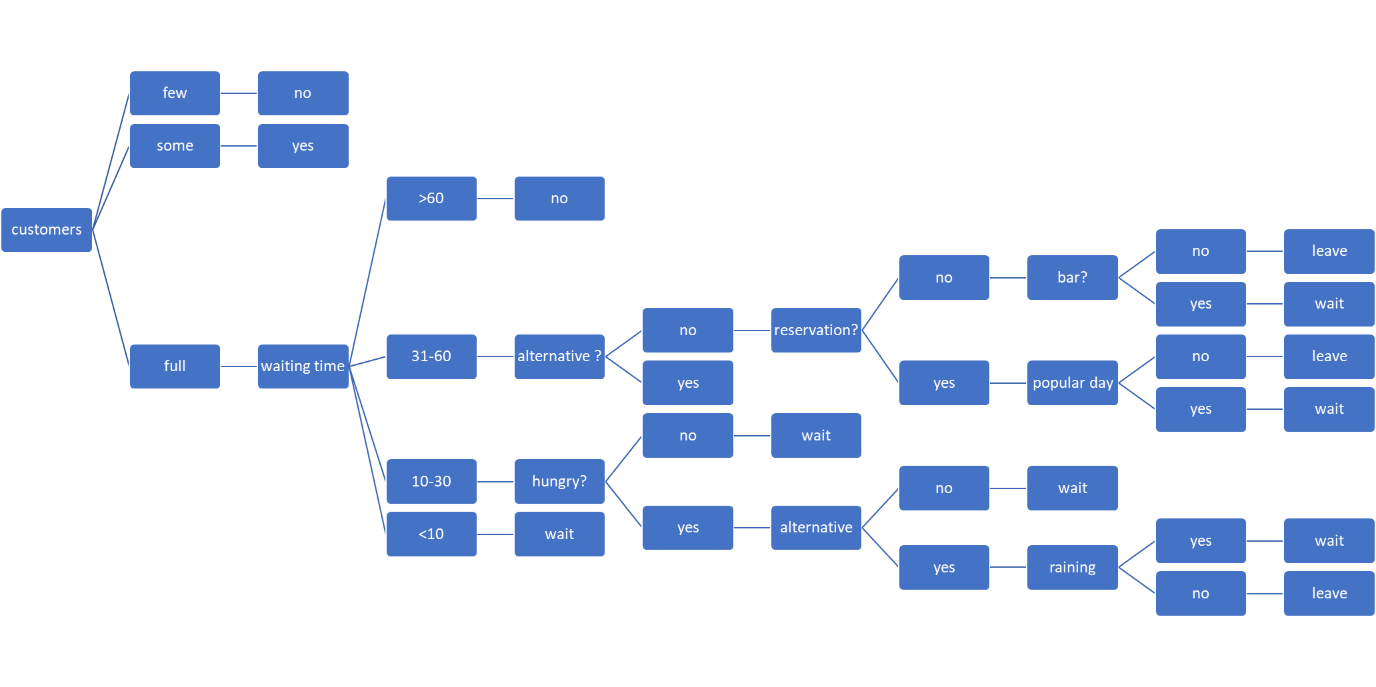
To improve the system in the future, it could be expanded to handle uncertainty by allowing answers outside the predefined sets in the rule base. Redesigning the rule base would make the system adaptable for different problem domains. Furthermore, exploring a hybrid approach that incorporates inductive learning could make consultations more dynamic, allowing the system to extract rules from examples and be able to adapt where responses would feel more like a conversation than a questionnaire.

# Conclusion

Summarise: implemented a forward-chaining expert system with conflict resolution by specificity; successfully advises customer whether to wait/leave; traceability achieved.

# Appendices

## Problem tree diagram



## Expert System Code

#include <iostream>

#include <cstdlib>

#include <utility>

#include <iomanip>

#include <vector>

#include <string>

#include <chrono>

#include <thread>

*/\**

*----------------------------------------------------------------*

*DECISION ENUM*

*----------------------------------------------------------------*

*\*/*

enum Decision {

    WAIT,

    LEAVE

};

*/\**

*----------------------------------------------------------------*

*FACT CLASS*

*----------------------------------------------------------------*

*\*/*

class Fact {

public:

    std::string factName = "";

    std::string factValue = "";

*//Constructor*

    Fact(std::string name, std::string value)

    {

        this->factName = name;

        this->factValue = value;

