VEHICLE DIAGNOSIS AND REPAIR RECOMMENDATIONS

Project 1 – CS4346: Advanced Artificial Intelligence

Abstract

This report examines the design, implementation, and execution of an expert system, which diagnoses vehicle issues and recommends repairs to the user. Report covers how the program employs course concepts and how it makes use of data structures and algorithms.

Author: Borislav S. Sabotinov

bss64@txstate.edu | TXST Student ID: A04626934

Team members: David Torrente and Randal Henderson

Table of Contents

List of Tables

List of Figures

1 Introduction

The application covered by this report serves as a virtual vehicle repair expert. Using Backward Chaining, it asks users questions to diagnose issues. The results obtained are used in Forward Chaining to identify a possible repair that would resolve the issue.

Modern vehicles are comprised of numerous complex systems. A typical vehicle is assembled from around 30,000 individual parts[1]. To further complicate matters, different manufacturers use different parts and employ wildly differing designs in their products. This makes diagnosing issues in a vehicle a challenging task. Most owners only possess surface-level knowledge of how their vehicle works. The intent of this system is to serve as an expert knowledge base; through interactions with this program, users may leverage this knowledge to diagnose issues and receive recommendations for necessary repairs. In a sense, the system emulates an expert mechanic servicing the vehicle – the user is asked a series of questions, prompting them to inspect various parts of the vehicle to arrive at a conclusion. Suppose a user has a vehicle that does not start, and they are not sure why. Normally, they would bring the car in for service at a repair shop. The mechanic would apply diagnostic techniques, inspecting the starter, timing belt, battery voltage, and other components to determine the issue. They may inspect warning lights displayed on the dashboard panel and reference manufacturer manuals or plug in code readers to extract error codes if the vehicle is equipped with an on-board diagnostics computer system, which monitors a vehicle's electronic sensors.

Our system's knowledge base and inference engine are separated. The knowledge base provided along with the submission is specifically tailored for vehicle issue diagnosis. There is, however, nothing preventing this program from being used for other applications (e.g., medical diagnosis) provided a knowledge base file is available. The system is dynamic; there is nothing hard coded.

The system will ask questions and use the answers provided to either provide a diagnosis of the issue or offer a repair recommendation.

1.1 Purpose of Report

The intent of this report is to demonstrate a thorough understanding of class concepts and provide a detailed explanation of the Project 1 vehicle diagnosis application. This report will attempt to exhaustively cover every aspect of the design, implementation, execution, and analysis of the program.

1.2 System Scope

The application for Project 1 is an intelligent expert system for diagnosing vehicle issues and recommending repairs to the user. It employs Backward Chaining to diagnose the issue. The results are then used by Forward Chaining to identify a repair to recommend.

The system uses as much realistic, real-world vehicle and mechanical data as possible. The system is not intended to serve as an actual diagnosis and repair system for real-world applications and it is primarily intended to demonstrate a thorough understanding of advanced artificial intelligence concepts for CS5346.

[1] Toyota Question Room. https://bit.ly/3sLlpyy

The system shall:

- Read in data from external file(s).
- Prompt users for data if the required values are not available.
- Provide users the ability to print out the entire knowledge base of the expert system.
- Attempt to be robust and handle error cases, such as malformed data or missing files.

The system is designed to compile and run on Texas State Linux hosts. It was additionally tested on Windows – provided certain prerequisites are met, it can execute on the Windows OS as well.

Complete code and documentation, as well as this report, may be found on GitHub here: https://github.com/TXST-CS5346-AI/project-one

1.3 Terms and Definitions

Table 1: Terms and Definitions

Term	Definition
The system	"The system" in this report shall refer to the application designed and implemented for Project 1, an intelligent expert system. The knowledge base files provided are specifically for vehicle diagnostics and repair recommendations but the inference engine is a separate entity and may handle any knowledge base, not just vehicle diagnostics.
KB	Knowledge Base. KB throughout the report shall refer to the knowledge base, containing logic for the vehicle diagnosis and repair system.
Statement	A statement is a single line in the knowledge base file. It has zero or more clauses and one and only one conclusion. Premises are on the left side, conclusions on the right. A conclusion may be used as an intermediate variable (premise) on the left-hand side of a statement. In our KB file, we used the following symbols: • THEN indicated by : • AND indicated by ^ • True or false were indicated by 'y' or 'n' Here is an example of a statement (single line in KB file): `issue = Failure to Start ^ has_fuel = y ^ has_voltage = n : repair = Dead Battery, Change the battery.` The statement shall be read as follows: IF issue equals "Failure to Start" and *has_fuel* equals yes and *has_voltage* equals false THEN conclusion is repair equals "Dead Battery, Change the battery."
Clause	A clause constitutes a single element of a statement. It may be either a premise or a conclusion (or in the case of "issue" it may serve as both premise in one statement and conclusion in another).

Premise	A premise is a variable, whose value is used to ascertain if a conclusion is valid. For example, *has_voltage* is a variable and if it is set to 'n' for No/False, then we determine the repair conclusion has a value of "Dead Battery, Replace the battery."
Conclusion	A conclusion may only occur once in a statement, on the right-hand side. It is a variable that is determined by the premise clauses on the left side. Our program has two possible conclusion variables: **issue** and **repair**.
CLI	Command line interface, a text-based user interface for interacting with this application on Texas State's Linux servers.

1.4 Report Overview

The Table of Contents shows where each section is located and contains a list of tables and figures used throughout the report for ease of reference.

Section 1 covers general, introductory information about the system created for this project. It shows terms and definitions used in this report.

Section 2 describes the system at a high level. It defines the intended audience and the types of users served by this system, a list of features. Any constraints encountered, or assumptions made, during design and implementation are also outlined here.

Section 3 dives into the system design in greater detail. It shows how to build and run the program, providing a complete listing of all available commands. We examine the decision tree diagram for the expert system and all its subsystems. We look at the variables list we derive from this decision tree. Finally, the system's class diagram is presented.

Section 4 dives into the source code and outlines each class, as well as any noteworthy data structures or algorithms employed.

Section 5 comprehensively covers the system by exercising it against all unique available options. In addition to the minimum required three sample runs, five additional runs are performed. Two issues diagnosis, two repair recommendations, printing the knowledge base, printing the help menu, and two intentional error scenarios are all exercised. Additionally, we consider inconclusive results, where the user exhausts the available options and the system does not contain a repair recommendation. A complete text output is presented in Appendix C.

The appendices provide large images of all diagrams (where necessary) for readability, as well as the complete source code. Any diagram too large to be readable in its entirety is dissected into sections and each section is presented separately, expanded to fill the screen.

1.5 Contributions

1.5.1 Individual Contributions

My contributions to this project include:

- Created the GitHub repo and initial workspace.
- Coded a Qt5 C++ GUI, which we collectively decided to scrap towards the end due to time constraints and difficulty in building the project.
- Designed and created the Decision Tree Diagram in its entirety.
- Created the list of variables derived from the Decision Tree.
- Facilitated pull requests/merges to ensure code synchronization; served as a project manager (of sorts) to facilitate meetings.
- Coded a way to further separate Knowledge Base from Inference engine, by adding a set to keep track of conclusions without duplicates.
- Touched all source code by refactoring code, adding features and error handling.

A more detailed list of issues I closed (via pull requests to the codebase) may be found below in Figure 2. Please note that all team members contributed equally and participated in the project, though GitHub may not reflect this. I was more comfortable using Git and GitHub, as I use these tools daily at work. I used these tools to maintain the code base and track most of my work.

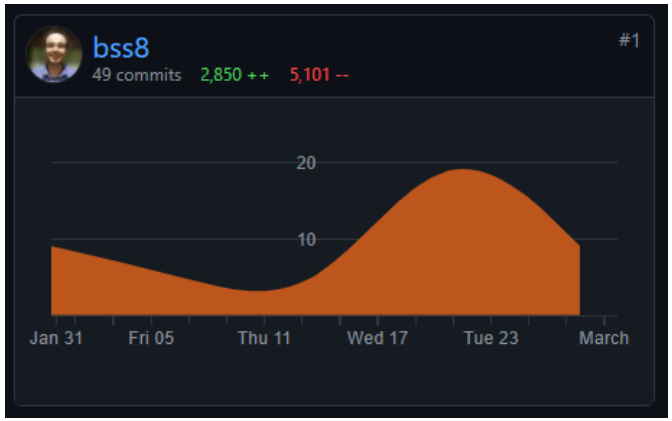

Figure 1: GitHub Contributions

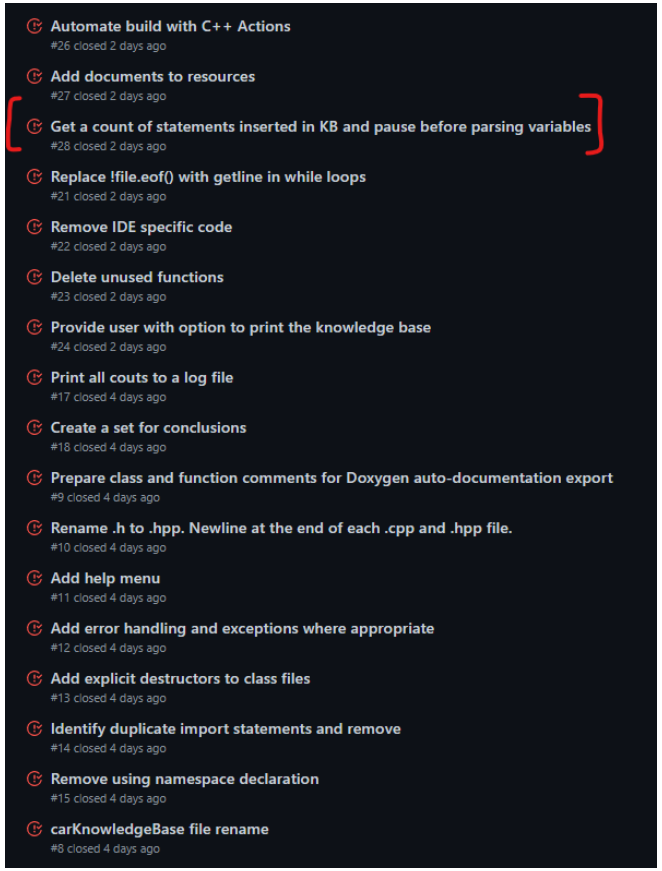

Figure 2: Issues closed by Boris; #28 (red brackets) implemented by Randy

1.5.2 Group Member Contributions

I felt all members were active participants in this project and came together to make it work. All members participated in planning, design, implementation, team meetings, and debugging activities. We were the first team to assemble and quickly came together to begin work on this project.

Please note that the power outage and water shutoff in Texas adversely impacted work on this project and all members were affected in some way. Despite the challenges, we worked as a team to successfully complete the system and I look forward to working with both David and Randy on the remaining projects.

Randy worked on the code parsing the knowledge base and building the premise and conclusions lists. David worked on the inference engine, re-designing how the new program performs back and forward chaining.

There were 85 commits and 19 pull requests into the code repository.

2 System Description

2.1 System Perspective

While the system is designed with a real end user in mind – someone who may need to diagnose and repair their car – the system is also tailored for computer science students and faculty. The output is verbose and intentionally signals what is going on behind the scenes during execution. Detailed sample runs may be found in section 5, with a complete textual output of sample run #1 in Appendix C.

2.2 System Features

The following lists shall comprise a complete listing of the system's available features:

1. Display a help menu on request.
2. Display a welcome message to the user on start-up.
3. Parse the knowledgeBase.txt data file and print results to the console as it does so.
4. Pause after loading the KB file to allow the user time to review.
5. Upon continuing, process the varialbesList.csv file.
6. Prompt the user if they want to display the KB in human readable format. Display on yes, skip on no.
7. Prompt the user for a conclusion to solve.
8. Use Backward Chaining to diagnose the issue. If the user is only interested in "issue" conclusion (i.e., a diagnosis), stop there. Otherwise pass results to Forward Chaining and determine recommended repair.
9. If the user exhausts all available options, display result as inconclusive.
10. Handle common error scenarios (missing KB file, incorrectly formatted statement in the KB file, etc.)

2.3 Design and Implementation Constraints

The system was built with Texas State Linux servers in mind. While it may operate properly in other environments (e.g., Windows-10, Ubuntu, SLES, etc.) it is guaranteed to run on either the Eros or Zeus TXST servers. Section 3.1 outlines detailed build and execution instructions, which are also available in a shortened format in the Project1-README-A04626934.txt file submitted alongside the source code.

Our team initially wanted to create a graphical user interface, using either the Qt5 framework or C#. Due to technical challenges, time constraints, and the unexpected state weather emergency, the group decided to scrap this activity and focus on the core tasks.

2.4 Assumptions and dependencies

We assume users of this program will be familiar with using a command line interface (CLI) on the Texas State Linux servers.

The program is dependent on:

- A C++ compiler, such as the GNU C++ compiler for Linux (g++).
- The C++11 language standard.

3 System Design and Specification

There are 46 variables used in the knowledge base. For a complete listing of all variables and their purpose, refer to Section 3.4.

The premise variables are designed to be “yes or no” type only. Each question is intentionally phrased as a yes or no question. For example, instead of asking the user to enter in the battery voltage as a number and determine if it is sufficient in the program, the user is asked only - "Is sufficient voltage available in the battery (Y/n)?"

The reason for this is because each battery manufacturer may have different voltage requirements. The user must refer to their user manual and determine the value. As an additional example - instead of asking the user to input their GPA and have the system determine whether it is above or below a certain threshold, the question would ask the student directly if their GPA is above a certain value (e.g., "Is your GPA above 3.5 (Y/n)?")

This simplifies the design and the user experience, allowing the user to enter only yes or no for each question.

3.1 How to Build and Run the Program

The program consists of these four files:

1. Project1-A04626934.cpp: source code file
2. Project1-KB-A04626934.txt: knowledge base file
3. Project1-README-A04626934.txt: instructions for building and running the program
4. Project1-Vars-A04626934.csv: variables for the knowledge base

This application is primarily designed to run on Texas State (TXST) Linux servers.
Eros: EROS.CS.TXSTATE.EDU (147.26.231.153)
Zeus: ZEUS.CS.TXSTATE.EDU (147.26.231.156)

You may use WinSCP, FileZilla, or an equivalent FTP software to transfer the project files to a TXST Linux host.

1. Project1-KB-A04626934.txt and Project1-Vars-A04626934.csv must be in the same directory as the Project1-A04626934.cpp file.
2. Build with this command:
    - **g++ -o Project1 Project1-A04626934.cpp -std=c++11**
3. Run with this command:
    - **./Project1**
4. To see help menu (optional):
    - **./Project1 -h**

3.2 Why use C++11?

A small aside but worth mentioning. Why use the C++11 compiler? C++11 now supports:

- lambda expressions,
- automatic type deduction of objects,
- uniform initialization syntax,
- delegating constructors,
- deleted and defaulted function declarations,
- nullptr,
- rvalue references

The language was overhauled with new container classes, algorithms, smart pointers, async capability, and multithreading support, in addition to other useful features. The C++11 compiler is readily available on Texas State Linux servers.

3.3 Decision Tree Diagram

The entire decision tree diagram is provided at the end of this report in Appendix A. It is a large diagram; for readability, it is broken up in sections and expanded here. There are a total of 88 nodes, 34 repair conclusion nodes and 8 issue conclusion nodes for a total of 42 conclusions (rectangle type nodes).

Node 1 in Figure 3 below is the start node. We begin by asking the user if there is an issue with the vehicle. If there is no issue, then the issue variable is set to "No Issue" and the repair variable is set to "No repair necessary."

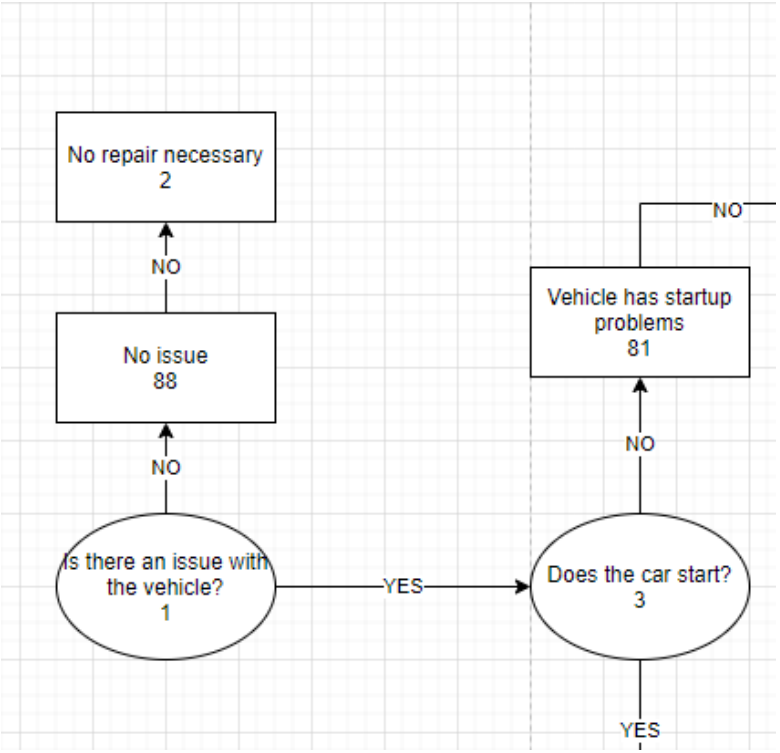

Figure 3: Beginning of Decision Tree

3.3.1 Failure to Start Diagnosis

From Figure 3 above, we ask the user if the car is starting. If the user selects no, issue is set to “Failure to Start,” which will be used as an intermediate variable in the premise list. We enter in the green box in Figure 4 – subsystem A of the expert system, which deals with vehicle failure to start diagnosis. Figure 4 displays all the possible options. There are 5 possible repair recommendations in this subsystem.

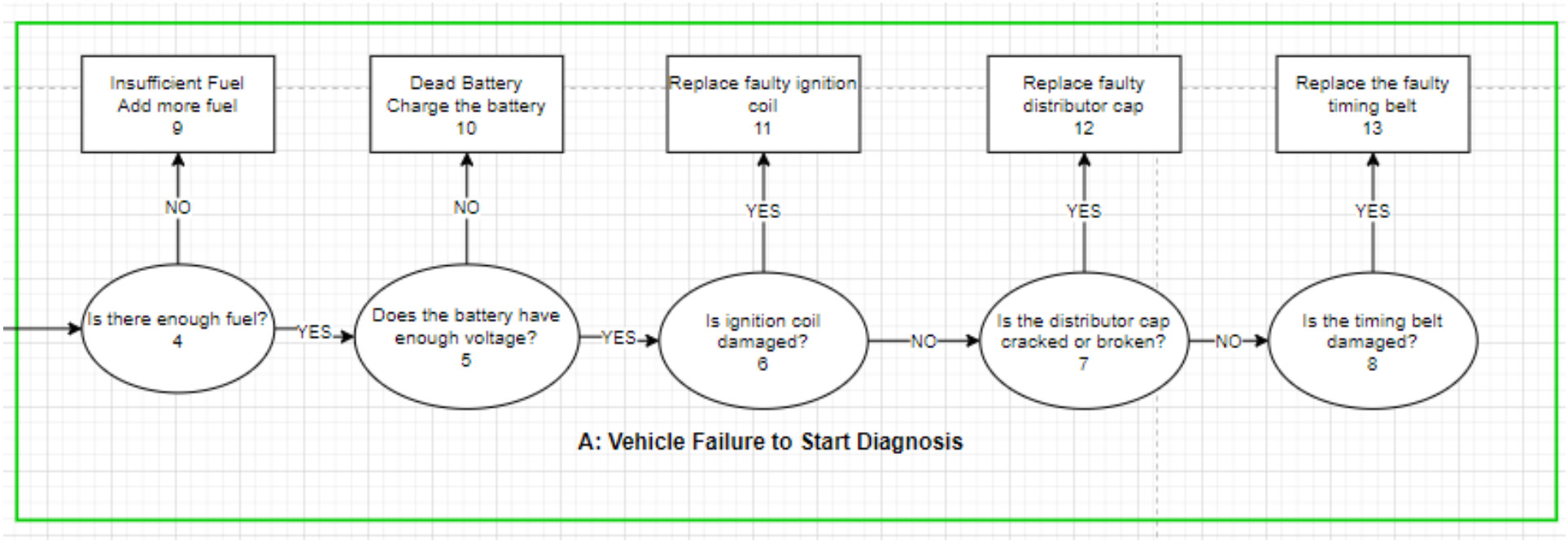

Figure 4: Failure to Start Diagnosis Subsystem of Decision Tree

3.3.2 Vehicle Noise Diagnosis

If the user selects yes when asked if the car is starting, we continue looking for the issue. Next the user is asked if there is a noise problem with the vehicle. If the user chooses yes, we enter subsystem B1 of the expert system in Figure 5, dealing with vehicle noise issue diagnosis. There are two main sections here – noise while the vehicle is stationary and noise while the vehicle is in motion. There are six repair recommendations available in this subsystem. For readability, the diagram is enlarged on the next page.

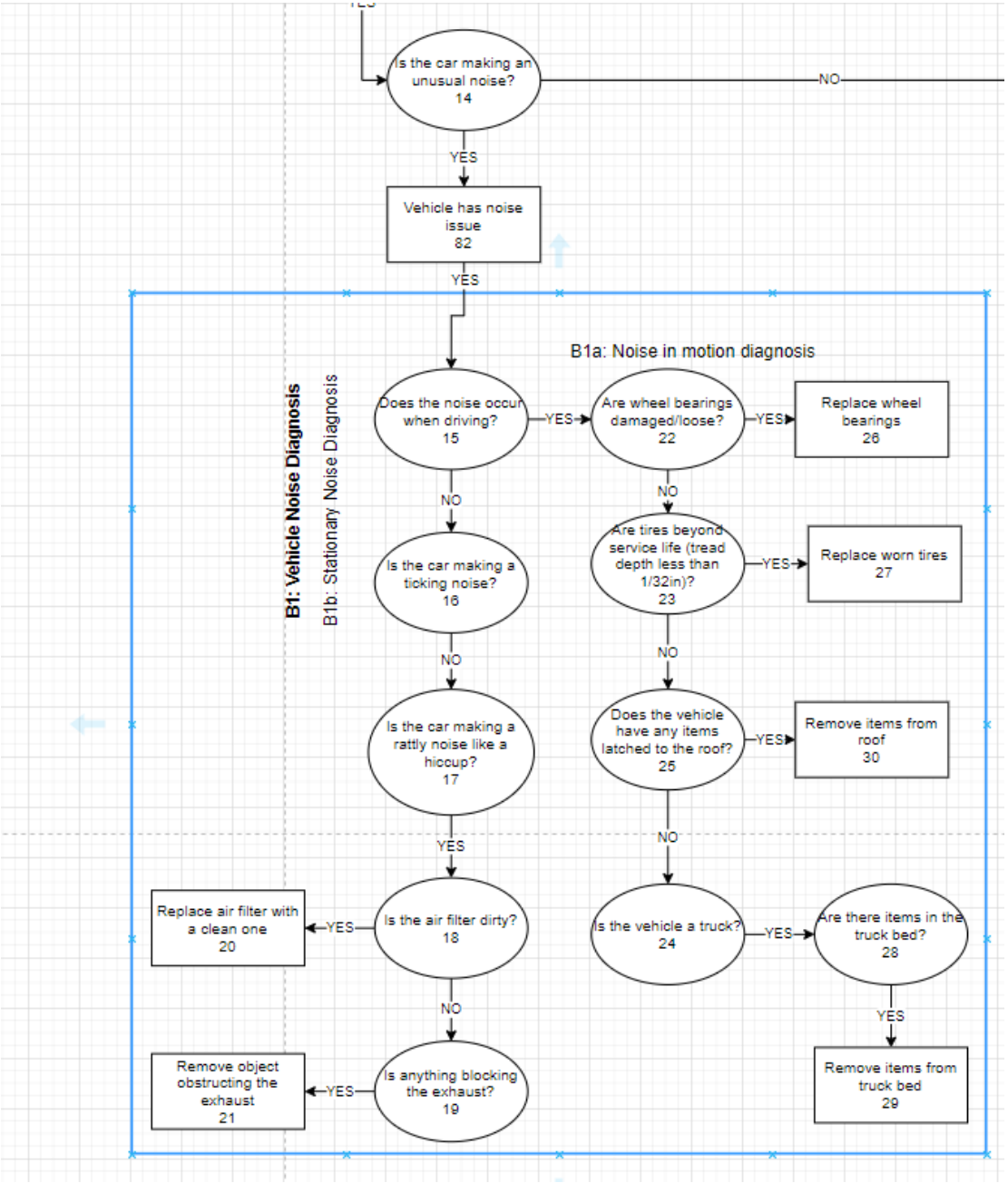

Figure 5: Vehicle Noise Diagnosis Subsystem of Decision Tree

3.3.3 Vehicle Overheating Diagnosis

If there is no noise issue, we continue the diagnosis. If the vehicle is overheating, issue is set to “Overheating Issue” and we enter the red box in Figure 6 – subsystem B2 dealing with vehicle overheating issues diagnosis. There are three possible repair recommendations in this subsystem. For readability, the diagram is enlarged on the next page.

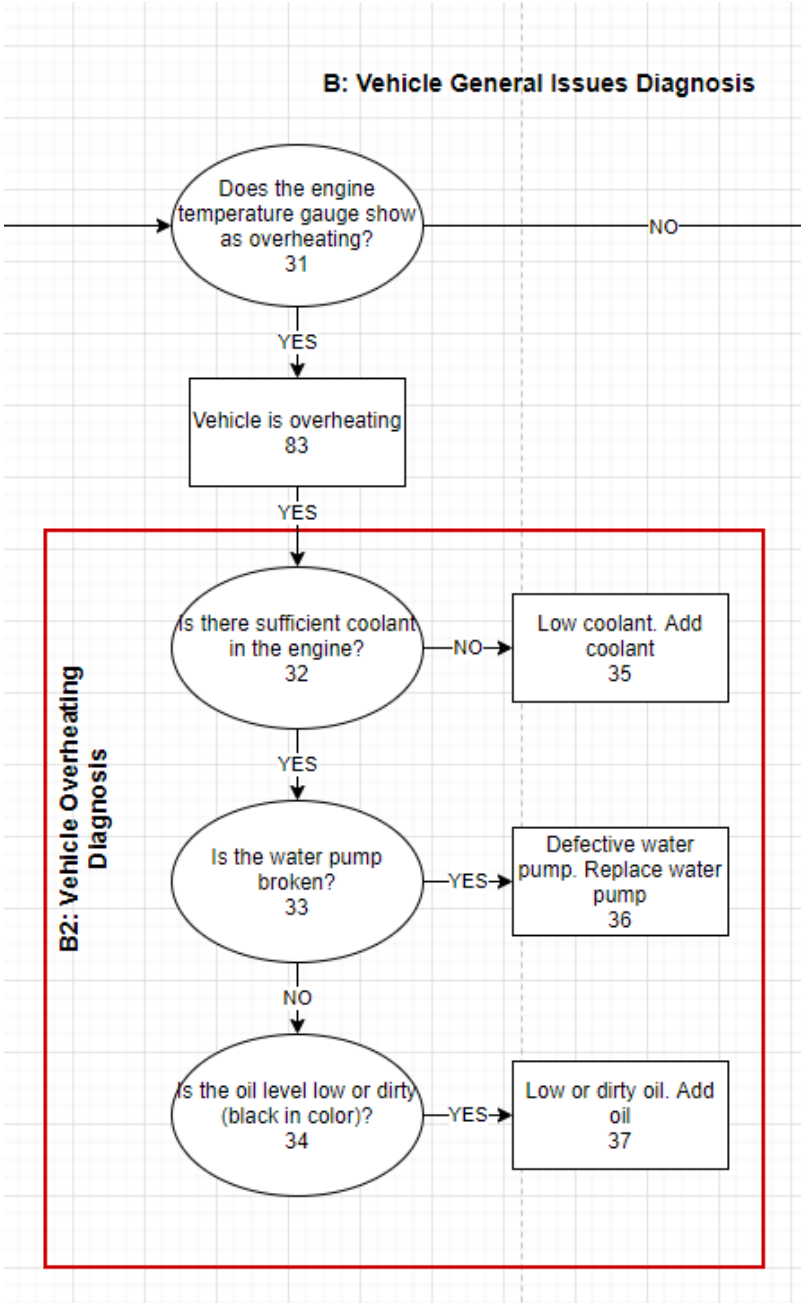

Figure 6: Vehicle Overheating Issue Diagnosis Subsystem of Decision Tree

3.3.4 Electrical Issues Diagnosis

The diagram for this subsystem is too large to be clearly displayed here. For readability and ease of access, it is included at the end of this report under Appendix A and is split up into two images.

If the vehicle starts, there is no noise issue, and the vehicle is not overheating, the user is asked if there are any noticeable electrical issues with the vehicle. If the answer is yes, we enter the yellow box in Figures 31 and 32 in Appendix A – subsystem B3 dealing with electrical issues diagnosis. There are 11 repair recommendations available in this subsystem.

3.3.5 Power Steering Issues Diagnosis

If there is no electrical issue, or the issue is resolved from previously provided diagnosis and repair recommendations, we ask the user if the steering wheel is turning. If the user answers no, we enter the pink box in Figure 7 – subsystem B4 dealing with diagnosing power steering issues. There are two repair recommendations available in this subsystem. For readability, the diagram is enlarged on the next page.

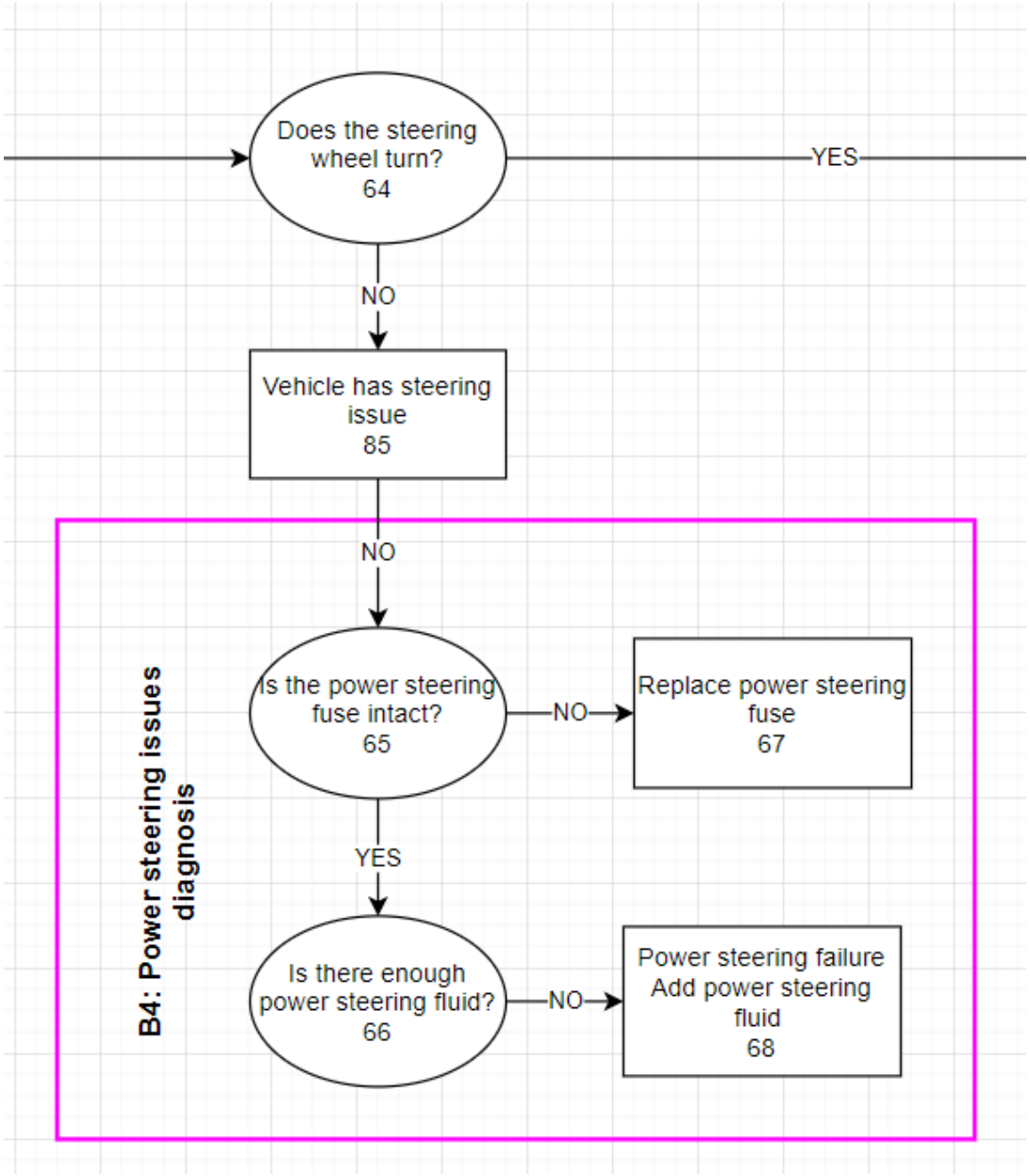

Figure 7: Power Steering Issue Diagnosis Subsystem of Decision Tree

3.3.6 Tire Issues Diagnosis

If there are no power steering issues, we ask the user if any of the tires are visibly deflated or if the tire pressure warning light in the cabin is on. If the user answers yes, then we enter the brown box in Figure 8 – subsystem B5 dealing with diagnosing tire issues. There are two available repair recommendations in this subsystem. For readability, the diagram is enlarged on the next page.

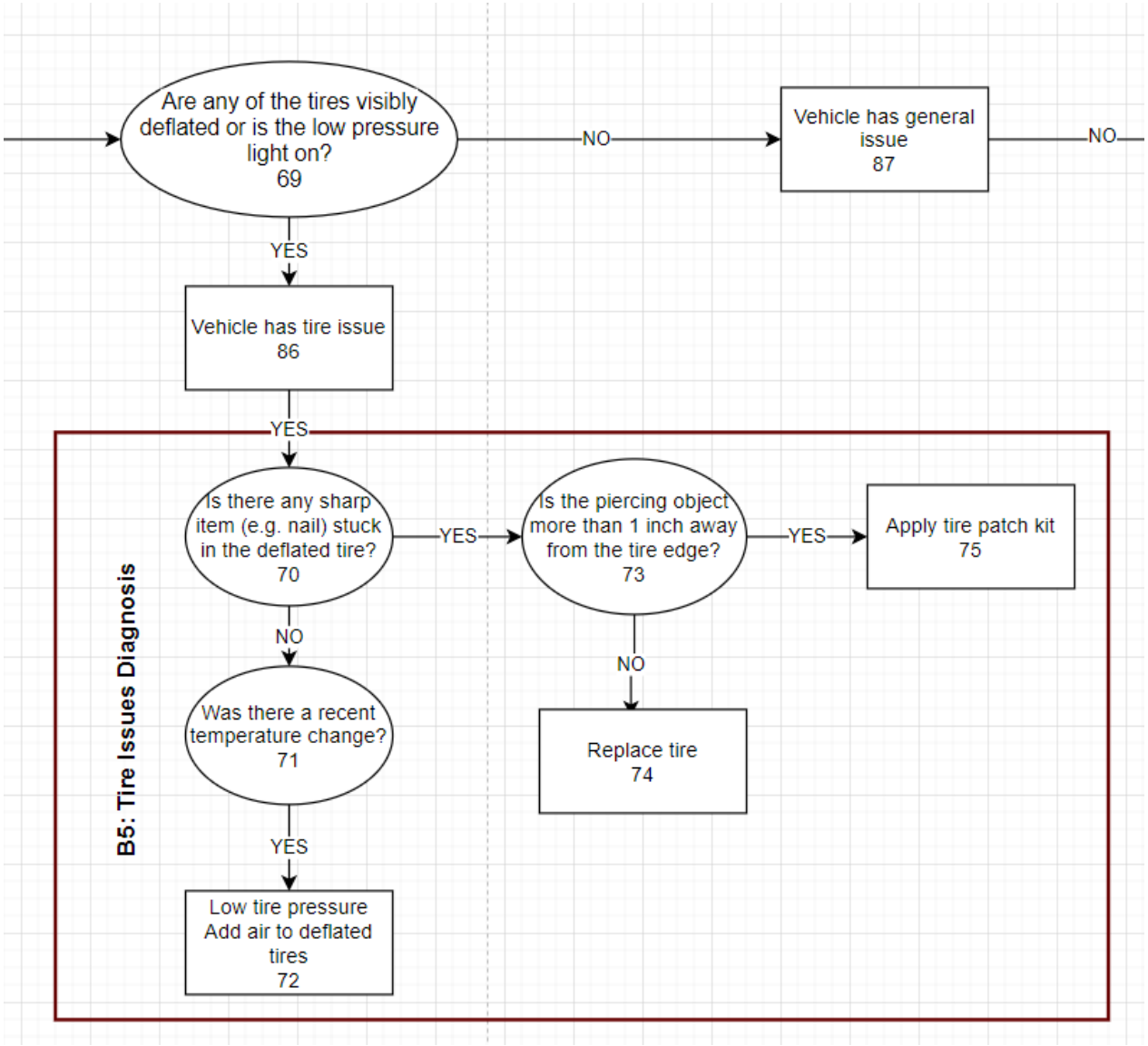

Figure 8: Tire Issue Diagnosis Subsystem of Decision Tree

3.3.7 General Diagnostics

If we exhaust all options up to this point, we automatically set the issue variable to “General Issue” and enter the black box in Figure 9 – subsystem B6, which deals with general vehicle issues diagnosis. There are three possible repair recommendations in this subsystem. It should be noted that if the user exhausts the available questions in any subsystem branch (i.e., hit a dead end), the conclusion will be inconclusive. We see an example of this in Section 5.8.