    }

};

*/\**

*----------------------------------------------------------------*

*RULE CLASS*

*----------------------------------------------------------------*

*\*/*

class Rule {

public:

    std::vector<Fact> conditions = {};

    Decision decision;

};

*/\**

*----------------------------------------------------------------*

*RULEBASE CLASS*

*----------------------------------------------------------------*

*\*/*

class RuleBase {

private:

    std::vector<Rule> rules =

    {

*//Few customers so leave*

        {

            {{"customers", "A few customers"}},

            LEAVE

        },

*//Some customers so wait*

        {

            {{"customers", "Some customers"}},

            WAIT

        },

*//Full of customers, waiting time greater than 60 mins so leave*

        {

            {{"customers", "Full of customers"}, {"wait time", "More than 60 minutes"}},

            LEAVE

        },

*//Full of customers, waiting time 31-60 mins, alternative restaurant yes, so leave*

        {

            {{"customers", "Full of customers"}, {"wait time", "Between 31 and 60 minutes"}, {"Alternative restaurant", "Yes"}},

            LEAVE

        },

*//Full of customers, waiting time 31-60 mins, alternative restaurant no, reservation no, bar no, so leave*

        {

            {{"customers", "Full of customers"}, {"wait time", "Between 31 and 60 minutes"}, {"Alternative restaurant", "No"}, {"Reservation", "No"}, { "Bar", "No" }},

            LEAVE

        },

*//Full of customers, waiting time 31-60 mins, alternative restaurant no, reservation no, bar yes, so wait*

        {

            {{"customers", "Full of customers"}, {"wait time", "Between 31 and 60 minutes"}, {"Alternative restaurant", "No"}, {"Reservation", "No"}, {"Bar", "Yes"}},

            WAIT

        },

*//Full of customers, waiting time 31-60 mins, alternative restaurant no, reservation yes, popular day no, so leave*

        {

            {{"customers", "Full of customers"}, {"wait time", "Between 31 and 60 minutes"}, {"Alternative restaurant", "No"}, {"Reservation", "Yes"}, {"Popular day", "No"}},

            LEAVE

        },

*//Full of customers, waiting time 31-60 mins, alternative restaurant no, reservation yes, popular day yes, so wait*

        {

            {{"customers", "Full of customers"}, {"wait time", "Between 31 and 60 minutes"}, {"Alternative restaurant", "No"}, {"Reservation", "Yes"}, {"Popular day", "Yes"}},

            WAIT

        },

*//Full of customers, waiting time 10-30 mins, hungry no, so wait*

        {

            {{"customers", "Full of customers"}, {"wait time", "Between 10 and 30 minutes"}, {"Hungry", "No"}},

            WAIT

        },

*//Full of customers, waiting time 10-30 mins, hungry yes, alternative no, so wait*

        {

            {{"customers", "Full of customers"}, {"wait time", "Between 10 and 30 minutes"}, {"Hungry", "Yes"}, {"Alternative restaurant", "No"}},

            WAIT

        },

*//Full of customers, waiting time 10-30 mins, hungry yes, alternative yes, raining yes, so wait*

        {

            {{"customers", "Full of customers"}, {"wait time", "Between 10 and 30 minutes"}, {"Hungry", "Yes"}, {"Alternative restaurant", "Yes"}, {"Raining", "Yes"}},

            WAIT

        },

*//Full of customers, waiting time 10-30 mins, hungry yes, alternative yes, raining no, so leave*

        {

            {{"customers", "Full of customers"}, {"wait time", "Between 10 and 30 minutes"}, {"Hungry", "Yes"}, {"Alternative restaurant", "Yes"}, {"Raining", "No"}},

            LEAVE

        },

*//Full of customers, waiting time less than 10 mins so wait*

        {

            {{"customers", "Full of customers"}, {"wait time", "Less than 10 minutes"}},

            WAIT

        }

    };

public:

    std::vector<Rule>*&* getRules();

};

std::vector<Rule>*&* RuleBase::getRules()

{

    return rules;

}

*/\**

*----------------------------------------------------------------*

*QUESTIONS CLASS*

*----------------------------------------------------------------*

*\*/*

class Questions {

public:

    std::vector<std::pair<std::string, std::string>> questions = {

        {"customers", "How many customers are in the restaurant?"},

        {"wait time", "How long is the waiting time before being seated in minutes?"},

        {"Hungry","Are you hungry?"},

        {"Alternative restaurant", "Are there alternative restaurants you can go to nearby?"},