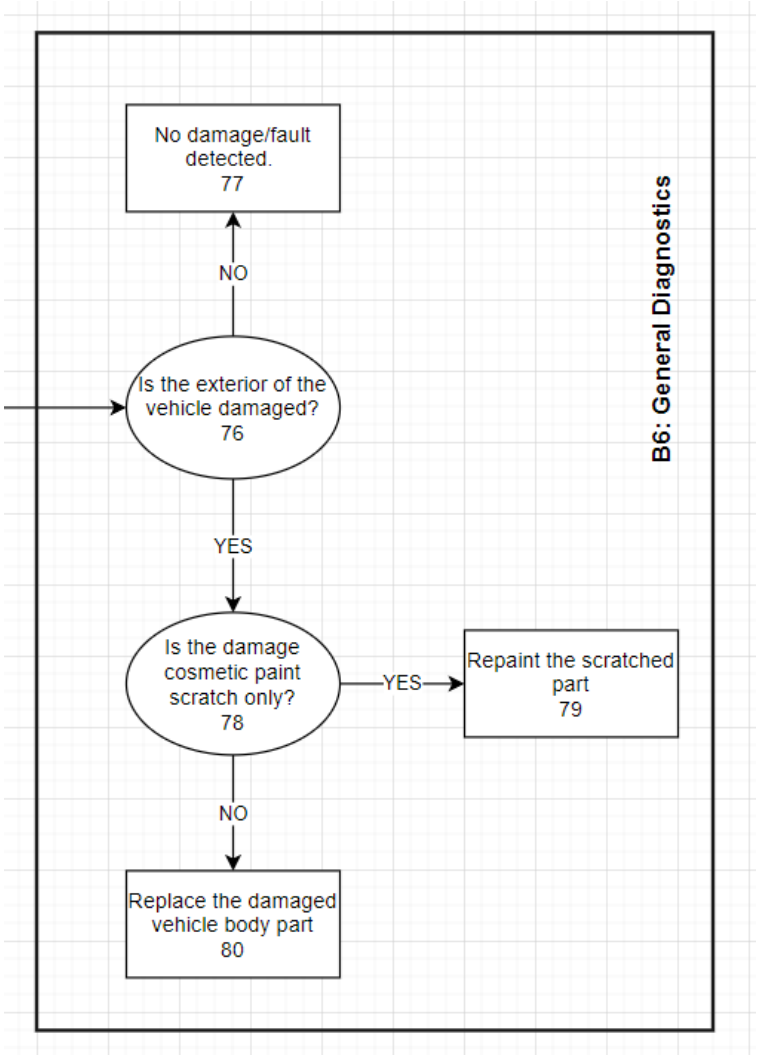

Figure 9: General Diagnosis Subsystem of Decision Tree

3.4 Variables List for Decision Tree

Table 2 below lists all 46 variables for this system, their name, their type, a description, the node with which the variable is associated, and the subsystem to which they belong. The description contains the literal text that will be printed to the console when prompting the user for a variable needed in the diagnosis or repair recommendation.

Table 2: Variables List, Descriptions, and Node Mapping by Subsystem

Variable Name	Type	Description / Meaning	Node(s)	Subsystem
has_issue	string	Is there an issue with the vehicle? (y/n)	1	Initial Position
is_starting	string	Does the car start? (y/n)	3	Vehicle Failure to Start Diagnosis
has_fuel	string	Is there enough fuel? (y/n)	4	Vehicle Failure to Start Diagnosis
has_voltage	string	Does teh battery have enough voltage? (y/n)	5	Vehicle Failure to Start Diagnosis
is_ignition_coil_damaged	string	Is the ignition coil in good condition? (y/n)	6	Vehicle Failure to Start Diagnosis
is_distributor_cap_damaged	string	Is the distributor cap cracked or broken? (y/n)	7	Vehicle Failure to Start Diagnosis
is_timing_belt_damaged	string	Is the timing belt damaged? (y/n)	8	Vehicle Failure to Start Diagnosis
is_making_noise	string	Is the car making an unusual noise? (y/n)	14	General - Vehicle Noise Diagnosis
is_noisy_while_driving	string	Does the noise occur when driving? (y/n)	15	General - Vehicle Noise Diagnosis
is_ticking_noise	string	Is the car making a ticking noise? (y/n)	16	General - Vehicle Noise Diagnosis
is_hiccup_noise	string	Is the car making a rattly noise like a hiccup? (y/n)	17	General - Vehicle Noise Diagnosis
is_air_filter_dirty	string	Is the air filter dirty? (y/n)	18	General - Vehicle Noise Diagnosis
is_exhaust_blocked	string	Is anything blocking the exhaust? (y/n)	19	General - Vehicle Noise Diagnosis
are_wheel_bearings_damaged	string	Are wheel bearings damaged/loose? (y/n)	22	General - Vehicle Noise Diagnosis
are_tires_bald	string	Are tires beyond service life (tread depth less than 1/32in)? (y/n)	23	General - Vehicle Noise Diagnosis
is_truck	string	Is the vehicle a truck? (y/n)	24	General - Vehicle Noise Diagnosis
has_items_in_truck_bed	string	Are there items in the truck bed? (y/n)	28	General - Vehicle Noise Diagnosis
has_items_on_roof	string	Does the vehicle have any items latched to the roof? (y/n)	25	General - Vehicle Noise Diagnosis

is_overheating	string	Does the temperature gauge show as overheating? (y/n)	31	General - Vehicle Overheating Diagnosis
has_coolant	string	Is there sufficient coolant in the engine? (y/n)	32	General - Vehicle Overheating Diagnosis
is_water_pump_broken	string	Is the water pump broken? (y/n)	33	General - Vehicle Overheating Diagnosis
is_oil_low_or_dirty	string	Is the oil level low or dirty (black in color)? (y/n)	34	General - Vehicle Overheating Diagnosis
has_nonfunctional_electronics	string	Are any electronics not functional? (y/n)	38	General - Electrical Issues Diagnosis
does_ac_power_on	string	Does the AC power on? (y/n)	39	General - Electrical Issues Diagnosis
does_ac_blow_cold	string	Does the AC blow cold? (y/n)	40	General - Electrical Issues Diagnosis
has_nonfunctional_headlights	string	Are any headlights or lights not turning on? (y/n)	41	General - Electrical Issues Diagnosis
is_ac_fuse_intact	string	Is the AC fuse intact? (y/n)	44	General - Electrical Issues Diagnosis
are_ac_wires_connected	string	Are AC ground and power wires connected? (y/n)	53	General - Electrical Issues Diagnosis
are_therm_settings_correct	string	Are the thermostat settings correct? (y/n)	47	General - Electrical Issues Diagnosis
is_evaporator_coil_frozen	string	Is the evaporator coil frozen? (y/n)	56	General - Electrical Issues Diagnosis
is_air_filter_dirty	string	Is the air filter dirty? (y/n)	62	General - Electrical Issues Diagnosis
is_nonfunct_light_fuse_intact	string	Is the respective non-working light fuse intact? (y/n)	49	General - Electrical Issues Diagnosis
are_nonfunct_light_wires_conn	string	Are ground and power wires connected? (y/n)	58	General - Electrical Issues Diagnosis
has_burning_plastic_smell	string	Do you smell any burning plastic / electric insulation? (y/n)	42	General - Electrical Issues Diagnosis
is_radio_working	string	Does the radio power on? (y/n)	43	General - Electrical Issues Diagnosis
is_radio_fuse_intact	string	Is the radio fuse intact? (y/n)	51	General - Electrical Issues Diagnosis
are_radio_wires_connected	string	Are radio ground and power wires connected? (y/n)	59	General - Electrical Issues Diagnosis
does_wheel_turn	string	Does the wheel turn stiffly or fail to turn? (y/n)	64	General - Power Steering Issues Diagnosis
is_power_steering_fuse_intact	string	Is the power steering fuse intact? (y/n)	65	General - Power Steering Issues Diagnosis
has_power_steering_fluid	string	Is there enough power steering fluid? (y/n)	66	General - Power Steering Issues Diagnosis
are_tires_deflated	string	Are any of the tires visibly deflated or is the low pressure light on? (y/n)	69	General - Tire Issues Diagnosis

has_piercing_object	string	Is there a sharp item (e.g., nail) stuck in the deflated tire? (y/n)	70	General - Tire Issues Diagnosis
is_obj_inch_away_from_edge	string	Is the piercing object more than 1in away from the tire edge? (y/n)	73	General - Tire Issues Diagnosis
is_recent_temp_change	string	Was there a recent temperature change? (y/n)	71	General - Tire Issues Diagnosis
is_exterior_damaged	string	Is the exterior of the vehicle damaged? (y/n)	76	General Issue Diagnosis
is_damage_cosmetic	string	Is the damage cosmetic paint scratch only? (y/n)	78	General Issue Diagnosis
repair	string	What repair should be recommended to the user? These are the decision nodes.	2, 9, 10, 11, 12, 13, 26, 27, 30, 20, 21, 29, 35, 36, 37, 45, 54, 46, 55, 61, 48, 57, 50, 52, 60, 63, 67, 68, 75, 74, 72, 77, 79, 80	Conclusion variable
issue	string	What is the general vehicle issue detected based on user's response?	81,82,83,84,85,86,87,88	Conclusion variable, also used as intermediate variable in premise list

3.5 System Class Diagram

Figure 10 below shows a class diagram for the program. I intentionally do not use a simplified UML diagram but instead opt to present full method signatures, including the arguments the member functions will expect. Each class is reviewed in detail in Section 4.

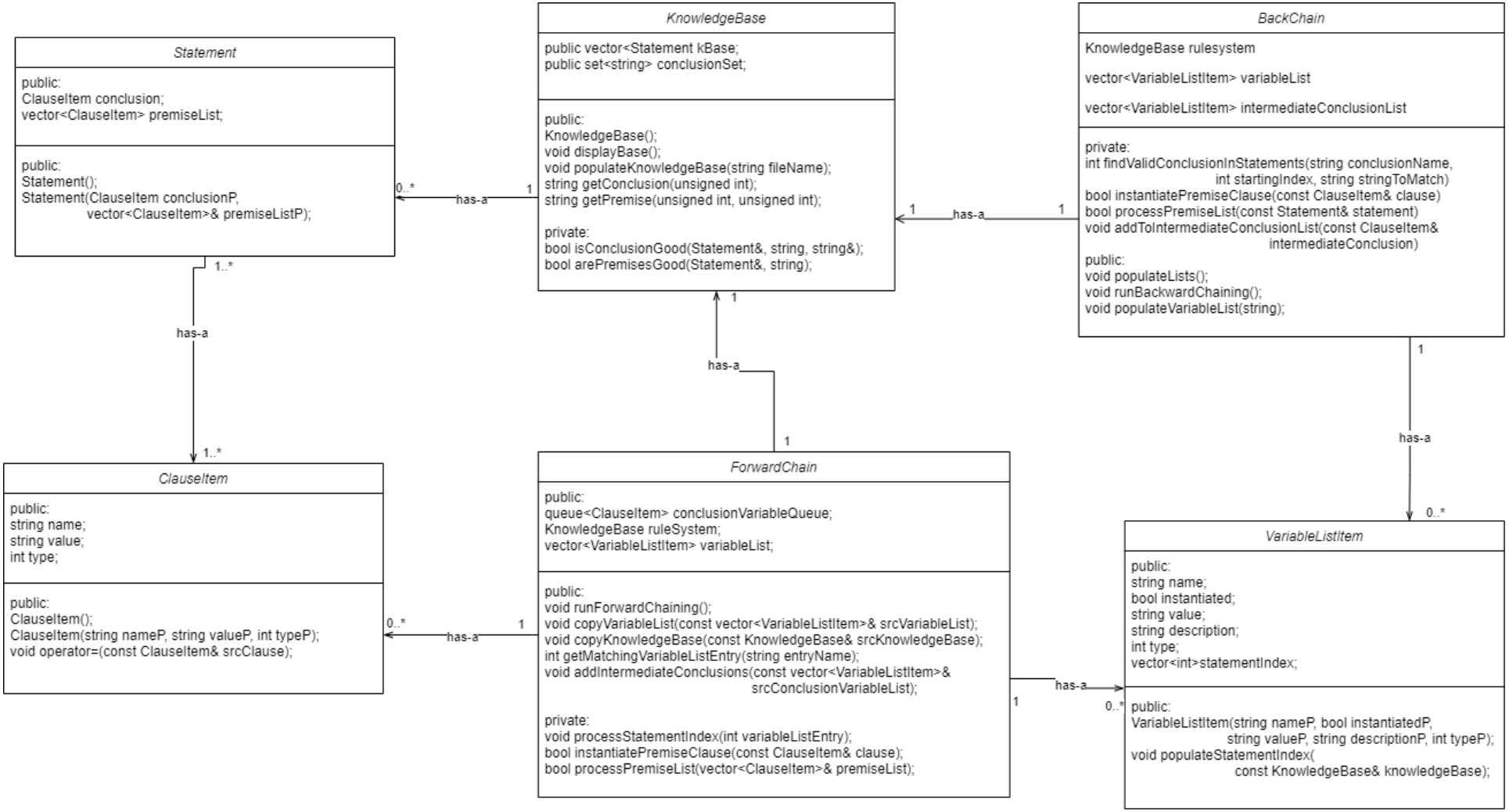

Figure 10: UML Class Diagram for Vehicle Diagnosis and Repair System

3.6 Analysis of Results

3.6.1 How Good Are the Results?

The program is successful in producing correct, repeatable results. It is also robust – it anticipates several edge cases and error scenarios and handles them appropriately. The number of possible repair recommendations in the real world may number in the hundreds of thousands. This system is limited in scope and handles only a few dozen. Thus, a user may exhaust a given branch of the decision tree. The system handles this by treating it as an inconclusive result. This means the user would have to seek help elsewhere. As our expert system is aware of its limitations and signals this to the user when this occurs, we may say the system is complete.

The separation between knowledge base and inference engine in our design also makes the system open ended. It is not only intended for vehicle issue diagnoses or repair recommendations – it may be used with any properly formatted KB file.

If the KB text file is missing or there is a malformed line (e.g., a missing colon token for THEN), the system's error handling capabilities will alert the user and provide recommendations for a solution, as well as direct them to the system documentation for troubleshooting.

I ran the program a total of 42 times – once for each available conclusion. There are 34 repair recommendations and 8 issue diagnoses – in all 42 cases, the system came to the correct conclusion and printed the right output to the console.

3.6.2 Memory and Speed

Disclaimer: only one student had Visual Studio, which contained a memory profiler. Our team collectively used this tool to profile the application's memory usage and take some snapshots. The screenshots below are from a team activity that was collectively performed during a Zoom meeting.

As we will see in Section 4, most loops in the program are O(n) with a few notable exceptions where nested loops exist. The program performs well and there are no noticeable slow-downs or performance issues. Loops were additionally optimized by adding a Boolean flag, so the loop may exit when, for example, we determine a conclusion is not valid. We see an example of this in BackChain.cpp (please refer to Appendix B for complete source code or Section 4.5 for an overview of this class). The for-loop in method findValidConclusionStatements() goes through the knowledge base, seeking a matching conclusion in all of the statements. An internal call is made to processPremiseList(), which loops through the list of premises. If the type is a conclusion but is not a valid value that we are seeking, isValid is set to false. The function returns false, setting isValid to false inside findValidConclusionStatements(), thus terminating the run, saving us from the need to continue parsing the remaining knowledge base.

Figures 11, 12, and 13 below show a captured recording of the program memory profile using Visual Studio 2019.

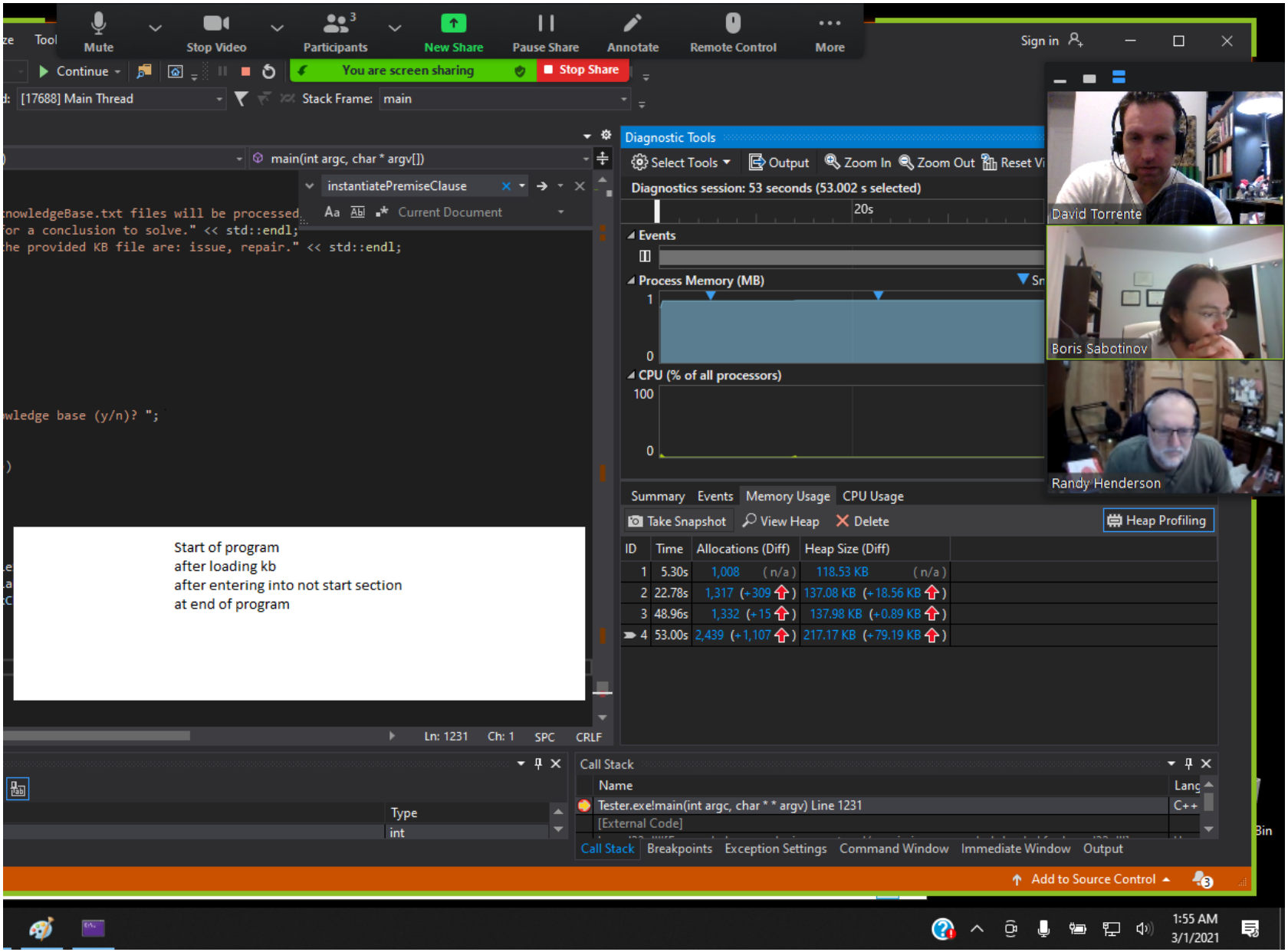

Figure 11: Memory profile of a vehicle failure to start diagnosis

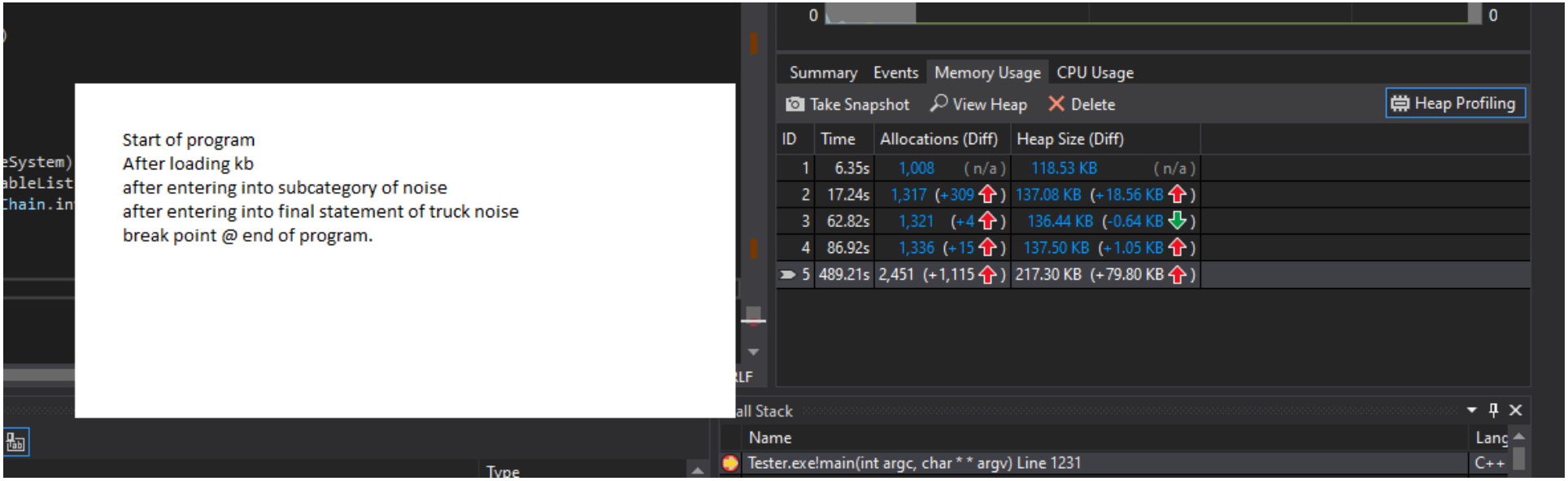

Figure 12: Memory Profile during a noise diagnosis

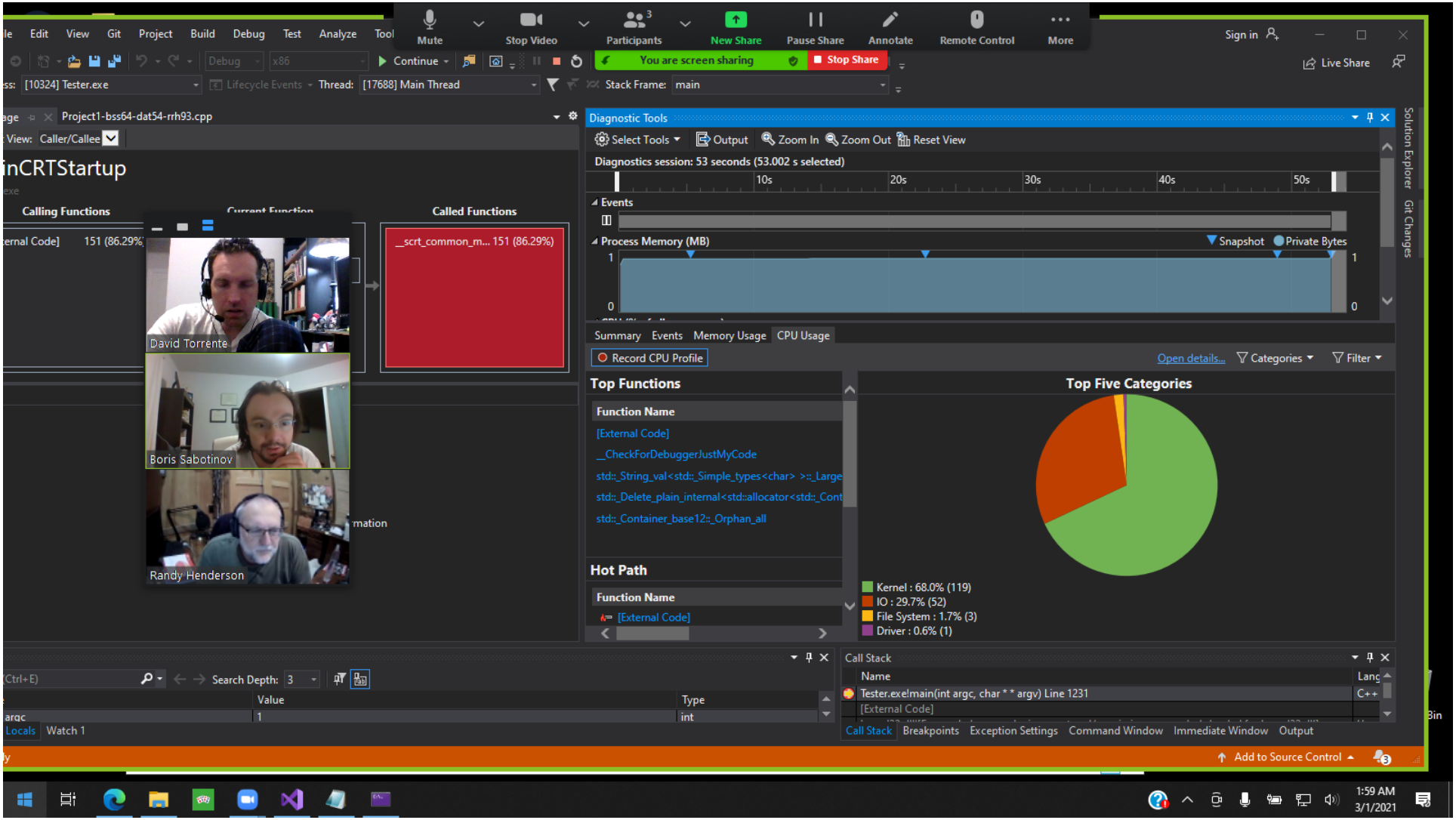

Figure 13: Kernel, I/O, and File System usage during memory profiling

3.6.3 Changes Made and Optimizations

Our team redesigned the program from the ground up, using object-oriented design principles. We kept the same algorithms, so both programs make use of backward and forward chaining. Aside from this, the two are very different programs and there is not much way to compare them in a one-to-one ratio. We completely decoupled the knowledge base from the inference engine, going as far as removing any hard-coded elements and creating parsers to process in a knowledge base text file and an accompanying comma delimited variables list. All global variables are gone, replaced with local class member variables. All goto statements and labels are gone. All switch statements are gone as well. All statements are dynamically read in from a knowledge base text file and are not hardcoded in the source code.

A notable improvement is when instantiating variables, there is no need to do this in forward chaining at all (i.e., there is no equivalent to the original code's function instantiate()). Variables are first encountered in back chaining and instantiated there. Forward chaining then will simply check if a variable was instantiated or not. And in back chaining, the process of instantiation is O(n) because there is only one loop in function in instantiatePremiseClause(const ClauseItem &clause) and it loops from 1 to variableList.size(), or n. By comparison, the original instantiate function was O(n+m), as there were two while loops to get through.

We recall that these are the two loops in instantiate() function in the original forward chaining program:

```
while ((strcmp(v, varlt[i]) != 0) && (i <= 10)) i=i+1;
```

and

```
while ((strcmp(v, cndvar[i] != 0) && (i <= 10)) i=i+1;
```

While in the current refactored program, we only need the for loop starting on line 194 of BackChain.cpp (refer to Appendix B).

The original program made use of a conclusion stack. Our system replaces this with a recursive call, which reduces the amount of code needed. Additionally, we use vectors in place of arrays. They provide the ability to grow dynamically. We use this to make our system robust and flexible – we cannot know in advance how long the premise clause list will be, for example, or how many statements a user may provide in the knowledge base. We want our system to work for any type and size of knowledge base, so using a dynamic structure such as a vector is ideal for this.

We improved on space complexity by reducing the original design on the knowledge base from three data structures (a knowledge base, conclusion and clause variable lists) to the single KnowledgeBase class. We can then access the statements, and the clause items it consists of, internally without duplication.

4 Classes Deep-Dive

Refer to Appendix B for complete source code. I will refer to file name and line number below. Only implementation CPP files will be considered.

We may say that class BackChain has a VariableListItem and a KnowledgeBase. Class ForwardChain has a VariableListItem, a KnowledgeBase, and ClauseItem (via queue). Class KnowledgeBase has a Statement. And class Statement has a ClauseItem.

4.1 ClauseItem

This class represents the basic building block of a Statement. It may represent either a premise, a conclusion. Some clause items may be a conclusion in one statement and a premise in another. For example, an issue may be a conclusion or an intermediate variable in the premise list, when used to speed up the program. A clause item has a name, a value, and a type. The type is an integer indicator and for this program, only option code 2 for STRING was used.

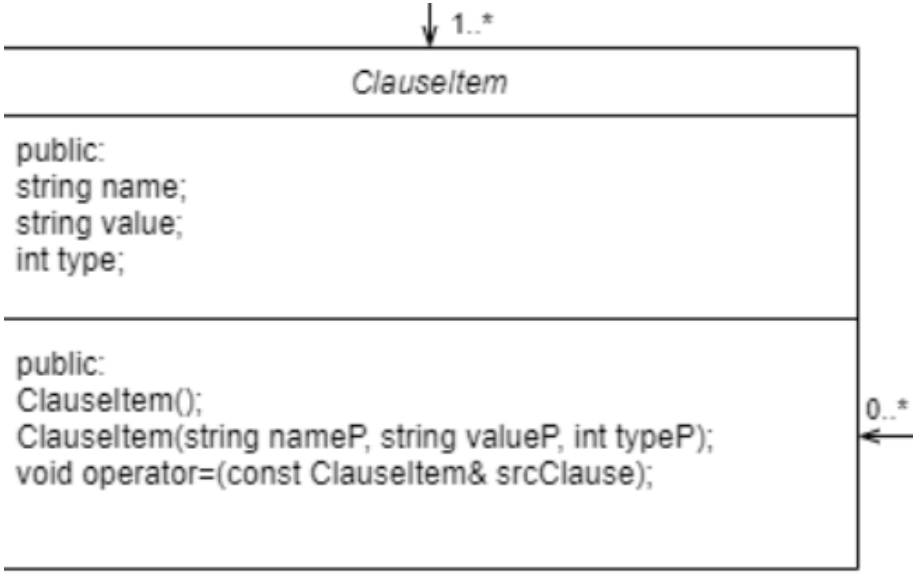

Figure 14: ClauseItem UML Class Node

4.1.1 Data Structures Used

The assignment operator '=' was overloaded, to allow the system to properly copy clause items. There are no special data structures or algorithms employed in this class.

4.2 Statement

Please refer to Section 1.3 for a detailed breakdown of a Statement and the symbols used. A statement is comprised of clause items. The Statement class represents a single line of clause items (premises and a conclusion) in the knowledge base file. Logically, a statement is of the form:

IF variable = <y/n> AND variable_2 = <y/n> THEN conclusion = <value>

Where AND is represented by '^' and THEN is represented by ':'

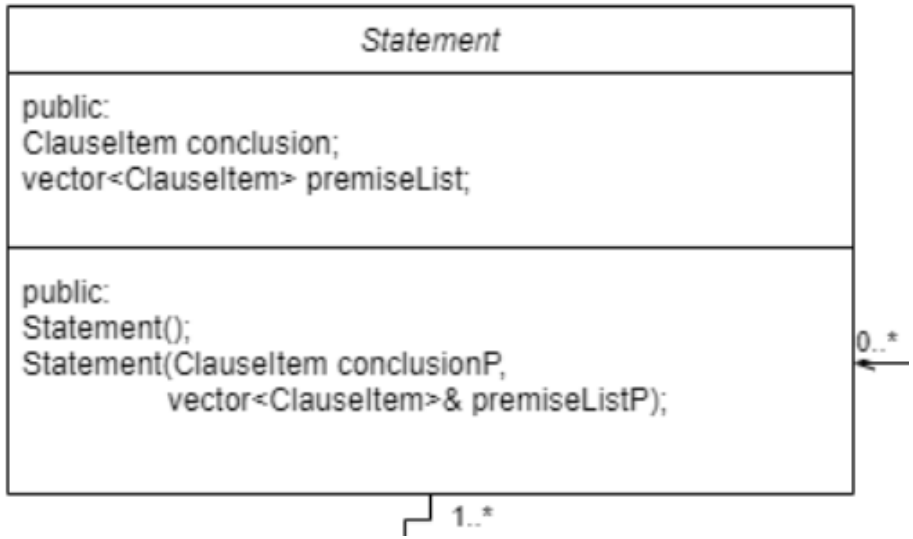

Figure 15: Statement UML Class Node

4.2.1 Data Structures Used

The Statement class uses a vector to keep a list of premises. There are no other special data structures or algorithms employed in this class.

4.3 VariableListItem

The VariableListItem class holds the values of the variables of the knowledge base. Put simply, the prompts the user sees when interacting with the system are internally represented using this class. Refer to Section 3.4 for the complete list of all variables and their values.

Recall that a variable has a name, a type (represented as an integer, in our program all variables are type 2 – STRING), and a value. The VariableListItem also keeps track if a variable is instantiated and maintains its description. The description is the textual prompt (i.e., the question) the user sees when the program asks them to provide the value of a variable. For example, when the user is asked "Does the temperature gauge show as overheating? (y/n)" they are being presented with the description of a VariableListItem. The user's response is captured in the **value** member variable at runtime. This value is used by both forward and backward chaining.

Initially, a VariableListItem is initialized with "false" for instantiated and an empty string for the value.

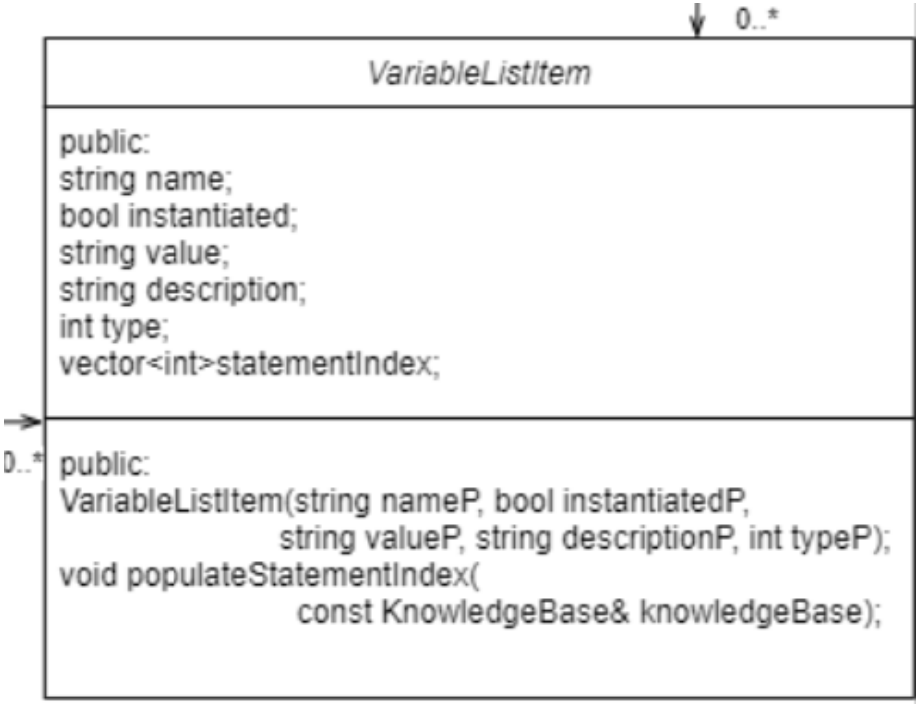

Figure 16: VariableListItem UML Class Node

4.3.1 Data Structures and Algorithms

Function populateStatementIndex(const KnowledgeBase& knowledgeBase) contains two nested for-loops. Runtime here is **O(n^m)**, where n is the size of the knowledge base in the outer loop and m is the size of the premise list in the inner loop.