        {"Reservation", "Do you have a reservation with the restaurant?"},

        {"Bar", "Does the restaurant have a bar?"},

        {"Raining", "Is it a rainy day?"},

        {"Popular day", "Is it one of the restaurants popular days?"},

    };

    std::vector<std::vector<std::string>> answers = {

        {"A few customers", "Some customers", "Full of customers"},

        {"More than 60 minutes", "Between 31 and 60 minutes", "Between 10 and 30 minutes", "Less than 10 minutes"},

        {"Yes", "No"},

        {"Yes", "No"},

        {"Yes", "No"},

        {"Yes", "No"},

        {"Yes", "No"},

        {"Yes", "No"}

    };

    std::vector<std::pair<int, int>> choiceNumbers =

    {

        {1, 3},

        {1, 4},

        {1, 2},

        {1, 2},

        {1, 2},

        {1, 2},

        {1, 2},

        {1, 2}

    };

};

*/\**

*----------------------------------------------------------------*

*WORKING MEMORY CLASS*

*----------------------------------------------------------------*

*\*/*

class WorkingMemory {

private:

    std::vector<Fact> facts = {};

public:

    void addFact(Fact f);

    bool isFactInWM(std::string name);

    std::string getFactValue(std::string name);

    std::vector<Fact>*&* getFactData();

    void clearFacts();

};

void WorkingMemory::addFact(Fact f)

{

*//Only add the fact into the working memory as long as there isn't duplicate found*

    if (!isFactInWM(f.factName))

    {

        facts.push\_back(f);

    }

}

bool WorkingMemory::isFactInWM(std::string name)

{

*//Runs a for loop to check if the fact is already in the working memory,*

*// used to help run traceStep, and find which answer to ask next*

    for (int i = 0; i < facts.size(); i++)

    {

        if (facts[i].factName == name)

        {

            return true;

        }

    }

    return false;

}

std::string WorkingMemory::getFactValue(std::string name)

{

*//We will always know if a fact is known by isFactInWM*

*// function before calling getFactValue so don't need to handle errors*

    for (int i = 0; i < facts.size(); i++)

    {

        if (facts[i].factName == name)

        {

            return facts[i].factValue;

        }

    }

*//If all else fails although this will never happen*

    return "";

}

std::vector<Fact>*&* WorkingMemory::getFactData()

{

    return facts;

}

void WorkingMemory::clearFacts()

{

    facts.clear();

}

*/\**

*----------------------------------------------------------------*

*INFERENCE ENGINE CLASS*

*----------------------------------------------------------------*

*\*/*

class InferenceEngine {

private:

    WorkingMemory wM;

    RuleBase rB;

    Questions q;

    int choice = 0;

    bool canExit = false;

    bool resetScreen = false;

public:

*//getters and setters for main loop*

    bool getCanExit();

    void setCanExit(bool var);

    bool getResetScreen();

    void setResetScreen(bool var);

*//Input validation function and reset choice*

    int validateInput(int minChoice, int maxChoice);

    void resetChoice();

*//Functions for main logic of inference engine*

    void askQuestion(std::string questionName);

    void fireQuestion();

    std::string traceStep();

    std::vector<Rule> matchRules();

    bool ruleReadyToFire(Rule*&* r);

    void presentDecision(Rule*&* r);

    Rule conflictResolution(std::vector<Rule>*&* candidates);

    void askToPlayAgain();

};

bool InferenceEngine::getCanExit()

{

    return canExit;

}

void InferenceEngine::setCanExit(bool var)

{

    this->canExit = var;

}

bool InferenceEngine::getResetScreen()

{

    return resetScreen;

}

void InferenceEngine::setResetScreen(bool var)

{

    this->resetScreen = var;

}

int InferenceEngine::validateInput(int minChoice, int maxChoice)

{

*//got this validation from my pokemon game year 1 and 2 assignments, if you need me to show you ask me*

    while (!(std::cin >> choice) || (choice < minChoice || choice > maxChoice))

    {

        std::cout << std::endl << "Invalid input, please choose a number between " << minChoice << " and " << maxChoice << " -> ";

*//clear the input and ignore any other characters on that line if user has typed in a sentence*

        std::cin.clear();

        std::cin.ignore(500, '\n');

    }

    return choice;

}

void InferenceEngine::resetChoice()