It is important to note that populateStatementIndex() essentially creates an inverted index, which is kept in the statementIndex member variable. An inverted index is simply a swapping of the direction we take to look up information in a traditional index. Instead of going from a position to the content, here we start with the content and find the location. This allows for quicker searching – as the variables are initialized during back chaining, we can use these values to limit how many statements we need to search through before finding a conclusion.

There are some notable differences in how a list of VariableItemList instances is maintained between forward and backward chaining.

Let us use the following two statements in this example:

1. has_issue = y ^ is_starting = n : issue = Failure to Start
2. issue = Failure to Start ^ has_fuel = n : repair = Insufficient Fuel, Add more fuel.

What would our list of variables be in Backward Chaining? We would only have premises, not conclusions, thus our list will contain **has_issue**, **is_starting**, and **has_fuel**. It will not contain either the **issue** or **repair** variables. Whereas in forward chaining, we add in issue after backward chaining completes, giving us **has_issue**, **is_starting**, **issue**, and **has_fuel** in our list.

4.4 KnowledgeBase

This class provides an internal, dynamic image of the system's knowledge base. It is worth noting that the program is flexible enough to handle any valid data set in knowledgeBase.txt. it does not have to be vehicle related. The system's backward and forward chaining algorithms (i.e., the inference engine) are completely decoupled from the knowledge base. As such, the data can be about anything, not just vehicle repairs.

Class KnowledgeBase parses the KB text file in method populateKnowledgeBase(string fileName). Vector kBase, which consists of a series of statements, is essentially acting as our in-memory database and contains all the statements in the KB file. We also maintain a set in this class, which keeps a list of unique conclusions available to the system. This list is used in BackChain.cpp – please refer to Section 4.5 for details.

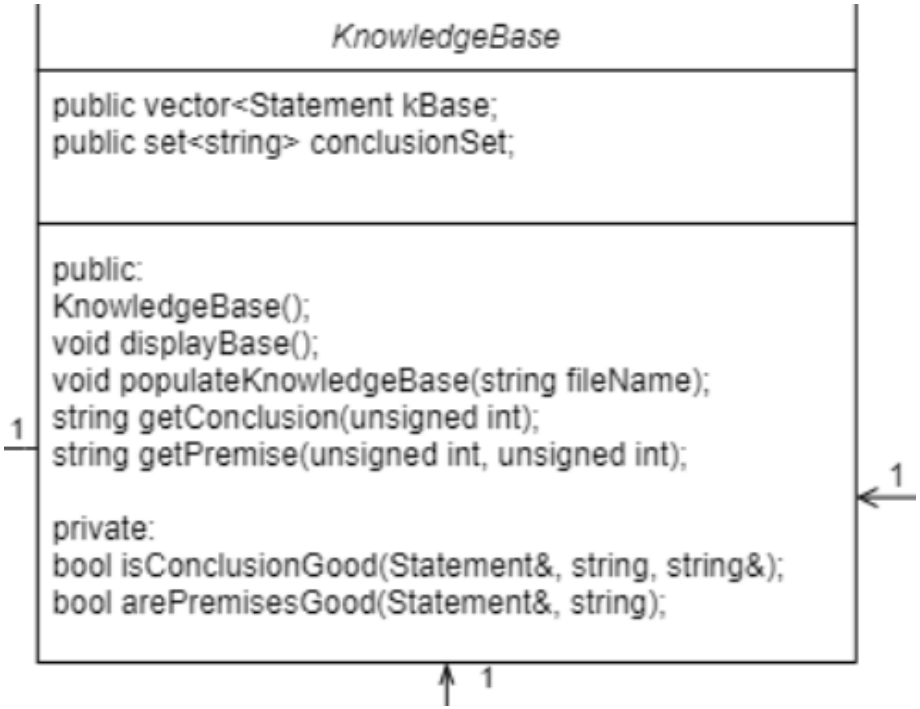

Figure 17: KnowledgeBase UML Class Node

4.4.1 Data Structures and Algorithms

In populateKnowledgeBase(string fileName) function we loop through the entire KB file via the while loop for O(n) runtime.

Function arePremisesGood(Statement& lList, std::string listPremise) uses a do-while loop to go through the entire premise list for a single statement. While it is of course much shorter than a knowledge base consisting of multiple statements, it too is O(n) as we let n be the size of the premise list.

In displayBase() we see two nested while loops, giving us a runtime of O(n^m), where n is the size of the knowledge base and m is the size of the premise list.

It is also interesting to note that KnowledgeBase maintains a set data structure to keep track of a unique (i.e., non-duplicate) list of available conclusions. The program is designed to work with any knowledge base data – medical, airline, vehicle repair – so it is flexible and dynamic. As such, we cannot know in advance the number (or names) of conclusion variables. We must determine this at run time, as we are parsing the KB text file. Thus, in function isConclusionGood(), we insert the conclusion name in our set near the end of the function on line 132 (refer to Appendix B).

4.5 Back Chain

Back chaining here makes use of depth-first search, has a composition relationship (HAS-A) with class KnowledgeBase, and maintains two lists as vectors. The first is a variable list, the second maintains a list of intermediate conclusion variables. That is, variables which are conclusions in one statement, and a premise in another. In our system, the "issue" variable is the only variable which is both a conclusion and an intermediate variable.

This variable is used to speed up processing and simplify the knowledge base and statements (so they are not as verbose). Instead of keeping track of N variables it took to get to one conclusion, we simply use the one conclusion in place of those variables in the statement.

For example:

1. has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = y : issue = Electronics Issue
2. issue = Electronics Issue ^ does_ac_power_on = n ^ is_ac_fuse_intact = n : repair = Replace defective AC fuse

Note how we do not repeat all the variables in statement #1, we simply make use of the issue conclusion as an intermediary.

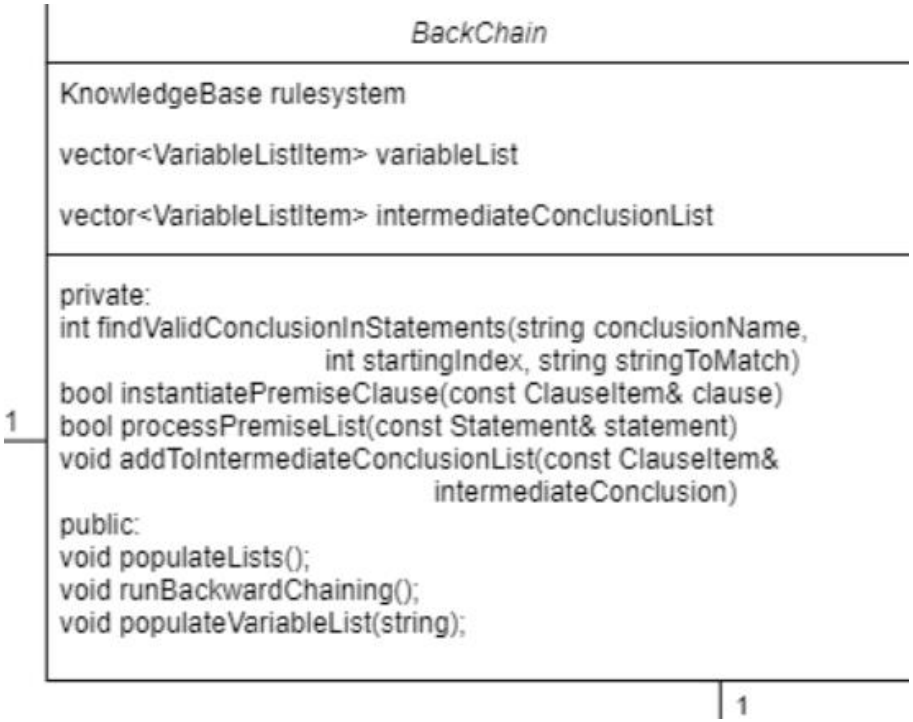

Figure 18: BackChain UML Class Node

4.5.1 Data Structures and Algorithms

In function populateVariableList(string filename), everything up to the while loop on line 57 is O(1) as it is a series of single statements. The while loop - while (getline(variableListFile, csvLine)) – is O(n). Why? Because we parse the file from 0 to n possible lines in the file.

In the function instantiatePremiseClause(const ClauseItem& clause), the for loop is also O(n) – we loop from premise clause = 1 to a worst case possible of variableList.size(). So if the list has n elements, we would loop from 1 to n. It is the same in function findValidConclusionStatements() – the single for loop in this function is O(n) as we loop until a worst case of ruleSystem.kBase.size() or the entire knowledge base.

Class BackChain maintains a vector containing a list of variables. The design of the system is such that variables are instantiated only once, in BackChain, and this is used to populate the forward chain to keep track of conclusions that were set during diagnosis. This reduces space used and improves performance, as ForwardChain may leverage work already performed in BackChain.

Recall that BackChain uses a depth-first search approach, so it will exhaust a branch all the way before trying a different one.

If our tree is A leads to B and C, B leads to D, C also leads to D and the user wants to find a valid conclusion for A, we will try ABD then ACD until it finds a valid conclusion match.

Simple example of a car failing to start:

1. User enters “issue” as a conclusion to solve
2. First match we find in KB is line #1
    a. has_issue = n : issue = No issue
3. We prompt the user with the question associated with variable *has_issue*. Variable is instantiated and user response – Yes – is captured.
4. There is an issue, so this conclusion is not valid, we keep searching and find a match on line #3:
    a. has_issue = y ^ is_starting = n : issue = Failure to Start
5. Going through the premise list, we skip *has_issue* as we see instantiated is true, so we do not need to ask again. We present the prompt for *is_starting* – user answers No.
6. The premise list is satisfied as all variables match, the issue = Failure to Start conclusion is valid. Thus, BackChaining diagnosed the *issue* as a failure to start.

4.6 ForwardChain

ForwardChain is in many ways akin to BackChain. For example, they both have a composition relationship (HAS-A) with class Knowledge base. But how does ForwardChain obtain this KnowledgeBase? It is passed in as a copy via the function copyKnowledgeBase(const KnowledgeBase &srcKnowledgeBase). My personal take on this approach is that it could be improved – instead of copying the KB, we could instead maintain only one static instance of it throughout the entire run of the program, where BackChain and ForwardChain can both access it via reference. This would be a good candidate for future improvement.

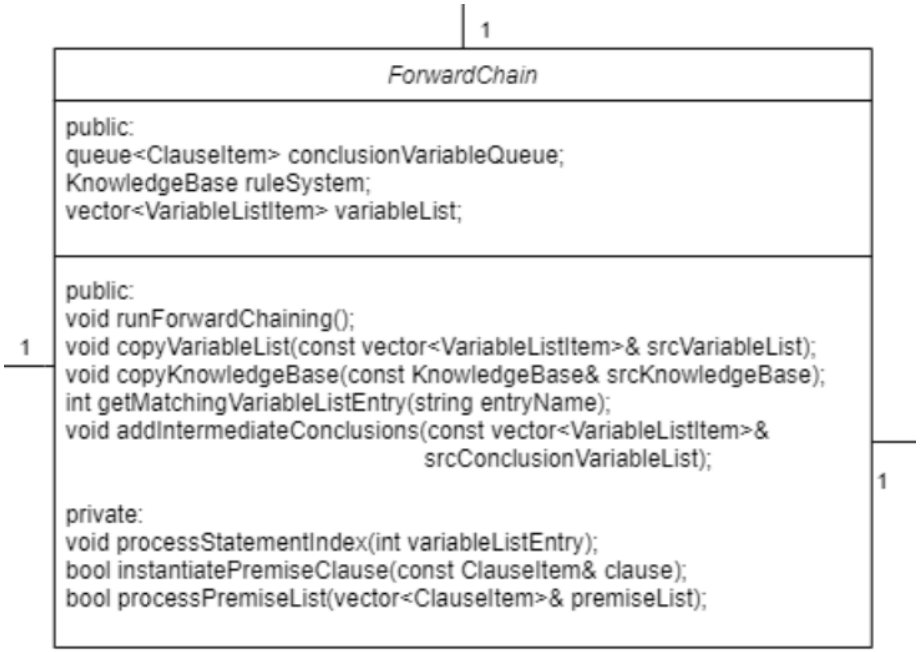

Figure 19: ForwardChain UML Class Node

4.6.1 Data Structures and Algorithms

Forward chaining uses the intermediate conclusion list (e.g., “issue”), which we keep track of in BackChain.

Recall also that VariableItemsList uses an inverted index in populateStatementIndex(); forward chaining uses one as well in processStatementIndex() function on line 138. We use this index to track which statement include a needed clause – this speeds up the program by eliminating unnecessary searching.

As forward chaining needs all premise clauses to be in the clause variable list, we use function addIntermediateConclusions() to add the list of intermediate conclusions we created in BackChain to the clause variable list. This function has a for-loop, which iterates from 0 to srcConclusionVariableList.size(), thus giving us O(n) runtime. The loops in functions copyVariableList() and copyKnowledgeBase() are also O(n).

1. has_issue = y ^ is_starting = n : issue = Failure to Start
2. issue = Failure to Start ^ has_fuel = n : repair = Insufficient Fuel, Add more fuel.

Looking at the two statements above, our system does not need to go through *has_issue* or *is_starting* again because those variables led to the conclusion "issue = Failure to Start" and this is now in the intermediateConclusionList we start keeping in BackChain, which ForwardChain leverages for improved performance.

So, what happens when a user wants a repair recommendation? Simply put, this is how ForwardChain determines a repair recommendation:

1. Ask the user what conclusion to solve, they choose "repair"
2. Look at the conclusionList for a match of "repair"
3. Next, look through the premiseList and ask the user the questions associated with the variables in the premiseList
    a. Note: All variables encountered is instantiated and initialized with the user's response in BackChain
    b. Note: We keep track of intermediate conclusions (if they exist) in the intermediateConclusionsList in BackChain
4. If the user's response matches the value of the variable in the premise list, then the conclusion is valid
5. Once we run BackChaining and obtain a diagnosis, ForwardChain copies the knowledge base, variables list, and adds in the intermediate conclusions to this list from the intermediateConclusionList we maintained in BackChain
6. We use intermediate variables to save time and find a matching repair conclusion

A simple example would be a vehicle not starting because it does not have fuel:

1. User selects "repair" for conclusion variable
2. System finds a match on statement #2, the first time we encounter repair
    a. issue = No issue : repair = No Repair Required
3. We look through the premiseList and find issue, issue is an intermediate conclusion. We look for issue and find a match on statement #1:
    a. has_issue = n : issue = No issue
4. We look at the premise list, see variable has_issue and display the description/question associated with it.
5. User responds, we instantiate the variable (no need to ever ask again) and save the response
6. User answers 'n' for no/false, so the conclusion is not valid. We look for another match, find it on statement #3:
    a. has_issue = y ^ is_starting = n : issue = Failure to Start
7. Recall has issue is already instantiated, no need to ask twice. What's left in the premise list of this statement is "is_starting" – we display the question associated with this variable. User answers 'n' for No, variable is instantiated and response captured. Because has_issue is y and is_starting is n, which are the expected values (user response matches knowledge base values), the conclusion is valid, so issue is Failure to Start.
8. BackChaining is done, we diagnosed the issue and found a valid conclusion. ForwardChaining now copies the KB, variableList, and adds in intermediate conclusions to the list

9. We search now for repair, find a match on statement #2 but the value of issue we diagnosed is “Failure to Start” and here it’s “No issue”
10. We continue, we find a match on statement #4 of the KB:
    a. issue = Failure to Start ^ has_fuel = n : repair = Insufficient Fuel, Add more fuel.
11. Issue is instantiated, recall we added it to the variableList from the intermediateConclusionList. The value matches “Failure to Start” and the value for has_fuel matches as well. Thus, for the conclusion the user entered to solve – repair – and for the values of the variables the user provided, we find a match
12. The issue is Failure to Start and the repair recommendation is the value of the repair conclusion variable on statement #4 in the KB file, which is “Insufficient Fuel, Add More fuel.” ForwardChain has successfully used the diagnosis performed in BackChain to find and recommend a repair to our user!

5 Sample Runs

The program prints the same content to the console at the beginning of each run. Both the welcome message and the CLI messages printed while parsing the knowledge base file do not differ between each subsequent run. A complete listing of this output may be found at the end of this report, under “**Appendix C: Complete Sample Output**.” The program always displays a welcome message to the user, parses the KB text file and prints contents to the console as it does so, and pauses. After the user hits Enter, it asks if the user wants to display the KB in human readable output. As such, the beginning section is only available in Appendix C to avoid repeating hundreds of lines.

Complete output of this program was obtained on eros.cs.txstate.edu. Program was invoked normally but standard output was piped to tee. Tee is a command which reads our standard I/O and writes it to both standard output and a text file, in our case a log file.

This section aims to comprehensively cover all options of the implemented system. While not every available repair recommendation will be shown here, every uniquely available system option will be exercised and presented.

5.1 Sample Run #1: Diagnosing a Tire Issue

In this sample run we impersonate a user experiencing a vehicle that is not operating properly. Our hypothetical user does not know it, but the vehicle has a flat tire. Through a careful application of back chaining, we ask the user questions (when appropriate) to determine what issue their vehicle is experiencing. They are not able to go down the road properly, so they leverage our system for diagnostics. Please note that the welcome message and Knowledge Base parsing command line output is the same for each program execution. Thus, the screenshots capture only the relevant portion of the output for this sample run. For complete program output, please refer to Appendix C.

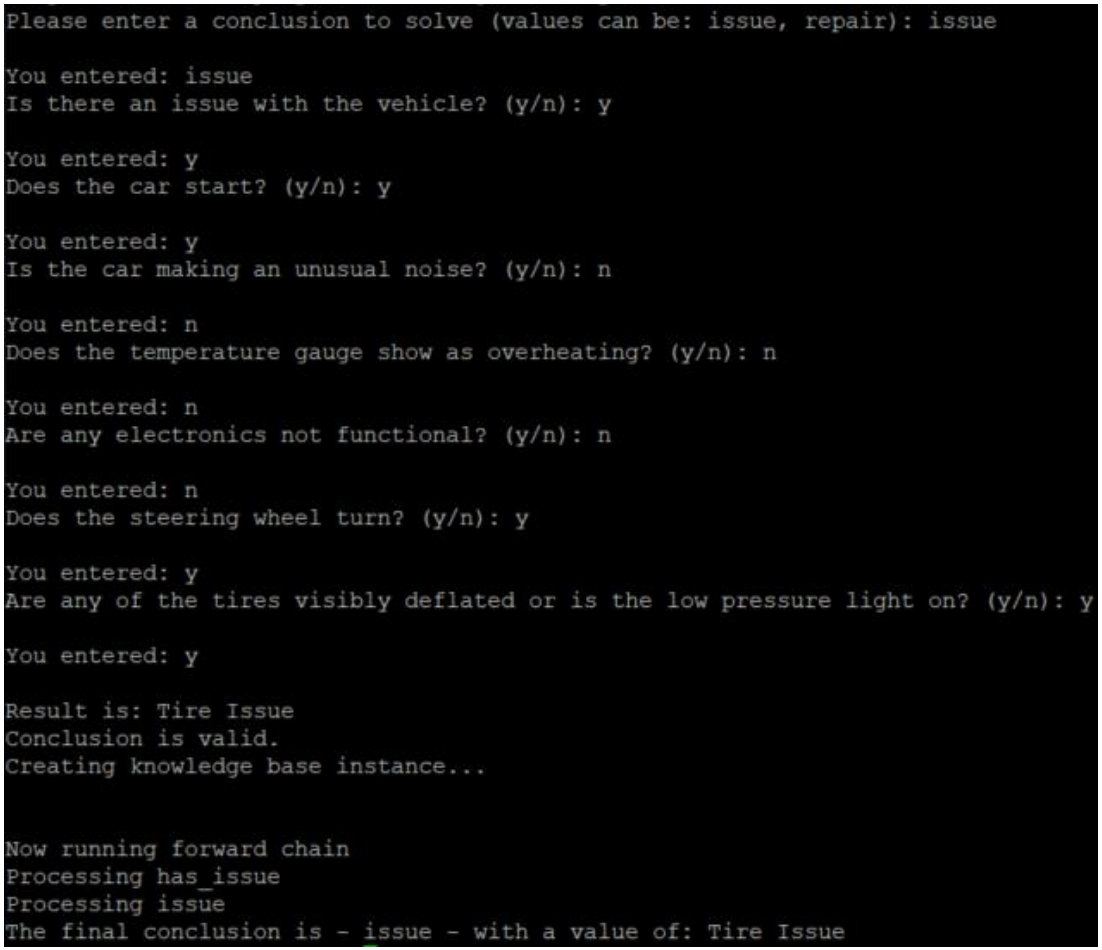

Figure 20: Sample Run #1 - Diagnosing Tire Issue

5.2 Sample Run #2: Diagnosing a Start-up Issue

In this sample run we impersonate a user experiencing a vehicle that is not operating properly. Our hypothetical user is not able to start the engine. In this run, we simply diagnose what the issue is using back chaining.

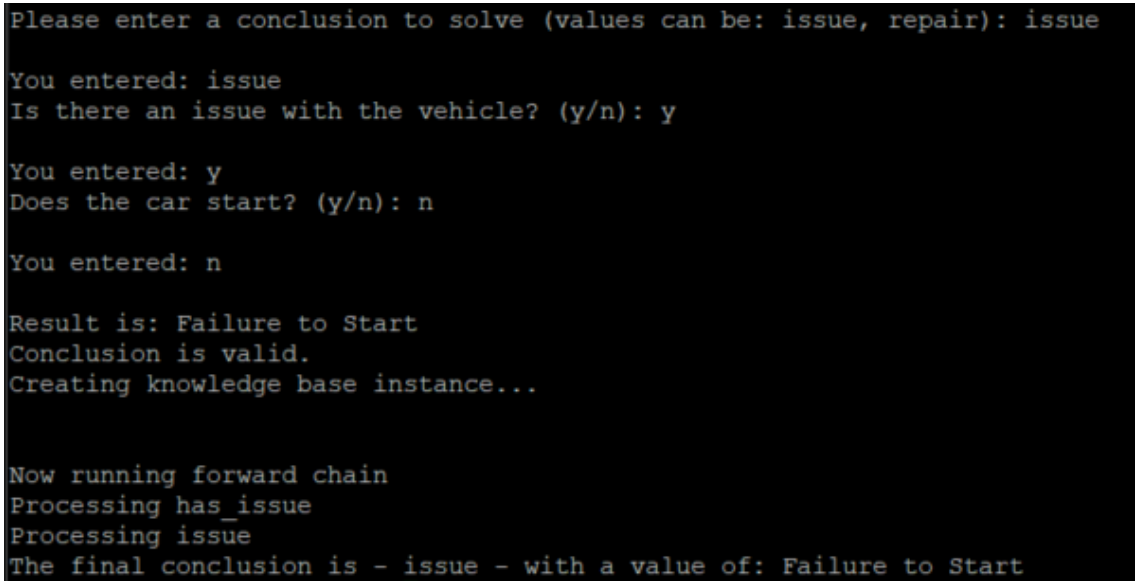

Figure 21: Sample Run #2 – Startup Issue

5.3 Sample Run #3: Repair Recommendation, Air Filter Replacement due to Noise

We now get into solving for a repair conclusion and thus offering our users a repair recommendation. Just as in sample runs one and two, we still perform diagnostics with back chaining. Variables encountered are instantiated. The KB instance is then passed to Forward Chaining, which determines which repair recommendation (if any) to display to the user. In this sample run, we simulate a user experiencing a noise issue. Through a series of questions, we determine the issue is a dirty air filter, clogging up air intake ability and causing an unusual noise in the cabin. The repair recommendation suggests to the user to replace the dirty air filter with a clean one.

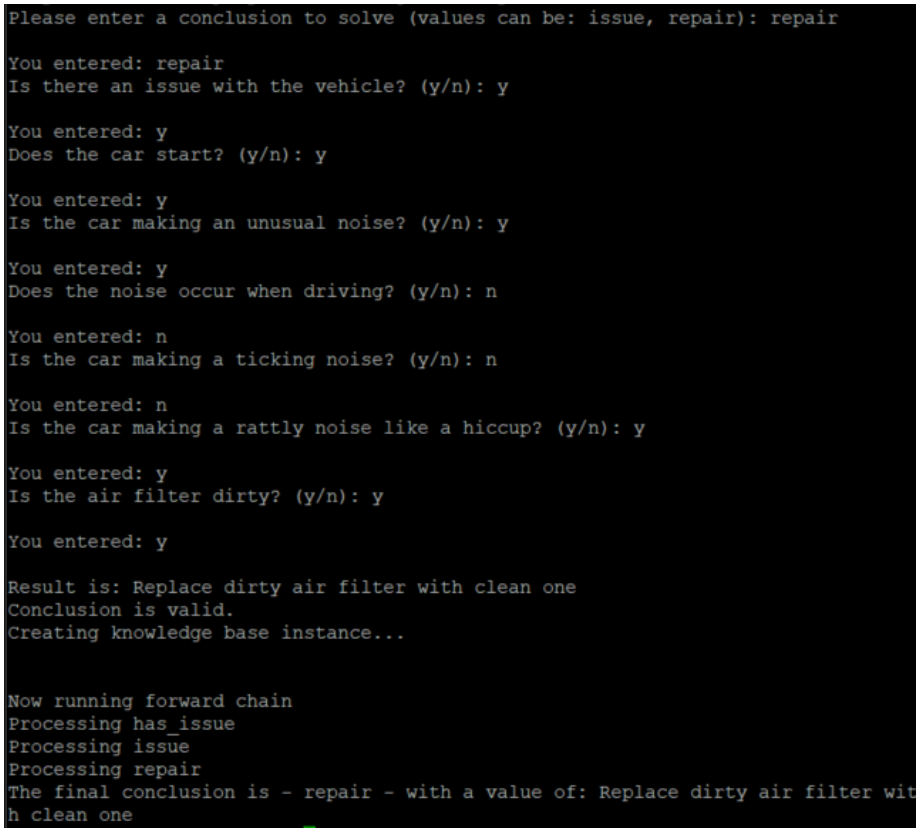

Figure 22: Sample Run #3 – Repair Recommendation, Change Air Filter

5.4 Sample Run #4: Repair Recommendation, Replacing a Defective Water Pump

In this sample run, we have a hypothetical vehicle that is overheating. The user does not know why. By using our system, we lead them to inspect the water pump. The user sees it is broken and reports this to our system, which recommends replacing the defective water pump as a possible repair.

```
Please enter a conclusion to solve (values can be: issue, repair): repair

You entered: repair
Is there an issue with the vehicle? (y/n): y

You entered: y
Does the car start? (y/n): y

You entered: y
Is the car making an unusual noise? (y/n): n

You entered: n
Does the temperature gauge show as overheating? (y/n): y

You entered: y
Is there sufficient coolant in the engine? (y/n): y

You entered: y
Is the water pump broken? (y/n): y

You entered: y

Result is: Defective Water Pump, replace water pump.
Conclusion is valid.
Creating knowledge base instance...


Now running forward chain
Processing has_issue
Processing issue
Processing repair
The final conclusion is - repair - with a value of: Defective Water Pump, replac
e water pump.
```

Figure 23: Sample Run #4 - Replace Defective Water Pump

5.5 Sample Run #5: Printing out the Knowledge Base when prompted

In Appendix C, we see a complete printout of sample run #1. In the beginning of each run, the knowledge base is parsed and each statement's status – successful or failed processing – is reported. However, we decided to provide users the additional capability of printing out the KB, after a successful import, in a more concise and human-readable format. Figure 24 and Appendix C both show this optional output.

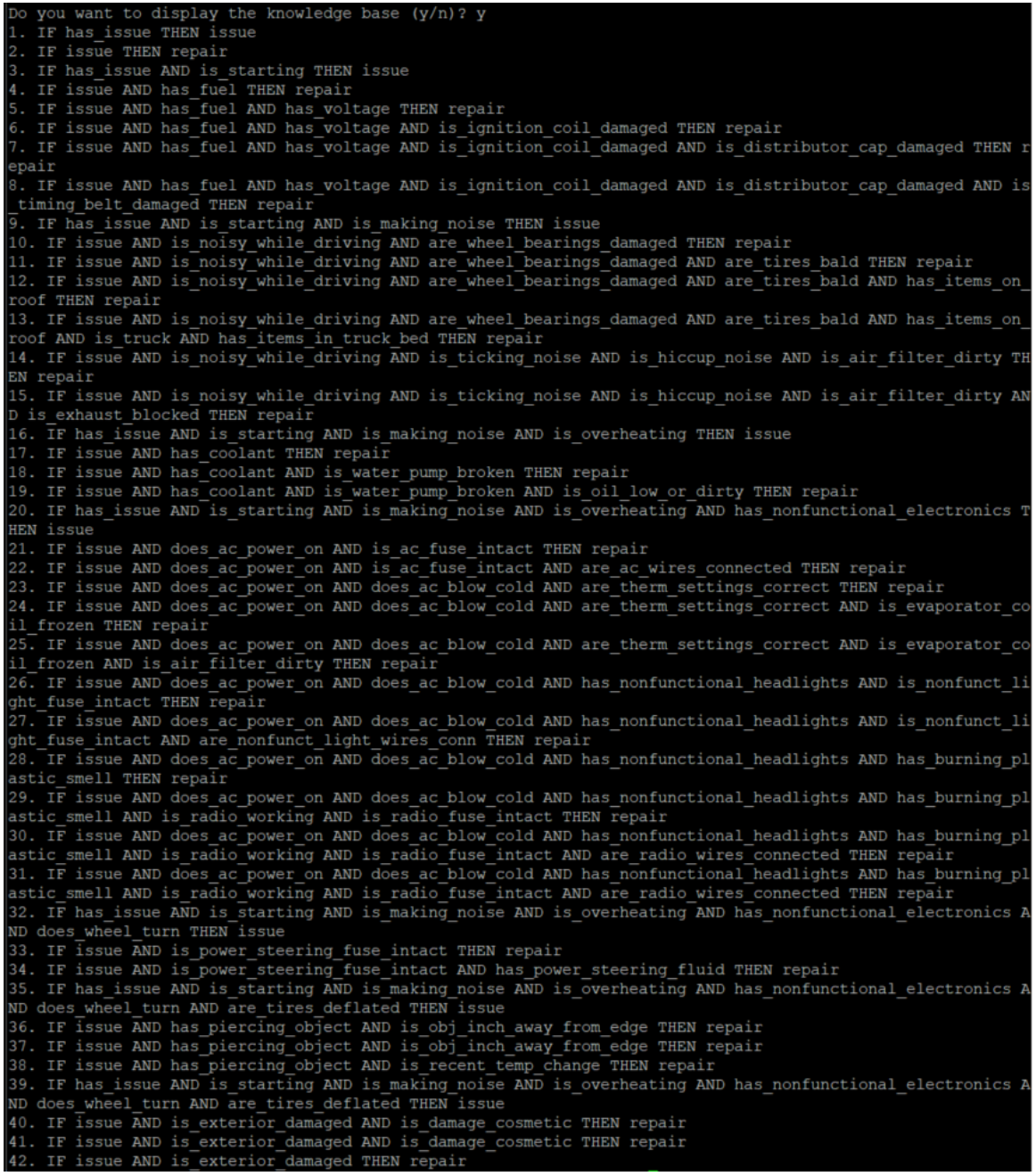

Figure 24: Sample Run #5 - Print Knowledge Base to Console

5.6 Sample Run #6: Printing the Help Menu

The system also offers a help menu should the user invoke the program with the following command: ./Project -h

This is a relatively brief help message, giving the user guidance on how to interact with the program and where to refer to accompanying documentation for more guidance.

```
[bss64@eros project-one]$ ./Project1 -h
Welcome to the Automobile Diagnostic Program.
Authors: David Torrente (dat54@txstate.edu), Randall Henderson (rrh93@txstate.edu), Borislav Sabotinov (bss
64@txstate.edu).

To use this program, please read the instructions below and re-launch.
Additional details for building and execution are also available in the README.md file.

1. variablesList.csv and knowledgeBase.txt files will be processed to create an instance of the knowledge b
ase
2. user will be prompted for a conclusion to solve.
     - Valid choices for the provided KB file are: issue, repair.
```

Figure 25: Sample Run #6 - Print the Help Menu

5.7 Sample Run #7: Intentionally Running with Missing Knowledge Base File to Examine Error Handling

If we tried to execute the program without a knowledge base, we would see the following output:

```
[bss64@eros project-one]$ ./Project1
Welcome to the Automobile Diagnostic Program.
Authors: David Torrente (dat54@txstate.edu), Randall Henderson (rrh93@txstate.edu), Borislav Sabotinov (bss
64@txstate.edu).


Creating knowledge base instance...
terminate called after throwing an instance of 'std::runtime_error'
  what():  Error reading Knowledge Base (KB) file. Please validate it uses the correct format. Invoke appli
cation with -h or -help for details.
Aborted
```

Figure 26: Sample Run #7 - Intentionally Test Error Handling

5.8 Sample Run #8: Exhausting the Options – Inconclusive

If a user exhausts a branch of the Decision Tree (i.e., their answers do not match any available conclusion), the system will mark the conclusion as “Inconclusive” as a valid conclusion was not located. The user may then seek guidance outside of this system.

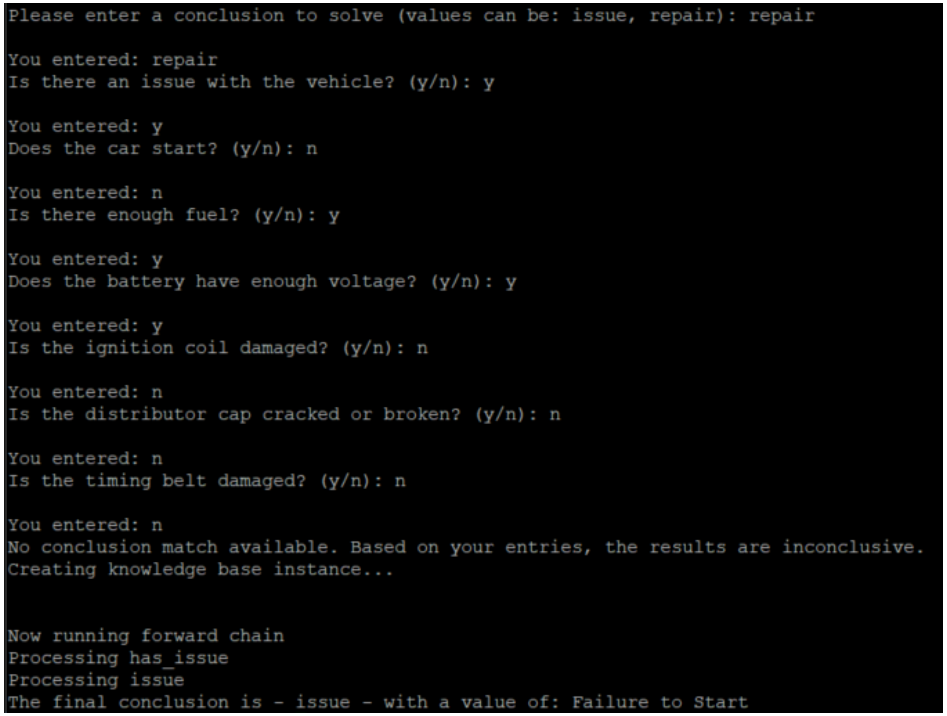

Figure 27: Sample Run #8 - Inconclusive Result

5.9 Sample Run #9: KB file is present but contains defective statements

Now suppose we provide a knowledge base text file, so the file exists. But we make a mistake in the data inside the file.