{

*//Simple but easier to type out a function for finding bugs later than just a line*

*//Had it happen in a past assignment and it was just easier to control from one area than multiple*

    choice = 0;

}

void InferenceEngine::askQuestion(std::string questionName)

{

*//Loop through the vector or string pairs to match the questionName passed in as a parameter*

*//To the question name within the vector of questions*

    for (int i = 0; i < q.questions.size(); i++)

    {

*//If questionName matches with the name in the vector then display the question to screen*

        if (q.questions[i].first == questionName)

        {

            std::cout << std::endl << q.questions[i].second << std::endl;

*//Display the answers appropriately using the answers variable, as we have the index it is easy to get*

            for (int x = 0; x < q.answers[i].size(); x++)

            {

                std::cout << x + 1 << " ->\t" << q.answers[i][x] << std::endl;

            }

*//Ask for the user choice*

            std::cout << std::endl << "Your choice -> ";

*//Validate input from user*

            choice = validateInput(q.choiceNumbers[i].first, q.choiceNumbers[i].second);

*//Use fact constructor with the information obtained*

            Fact f(q.questions[i].first, q.answers[i][choice - 1]);

*//Add that fact to working memory*

            wM.addFact(f);

*//Reset the choice so that it doesn't cause issues on the next question asked*

            resetChoice();

*//Return if the question has been found and asked so it doesn't continue looping unecessarily*

            return;

        }

    }

}

void InferenceEngine::fireQuestion()

{

*//If the workingMemory has no facts ask the first question, no need to trace the step yet*

    if (wM.getFactData().empty())

    {

        askQuestion("customers");

        return;

    }

*//Create rule object to store the rule that has been satisfied if one is found*

    Rule satisfiedRule;

*//Check if a rule has been satisfied*

    if (ruleReadyToFire(satisfiedRule))

    {

*//If so then present the decision to the user, then ask if the user wants to play again*

*//We want to make sure the loop definitely terminates even though setCanExit is called within presentDecision*

        presentDecision(satisfiedRule);

        return;

    }

*//If no rule is satisfied or isn't the first question then trace the next question to ask the user*

    std::string nextQuestionName = traceStep();

*//As long as the questionName doesn't return empty then we can ask the next question*

*//If somehow it does then something went wrong*

    if (!nextQuestionName.empty())

    {

        askQuestion(nextQuestionName);

    }

    else

    {

*//If all else fails somehow we need to just abort so nothing crashes*

        std::cout << std::endl << "Couldn't make a decision based on the answers you have provided. Exiting program shortly..." << std::endl;

        setCanExit(true);

    }

}

std::string InferenceEngine::traceStep()

{

*//Get a list of rules that could possibly be satisfied by the data currently in the working memory*

    std::vector<Rule> candidateRules = matchRules();

*//Make sure the program doesn't crash by returning an empty string if somehow the candidateRules are empty*

    if (candidateRules.empty())

    {

        return "";

    }

    else

    {

*//Apply conflict resolution by specificity, select the rule with the largest number of conditions*

        Rule chosenRule = conflictResolution(candidateRules);

*//Loop through the chosenRule until it finds a fact that is not in the workingMemory*

        for (int i = 0; i < chosenRule.conditions.size(); i++)

        {

            if (!wM.isFactInWM(chosenRule.conditions[i].factName))

            {

*//Return the question name so the next question can be asked*

                return chosenRule.conditions[i].factName;

            }

        }

    }

*//If something goes wrong just return an empty string*

    return "";

}

std::vector<Rule> InferenceEngine::matchRules()

{

*//Create vector of rules to store the matching rules inside*

    std::vector<Rule> matchingRules;

*//Cuts down the length of variables and also only gets a refer*

    std::vector<Rule>*&* rules = rB.getRules();

*//Initialise variable here and control value within loop*

    bool isMatching = true;

*//Loop through the list of rules*

    for (int i = 0; i < rules.size(); i++)

    {

        isMatching = true;

*//For each rule within the list of rules, check the conditions to make sure that the facts in the*

*//working memory match up with the facts within those rules*

        for (int j = 0; j < rules[i].conditions.size(); j++)

        {

*//Easier for looking through the code and seeing what is going on than repeatedly long names*

*//Passing in by reference so it doesn't make a copy everytime*

            Fact*&* currentCondition = rules[i].conditions[j];

*//If the fact from the working memory does not match with one of the facts within the rule*