The system will attempt to parse the file and load all properly formatted statements, which use correct syntax. Statements that are malformed will be counted and skipped over. Figure 28 shows the program output once the defective KB file is processed:

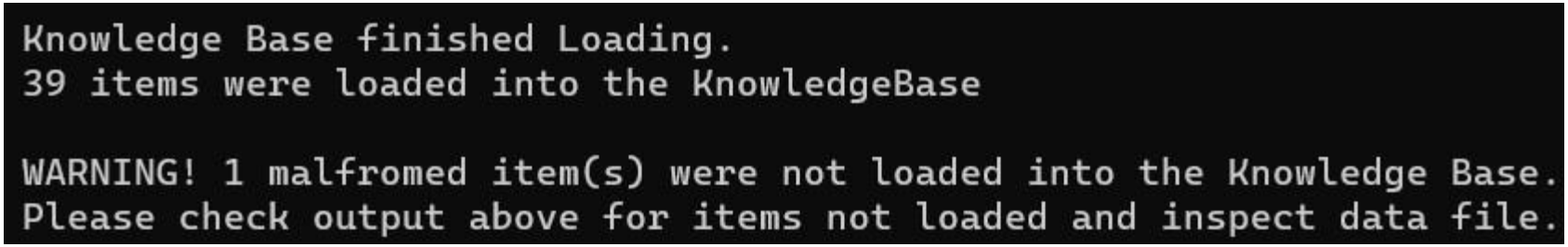

Figure 28: System CLI summary output, after KB file is loaded, if a defective statemetn is found

This alerts the user that something is amiss – they may then scroll up and inspect the detailed knowledge base parsing CLI output (full text in Appendix C). There, they can locate the specific line with the defect. Figure 29 shows a defective statement, which caused the warning in Figure 28. Note that the list was not updated because the conclusion is not formatted correctly. This detailed error handling output helps the user locate the issue, which in this case is the use of a colon instead of an equals sign after repair.

repair: Remove items from roof

Should instead be:

repair = Remove items from roof

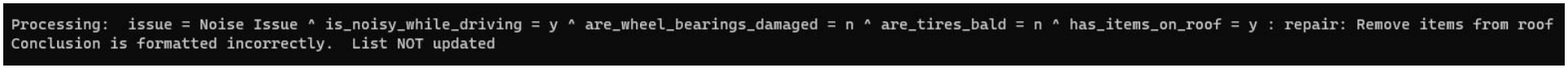

Figure 29: System CLI output example for a defective statement in KB file

6 Conclusion

This was a highly successful project. Despite the challenges with water and power outages and time lost, our team came together to deliver a working solution, which took the original program and made several improvements in the new design. We added numerous features for improved error handling and user experience. Our team was proud to completely separate the knowledge base from the inference engine and create a flexible, multi-purpose solution which can handle any knowledge base. While the provided knowledge base text file with this submission is specifically tailored for diagnosing vehicle issues and recommending repairs, there are no technical limitations preventing anyone from using this program for other purposes (e.g., medical, engineering, or agriculture). This program could, for example, be used to diagnose commercial crop issues and recommend pesticide treatments or other solutions to farmers.

The final solution provided in this submission correctly processes the knowledge base text file, the variables list CSV, and produces correct results for all 42 statements in the knowledge base.

References

1. Gaddis, Tony. "Starting out with C++ From Control Structures through Objects, Ninth Edition." Chapter 10 (c-strings & the string class), Chapter 17.3 the Vector Class.

2. Huntington, Dustin. "Back to Basics –. Backward Chaining: Expert System Fundamentals." http://www.exsys.com/pdf/BackwardChaining.pdf

3. C++03 Standard [2.1.1.2]. https://gcc.gnu.org/legacy-ml/gcc/2001-07/msg01120.html

4. Resources, PPT slides, and course materials on Canvas.

Appendix A: Decision Tree Diagram

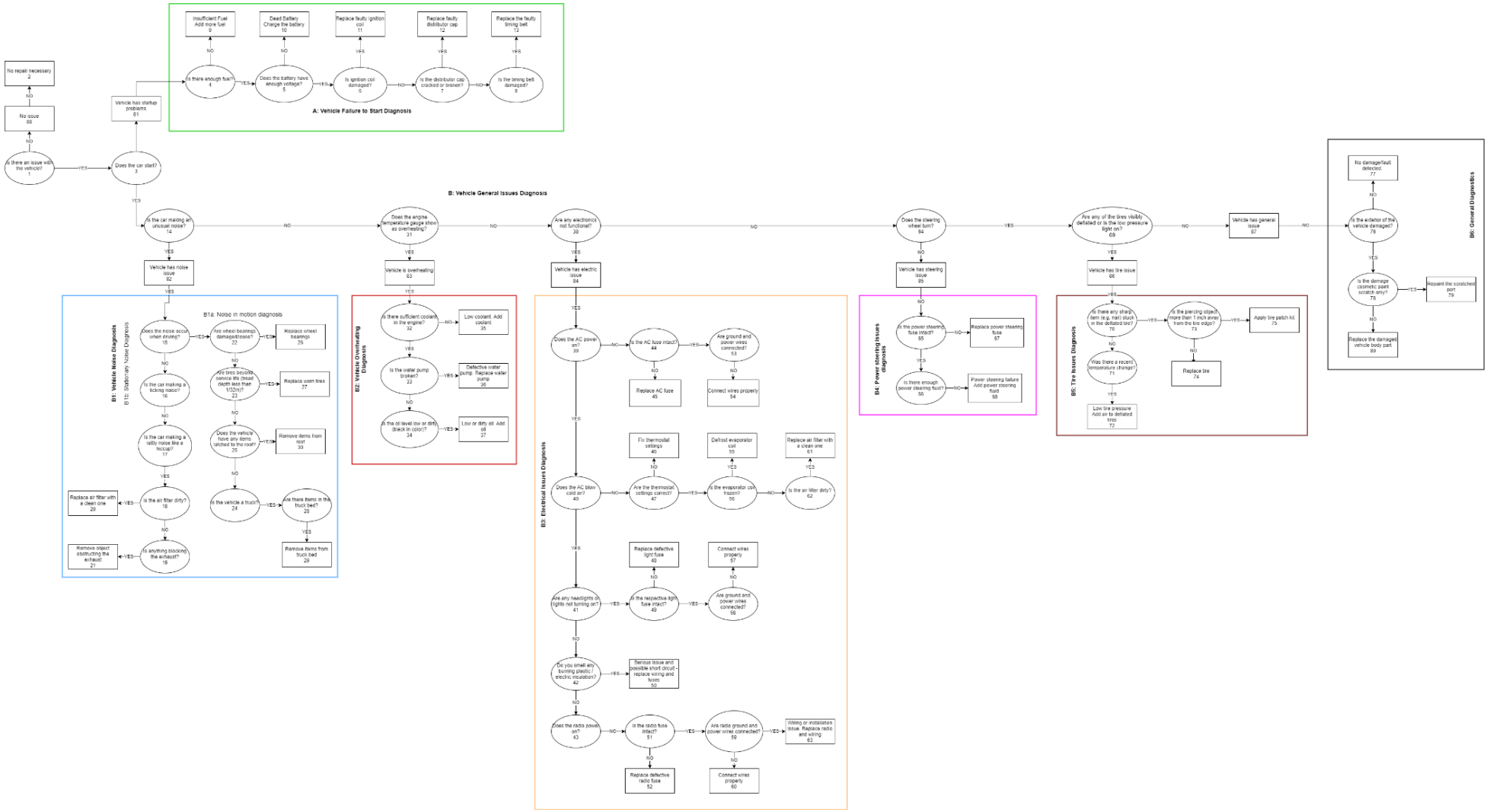

Figure 30: Full Decision Tree Diagram

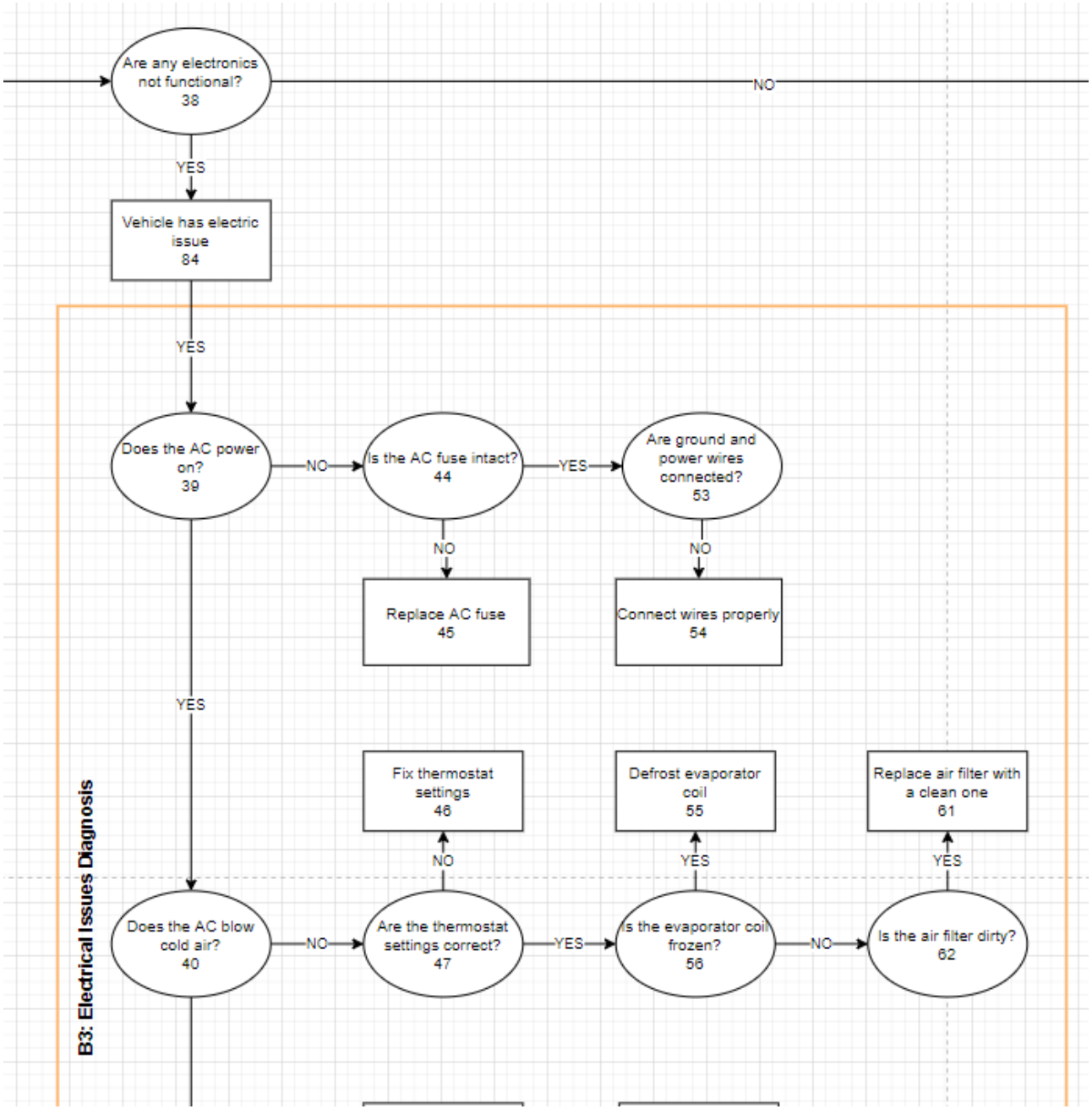

Figure 31: Electrical Issue Diagnosis Subsystem Part 1

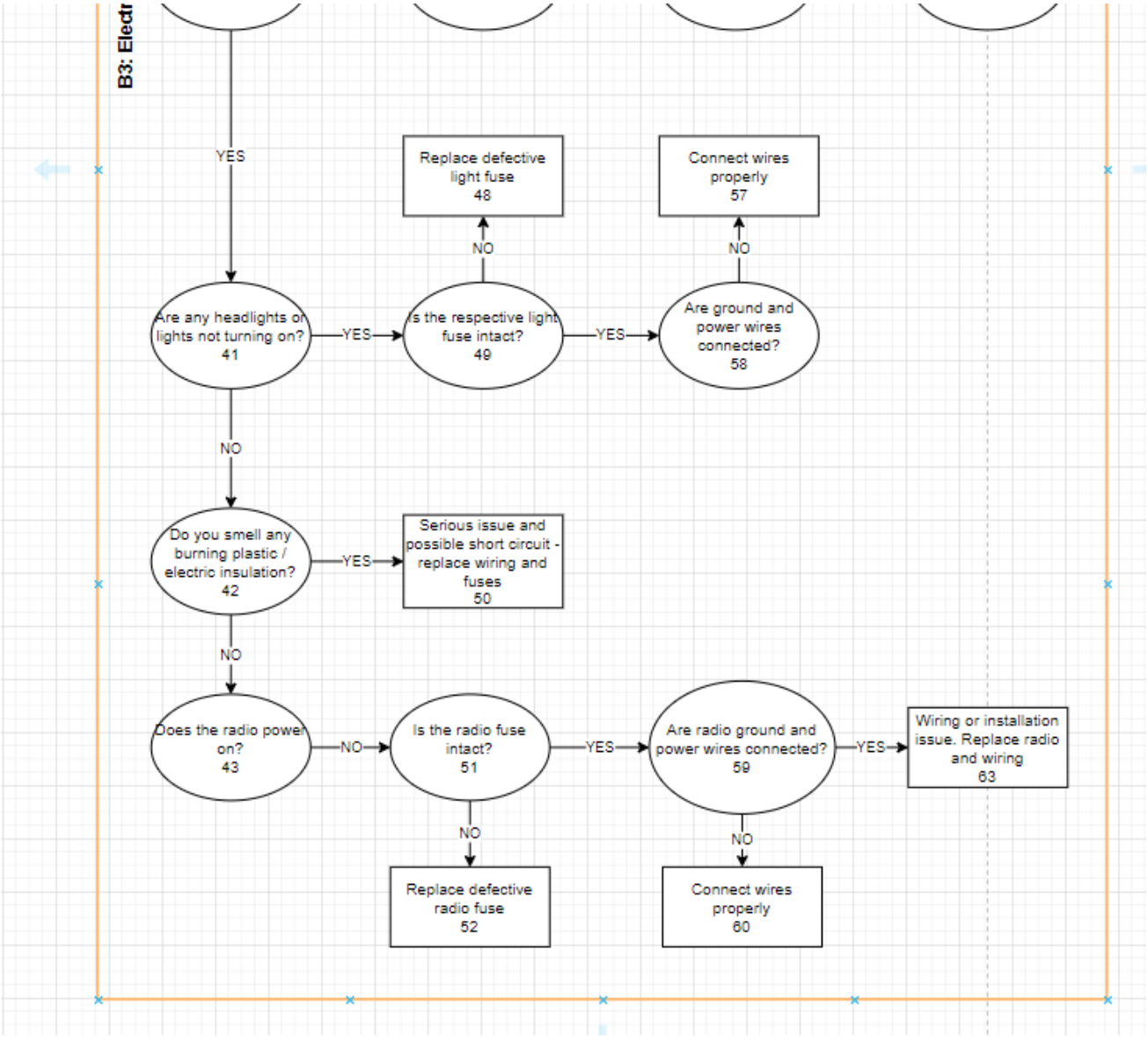

Figure 32: Electrical Issue Subsystem Diagnosis Part 2

Appendix B: Source Code

ClauseItem.hpp

```
1. #ifndef CLAUSE_ITEM_H
2. #define CLAUSE_ITEM_H
3.
4. #include <string>
5.
6. #define INT 1
7. #define STRING 2
8. #define FLOAT 3
9.
10./**
11. * ClauseItem - Represents either a premise or a conclusion (or both).
12. * How a variable is treated depends on the data structure in which it
13. * resides (e.g., premiseList or conclusionList)
14. */
15.class ClauseItem
16.{
17.public:
18.    std::string name;
19.    std::string value;
20.    int type;
21.
22.    ClauseItem();
23.    ClauseItem(std::string nameP, std::string valueP, int typeP);
24.    void operator=(const ClauseItem& srcClause);
25.
26.};
27.
28.#endif // !CLAUSE_ITEM_H
29.
30.
```

ClauseItem.cpp

```
1. #include "ClauseItem.hpp"
2.
3. // These defines are used throughout the program to specify what type of clause
4. // is being processed. The current implementation is all strings. As such, they
5. // are retained only for future development.
6. #define INT 1
7. #define STRING 2
8. #define FLOAT 3
9.
10./**
11. * Constructor | ClauseItem | ClauseItem
12. *
13. * Summary: Instantiates a default clause with NULL values. Note that if the
14. *          knowledge base contains the value of NULL, some care must be taken.
15. *          However, a guard is in place as each index begins at element 1.
16. *
17. */
18.ClauseItem::ClauseItem()
19.{
20.    name = "NULL";
21.    value = "NULL";
22.    type = STRING;
23.}
24.
25./**
26. * Constructor | ClauseItem | ClauseItem
27. *
28. * Summary: Instantiates a clause item with the specified values. Clauses follow
29. *          the pattern of Name = Value. The type parameter is for future use.
30. *          All types are currently set to type string.
31. *
32. * @param string nameP:   The name portion of the clause.
```

Statement.hpp

```
1. #ifndef STATEMENT_H
2. #define STATEMENT_H
3.
4. #include <vector>
5.
6. #include "ClauseItem.hpp"
7.
8. /**
9.  * Represents a single line entry in the knowledge base file.
10. * 0 to n premises and 1 conclusion. A statement may be atomic (e.g., var = true).
11. * Here is an example of a statement (single line in KB file):
12. * {@code issue = Failure to Start ^ has_fuel = y ^ has_voltage = n : repair = Dead Battery, Change the ba
    ttery.}
13. *
14. * Here, we use intermediate variable issue (which may also be a conclusion).
15. * IF issue is equal to "Failure to Start AND (^ indicates logical AND) has_fuel is true AND
16. *    has_voltage is false THEN repair recommendation is "Dead Battery, Change the battery."
17. */
18.class Statement
19.{
20.public:
21.    Statement();
22.    Statement(ClauseItem conclusionP, std::vector<ClauseItem>& premiseListP);
23.    ClauseItem conclusion;
24.    std::vector<ClauseItem> premiseList;
25.};
26.
27.#endif // !STATEMENT_H
28.
29.
```