*//Then it is disregarded and moves onto the next iteration*

            if (wM.isFactInWM(currentCondition.factName)

                && wM.getFactValue(currentCondition.factName) != currentCondition.factValue)

            {

*//No need to continue looking through this rule if it does not match, move onto the next one*

                isMatching = false;

                break;

            }

        }

*//If isMatching is still true meaning that the rule meets the criteria  then*

*// it gets pushed back into the matchingRules vector*

        if (isMatching)

        {

            matchingRules.push\_back(rules[i]);

        }

    }

    return matchingRules;

}

bool InferenceEngine::ruleReadyToFire(Rule*&* r)

{

*//Call a reference to the rules within rulebase,*

*//saves having to repeatedly call it and the length of variable names*

    std::vector<Rule>*&* rules = rB.getRules();

*//boolean to control whether a rule has been satisfied*

    bool ruleIsSatisfied = true;

*//Loop through vector of rules*

    for (int i = 0; i < rules.size(); i++)

    {

*//reset boolean value on every iteration of loop*

        ruleIsSatisfied = true;

*//Loop through condition within the rule, check for if rule does not match and change boolean to false*

*//That way an unsatisfied rule does not get fired*

        for (int j = 0; j < rules[i].conditions.size(); j++)

        {

*//Easier for looking through the code and seeing what is going on than repeatedly long names*

*//Don't want to make a copy every time we call this so using a reference*

            Fact*&* currentCondition = rules[i].conditions[j];

            if (!wM.isFactInWM(currentCondition.factName)

                || wM.getFactValue(currentCondition.factName) != currentCondition.factValue)

            {

*//No need to continue looking through this rule if it does not match, move onto the next one*

                ruleIsSatisfied = false;

                break;

            }

        }

*//if boolean still true then return as true and set the Rule object passed in as a parameter*

*// as the current index of the loop*

        if (ruleIsSatisfied)

        {

            r = rules[i];

            return true;

        }

    }

    return false;

}

void InferenceEngine::presentDecision(Rule*&* r)

{

*//First couple sentences for what is the end area*

    std::cout << std::endl << "A decision has been reached from the facts you have given." << std::endl;

    std::cout << "The decision is that you should " << (r.decision == WAIT ? "wait for a seat" : "leave the restaurant") << std::endl << std::endl;

    std::cout << "The reasoning for this is because of these facts you gave me:" << std::endl;

*//Loop through the conditions of the satisfied rule to print out the #*

*// reasoning for why the expert system came to that conclusion*

    for (int i = 0; i < r.conditions.size(); i++)

    {

        switch (i)

        {

        case 0:

            //Make a sentance of the number of customers within the restaurant

            std::cout << "-> " << r.conditions[i].factValue << " in the restaurant." << std::endl;

            break;

        case 1:

            //Switched up the wording so it will make more sense when displaying the wait time

            std::cout << "-> " << r.conditions[i].factValue << " was the " << r.conditions[i].factName << " to be seated." << std::endl;

            break;

        default:

            //Rest of results will just display as the title and whether it was yes or no

            std::cout << "-> " << r.conditions[i].factName << "?  -> " << r.conditions[i].factValue << std::endl;

            break;

        }

    }

    askToPlayAgain();

}

Rule InferenceEngine::conflictResolution(std::vector<Rule>& candidates)

{

    //Conflict resolution by specificity, by choosing this the system will tend to ask more

    //Informative questions reducing unecessary dialogue by being able to cut down the amount of questions quickly

    //Set the largest index as the first index

    int largestIndex = 0;

    //Loop through vector of candidate rules to find which rule has the largest amount of conditions

    for (int i = 0; i < candidates.size(); i++)

    {

        //set the next largest index if value at current index of the loop is larger

        if (candidates[i].conditions.size() > candidates[largestIndex].conditions.size())

        {

            largestIndex = i;

        }

    }

    //return the rule with the biggest amount of conditions

    return candidates[largestIndex];

}

void InferenceEngine::askToPlayAgain()

{

    //Asks the player if they would like to replay

    std::cout << std::endl << "Would you like to play again?" << std::endl;

    std::cout << "1| Yes" << std::endl << "2| No" << std::endl << std::endl << "Your choice -> ";

    //Get user input

    choice = validateInput(1, 2);

    switch (choice)

    {

    case 1:

        //clears the facts within working memory so that system can start from scratch

        std::cout << std::endl << "Great let's play again!" << std::endl;

        wM.clearFacts();

        setResetScreen(true);

        //Probably not needed but just incase anything else changes it to true

        setCanExit(false);

        break;

    case 2:

        //exits the program after letting the user know

        std::cout << std::endl << "No problem, exiting program now..." << std::endl;

        std::cout << std::endl << "Thank you for enquiring the wait for table expert system today, goodbye!" << std::endl << std::endl;

        setCanExit(true);

        break;

    default:

        //If an error were to ever happen then it would just exit the program

        std::cout << "Something went wrong here, exiting program..." << std::endl;

        setCanExit(true);

        break;

    }

}

/\*

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

            APPLICATION MANAGER CLASS

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

\*/

//THIS CLASS WAS TAKEN FROM MY DATA STRUCTURES AND ALGORITHMS SECOND ASSIGNMENT, ADJUSTMENTS MADE SO IT WORKS

//FOR THIS BUT IF YOU NEED PROOF I GOT IT FROM THERE ASK ME

class ApplicationManager

{

private:

    InferenceEngine iE;

public:

    void init();

    void showTitle();

    void update();

    void startDelay(double secondsToDelay, std::string messageToDisplay);

    void clearScreen();

};

void ApplicationManager::init()

{

    //Initialise colour and title of project

    system("color 1F");

    showTitle();

    //welcome message

    std::cout << "Welcome to the Wait for table Expert System" << std::endl;

}

void ApplicationManager::showTitle()

{

    //set the width for the title

    std::cout << "------------------------------------------------------------------------------------------------------------------------" << std::endl;

    std::cout << std::setw(70) << "-- -- Wait for Table Expert System -- --" << std::endl;

    std::cout << "------------------------------------------------------------------------------------------------------------------------" << std::endl;

    std::cout << std::endl;

}

void ApplicationManager::update()

{

    //continue to loop until the user decides to exit

    while (iE.getCanExit() == false)

    {

        iE.fireQuestion();

        if (iE.getResetScreen())

        {

            startDelay(2.0, "Loading");

            clearScreen();

            iE.setResetScreen(false);

        }

    }

    //Exit now that everything needed has been completed

    exit(0);

}

void ApplicationManager::startDelay(double secondsToDelay, std::string messageToDisplay)

{

    //Timer splits into 3 and adds a . to the screen then waits a quarter of the time passed in before moving on

    double third = secondsToDelay / 3;

    double quarter = secondsToDelay / 4;

    //display loading message or whatever is passed in

    std::cout << messageToDisplay;

    for (int i = 0; i < 3; i++)

    {

        //idea of the timer obtained from https://www.geeksforgeeks.org/how-to-add-timed-delay-in-cpp/ , then implemented it for microseconds so i could use a double value

        std::this\_thread::sleep\_for(std::chrono::milliseconds(static\_cast<int>(third \* 1000)));

        std::cout << ".";

    }

    //last touch so user can see it has finished loading

    std::this\_thread::sleep\_for(std::chrono::milliseconds(static\_cast<int>(quarter \* 1000)));

    std::cout << std::endl;

}

void ApplicationManager::clearScreen()

{

    //Clear the screen and show the title at the top before next set of outputs are displayed

    system("cls");

    showTitle();

}

/\*

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

            MAIN

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

\*/

int main()

{

    //Call applicationManager object

    ApplicationManager appMgr;

    //Initialise the screen

    appMgr.init();

    //Run main loop

    appMgr.update();

    return 0;

}

## Panopto demonstration

A Panopto demonstration has been created for this assignment, you can find there video here at

# References

Hua, J. (2008) ‘Study on knowledge acquisition techniques’. 2008 Second International Symposium on Intelligent Information Technology Application, pp. 181–185. doi: https://doi.org/10.1109/IITA.2008.152.

Kaisler, S. (1986) ‘Expert systems: An overview’. IEEE Journal of Oceanic Engineering 11(4), pp. 442–448. doi: https://doi.org/10.1109/JOE.1986.1145205.

Marsland, S. (2015) Machine learning : an algorithmic perspective /. Second edition. CRC Press,.

Negnevitsky, M. (2011) Artificial intelligence: a guide to intelligent systems. 3rd ed. Pearson Addison Wesley. Available at: https://www.vlebooks.com/Product/Index/305020?page=0&startBookmarkId=-1 (Accessed: 5 October 2025).