Statement.cpp

```cpp
#include "Statement.hpp"

/**
 * Constructor | Statement | Statement
 *
 * Summary: Instantiates a default statement with NULL values. Note that if the
 *          knowledge base contains the value of NULL, some care must be taken.
 *          However, a guard is in place as each index begins at element 1.
 *
 */
Statement::Statement()
{
    conclusion.name = "NULL";
    conclusion.value = "NULL";
    conclusion.type = STRING;
    premiseList.push_back(ClauseItem());
}

/**
 * Constructor | ClauseItem | ClauseItem
 *
 * Summary: Instantiates a statement item with the specified values. Statements
 *          follow the pattern of premiseList then conclusion.
 *          All types are currently set to type string.
 *
 * @param ClauseItem conclusionP: The conclusion portion of the statement
 * @param std::vector<ClauseItem>& premiseListP> :    A vector of premise
 *                                  clauses to be used in the statement.
 *
 */
Statement::Statement(ClauseItem conclusionP, std::vector<ClauseItem>& premiseListP)
{
    conclusion = conclusionP;
    premiseList = premiseListP;
```

```
}
```

VariableListItem.hpp

```
#ifndef VARIABLE_LIST_ITEM_H
#define VARIABLE_LIST_ITEM_H

#include <string>
#include <vector>

#include "KnowledgeBase.hpp"

/**
* A list of variables and their associated values that must accompany the knowledgeBase.txt file
* The variablesList.csv file contains a comma delimited list of variables and their contents to display.
* For example:
* {@code is_distributor_cap_damaged,Is the distributor cap cracked or broken? (y/n),STRING}
*
* first value is the variable name, next is the description that will be displayed to the user if we need to obtain th
eir
* input about this variable, and last is the type. Currently all variables are of string type, as we read them in via
text file.
*/
class VariableListItem
{
public:
    VariableListItem(std::string nameP, bool instantiatedP, std::string valueP, std::string descriptionP, int typeP);
    void populateStatementIndex(const KnowledgeBase& knowledgeBase);

    std::string name;
    bool instantiated;
    std::string value;
    std::string description;
    int type;
```

```
29.     std::vector<int>statementIndex;
30. };
31. 
32. #endif // !VARIABLE_LIST_ITEM_H
33. 
```

VariableListItem.cpp

```
1. #include <vector>
2. #include <string>
3. 
4. #include "VariableListItem.hpp"
5. 
6. /**
7.  * Constructor | VariableListItem | VariableListItem
8.  *
9.  * Summary: Instantiates a variable item with the specified values. Note that no
10. *          values are passed in to the statement index. This is because
11. *          this index is only used for forward chaining and is populated
12. *          at that time.
13. *
14. * @param string nameP:   Name of the clause variable .
15. * @param bool instantiatedP: If it has been given a value. Will start as false.
16. * @param string valueP:  Initial value of the clause.
17. * @param string descriptionP: A prompt to display when assigning a value.
18. * @param int typeP:  The type of clause, usually set to STRING.
19. *
20. */
21. VariableListItem::VariableListItem(std::string nameP, bool instantiatedP, std::string valueP, std::string descriptionP
    , int typeP)
22. {
23.     name = nameP;
24.     instantiated = instantiatedP;
25.     value = valueP;
26.     description = descriptionP;
```

```
27.     type = typeP;
28.     statementIndex.push_back(-1); // To keep in line with the other indexes. But use -1 to note bad values
29. }
30. 
31. /**
32.  * CMember function | VariableListItem | populateAtatementIndex
33.  *
34.  * Summary: Creates an inverted index to allow for quicker searching when using
35.  *          forward chaining. Also helps prevent the need to double search or
36.  *          process needless entries.
37.  *
38.  * @param const KnowledgeBase& knowledgeBase: Used to find entries that match
39.  *          the name for this variable list entry.
40.  */
41. void VariableListItem::populateStatementIndex(const KnowledgeBase& knowledgeBase)
42. {
43.     bool isFound = false;
44. 
45.     for (int statementIter = 1; statementIter < knowledgeBase.kBase.size(); statementIter++)
46.     {
47.         isFound = false;
48.         for (int premiseIter = 1; (!isFound && premiseIter < knowledgeBase.kBase.at(statementIter).premiseList.size())
    ; premiseIter++)
49.         {
50.             if (name == knowledgeBase.kBase.at(statementIter).premiseList.at(premiseIter).name)
51.             {
52.                 isFound = true;
53.                 statementIndex.push_back(statementIter);
54.             }
55.         }
56.     }
57. }
58. 
59. 
```

KnowledgeBase.hpp

```
1. #ifndef KNOWLEDGE_BASE_H
2. #define KNOWLEDGE_BASE_H
3.
4. #include <vector>
5. #include <string>
6. #include <set>
7.
8. #include "ClauseItem.hpp"
9. #include "Statement.hpp"
10.
11./**
12.* KnowledgeBase - parses a text file of statements and acts as a database; it contains a vector of stateme
   nt.
13.* This class allows us to separate the knowledge base more clearly from the inference enginge.
14.*/
15.class KnowledgeBase
16.{
17.public:
18.    KnowledgeBase();  // Constructor
19.    void displayBase();  // Display Entire KnowledgeBase
20.    void populateKnowledgeBase(std::string fileName);
21.    std::string getConclusion(unsigned int);  //  get a conclusion from index provided
22.    std::string getPremise(unsigned int, unsigned int); //first UI is kBase index, second is premise index
23.    std::vector<Statement> kBase;
24.    std::set<std::string> conclusionSet;
25.private:
26.    bool isConclusionGood(Statement&, std::string, std::string&);
27.    bool arePremisesGood(Statement&, std::string);
28.};
29.
30.#endif //KNOWLEDGE_BASE_H
```

KnowledgeBase.cpp

```cpp
#include <iostream>
#include <fstream>
#include <stdexcept>

#include "KnowledgeBase.hpp"

KnowledgeBase::KnowledgeBase()
{
    std::cout << "\nCreating knowledge base instance..." << std::endl;
}

/**
 * populateKnowledgeBase - reads in a data file and creates the knowledge base (KB)
 * accordingly, provided the right format is used.
 * File format example
 * issue = Failure to Start ^ has_fuel = n : repair = Insufficient Fuel, Add more fuel.
 *
 * = is used to separate variable and value
 * ^ is logical AND
 * : separates clause and conclusion
 *
 * The above example should be read as follows. If issue is equal to failure to start, and has fuel is false
 * then repair conclusion equals "Insufficient Fuel, Add more fuel."
 *
 * @param string fileName - the name of the file containing the knowledge base. knowledgeBase.txt
 *
 * @return none
 *
 */
void KnowledgeBase::populateKnowledgeBase(std::string fileName)
{
    int total_good = 0, total_bad = 0;

```

```
34.     std::string inputBuffer;
35.     std::ifstream inputFile;
36.     inputFile.open(fileName);
37.
38.     if (inputFile)
39.     {
40.         while (getline(inputFile, inputBuffer))  // while read was successfull
41.         {
42.             std::cout << "Processing:  " << inputBuffer << std::endl;
43.             if (inputBuffer.size() > 0)
44.             {
45.                 Statement lList;
46.                 std::string listPremise;
47.
48.                 int indicatorLocation;
49.                 if (isConclusionGood(lList, inputBuffer, listPremise))
50.                 {
51.                     if (arePremisesGood(lList, listPremise))
52.                     {
53.                         std::cout << "Conclusion and premise(s) are good => List Updated\n";
54.                         kBase.push_back(lList);
55.                         ++total_good;
56.                     }
57.                     else
58.                     {
59.                         std::cout << "Premise List is formatted incorrectly.  List NOT updated\n";
60.                         ++total_bad;
61.                     }
62.                 }
63.                 else
64.                 {
65.                     std::cout << "Conclusion is formatted incorrectly.  List NOT updated\n";
66.                     ++total_bad;
67.                 }
68.             }
69.             std::cout << std::endl;
```

```
70.         }
71.     }
72.     else {
73.         throw std::runtime_error("Error reading Knowledge Base (KB) file. Please validate it uses the correct format.
   Invoke application with -h or -help for details.");
74.     }
75.     std::cout << "\nKnowledge Base finished Loading.\n" << total_good << " items were loaded into the KnowledgeBase\n"
   ;
76.     if ( total_bad > 0 )
77.         std::cerr << "\nWARNING! " << total_bad << " malfromed item(s) were not loaded into the Knowledge Base. " <<
78.         "\nPlease check output above for items not loaded and inspect data file.\n";
79.     std::cout << "<CR/Enter> to continue  ";
80.     std::cin.ignore();
81.}
82.
83./**
84. * isConclusionGood - helper function to check if the conclusion in the premise is valid and trim any white spaces
85. *
86. * @param Statement& lList - a statement containing the premise list and the conclusion it leads to
87. * @param std::string iBuffer - the entire statement (premise and conclusions)
88. * @param std::string& listPremise - a list of the premises (left side of expression)
89. *
90. * @return bool - if the conclusion is valid, true. Otherwise false.
91. */
92.bool KnowledgeBase::isConclusionGood(Statement& lList, std::string iBuffer, std::string& listPremise)
93.{
94.     ClauseItem nClause;
95.     std::string listConclusion;
96.     int indicatorLocation = iBuffer.find(':', 0);  // THEN symbol
97.
98.     if (indicatorLocation == -1)
99.         return false;
100.
101.          listPremise = iBuffer.substr(0, indicatorLocation);
102.          listConclusion = iBuffer.substr(indicatorLocation + 1, iBuffer.size());
103.          indicatorLocation = listConclusion.find('=', 0);  // Assignment Symbol
```

```
104.
105.         if (indicatorLocation == -1)
106.             return false;
107.
108.         lList.conclusion.name = listConclusion.substr(0, indicatorLocation);
109.
110.         // trim white spaces in conclusion name, first front, then back
111.         if (lList.conclusion.name.front() == ' ')
112.         {
113.             lList.conclusion.name = lList.conclusion.name.substr(1);
114.         }
115.         if (lList.conclusion.name.back() == ' ')
116.         {
117.             lList.conclusion.name = lList.conclusion.name.substr(0, lList.conclusion.name.size() - 1);
118.         }
119.
120.         lList.conclusion.value = listConclusion.substr(indicatorLocation + 1, listConclusion.size());
121.         // trim white spaces in conclusion value, first front, then back
122.         if (lList.conclusion.value.front() == ' ')
123.         {
124.             lList.conclusion.value = lList.conclusion.value.substr(1);
125.         }
126.         if (lList.conclusion.value.back() == ' ')
127.         {
128.             lList.conclusion.value = lList.conclusion.value.substr(0, lList.conclusion.value.size() - 1);
129.         }
130.
131.         lList.conclusion.type = STRING;
132.         conclusionSet.insert(lList.conclusion.name); // add conclusion to set, maintaining unique list of conclusion
    s
133.
134.         std::cout << "Conclusion is good; ";
135.         return true;
136.     }
137.
138.     /**
```

```
139.      * isConclusionGood - helper function to check if the conclusion in the premise is valid and trim any white spac
   es
140.      *
141.      * @param Statement& lList - a statement containing the premise list and the conclusion it leads to
142.      * @param std::string& listPremise - a list of the premises (left side of expression)
143.      *
144.      * @return bool - if the premise is valid, true. Otherwise false.
145.      */
146.     bool KnowledgeBase::arePremisesGood(Statement& lList, std::string listPremise)
147.     {
148.         ClauseItem nClause;
149.         std::string listRight,
150.             listLeft,
151.             tmpList;
152.         do
153.         {
154.             int andLocation = listPremise.find('^', 0),
155.                 equalsLocation;
156.
157.             if (andLocation == -1 && listPremise.size() == 0) // no premise
158.                 return false;
159.
160.             if (andLocation == -1)
161.                 andLocation = listPremise.size();
162.
163.             tmpList = listPremise.substr(0, andLocation);
164.             listPremise.erase(0, andLocation + 1);
165.             equalsLocation = tmpList.find('=', 0);  // assignment Symbol
166.
167.             if (equalsLocation == -1)
168.                 return false;
169.
170.             listRight = tmpList.substr(0, equalsLocation);
171.             listLeft = tmpList.substr(equalsLocation + 1, tmpList.size());
172.
173.             if (listLeft.size() == 0 || listRight.size() == 0)  // missing something
```

```
174.                 return false;
175.
176.             // trim white space from clause name, first front then back
177.             nClause.name = listRight;
178.             if (nClause.name.front() == ' ')
179.             {
180.                 nClause.name = nClause.name.substr(1);
181.             }
182.             if (nClause.name.back() == ' ')
183.             {
184.                 nClause.name = nClause.name.substr(0, nClause.name.size() - 1);
185.             }
186.
187.             // trim white space from clause value, first front then back
188.             nClause.value = listLeft;
189.             if (nClause.value.front() == ' ')
190.             {
191.                 nClause.value = nClause.value.substr(1);
192.             }
193.             if (nClause.value.back() == ' ')
194.             {
195.                 nClause.value = nClause.value.substr(0, nClause.value.size() - 1);
196.             }
197.
198.             nClause.type = STRING;
199.             lList.premiseList.push_back(nClause);
200.         } while (listPremise.size() != 0);
201.
202.         std::cout << "Premise list is good!  " << lList.premiseList.size() - 1 << " premise(s) loaded\n";
203.         return true;
204.     }
205.
206.     /**
207.      * displayBase - prints out the imported knowledge base to the screen
208.      * We iterate over the knowledge base vector and print the premise list
209.      * and the conclusion they lead to in a human readable format.
```

```
210.      */
211.     void KnowledgeBase::displayBase()
212.     {
213.         Statement n;
214.         unsigned int pntr = 1;
215.         while (pntr < kBase.size())
216.         {
217.             unsigned int pPntr = 1,
218.                 lastPntr = kBase.at(pntr).premiseList.size();
219.             std::cout << pntr << ". IF ";
220.             while (pPntr < kBase.at(pntr).premiseList.size())
221.             {
222.                 std::cout << kBase.at(pntr).premiseList.at(pPntr).name;
223.                 pPntr++;
224.                 if (pPntr < lastPntr)
225.                     std::cout << " AND ";
226.
227.             }
228.             std::cout << " THEN " << kBase.at(pntr).conclusion.name << std::endl;
229.             pntr++;
230.         }
231.     }
232.
233.     /**
234.      * getConclusion - helper function, which allows to get a conclusion of a specific index in the KB
235.      *
236.      * @param unsigned int index - position of the statement in the knowledge base vector, whose conclusion we want
237.      *
238.      * @return std::string the string representation of the conclusion
239.      */
240.     std::string KnowledgeBase::getConclusion(unsigned int index)
241.     {
242.         return kBase.at(index).conclusion.name;
243.     }
244.
245.     /**
```

```
246.      * getConclusion - helper function, which allows to get a conclusion of a specific index in the KB
247.      *
248.      * @param unsigned int index - position of the statement in the knowledge base vector, whose conclusion we want
249.      * @param unsigned int index_1 - once we know which statement, we need to know which premise
250.      *                               as a statement may contain one or more premises.
251.      *
252.      * @return std::string the string representation of the premise
253.      */
254.     std::string KnowledgeBase::getPremise(unsigned int index, unsigned int index_1)
255.     {
256.         return kBase.at(index).premiseList.at(index_1).name;
257.     }
258.
259.
```

BackChain.hpp

```
1. #ifndef BACK_CHAIN_H
2. #define BACK_CHAIN_H
3.
4. #include <string>
5. #include <vector>
6.
7. #include "Statement.hpp"
8. #include "VariableListItem.hpp"
9. #include "ClauseItem.hpp"
10.#include "KnowledgeBase.hpp"
11.
12.class BackChain
13.{
14.public:
15.    void populateLists();
16.    void runBackwardChaining();
17.    void populateVariableList(std::string);
18.
19.    KnowledgeBase ruleSystem;
20.    std::vector<VariableListItem> variableList;
21.
22.    // Due to the design of the system (use the info from the backward
23.    // chain to populate the forward chain), this list is needed
24.    // to keep track of the conclusions that were set.
25.    std::vector<VariableListItem> intermediateConclusionList;
26.
27.private:
28.    int findValidConclusionInStatements(std::string conclusionName, int startingIndex, std::string stringToMatch);
29.    bool instantiatePremiseClause(const ClauseItem& clause);
30.    bool processPremiseList(const Statement& statement);
31.    void addToIntermediateConclusionList(const ClauseItem& intermediateConclusion);
32.
33.};
34.
```

```
35.#endif // !BACK_CHAIN_H
36.
```

BackChain.cpp

```
1. #include <iostream>
2. #include <fstream>
3.
4. #include "ClauseItem.hpp"
5. #include "BackChain.hpp"
6.
7. /**
8.  * Member Function | BackChain | populateLists
9.  *
10. * Summary: Populates the knowledge base and variable lists for the back
11. *          chaining portion of the program. Typically read from a csv or text
12. *          file. Takes the outside representation of the knowledge base and
13. *          allows the inference engine to act on it.
14. */
15.void BackChain::populateLists()
16.{
17.    // To offest the vectors by 1, populate index 0 with NULL or Empty elements.
18.    ruleSystem.kBase.push_back(Statement());
19.    variableList.push_back(VariableListItem("Empty", false, "", "This is an error string", STRING));
20.    intermediateConclusionList.push_back(VariableListItem("Empty", false, "", "This is an error string", STRING));
21.
22.    // Populate the knowledge base and variable list.
23.    ruleSystem.populateKnowledgeBase("knowledgeBase.txt");
24.    populateVariableList("variablesList.csv");
25.}
26.
27./**
28. * Member Function | BackChain | populateVariableList
29. *
30. * Summary: Populates the variable list for back chaining. This initial list
31. *          contains only the premises. Note that is is later passed on
```

```
 *          to forward chaining in a modified format.
 *
 * @param string fileName: The name of the file to read entries from. This file
 *                  is in a CSV format of name, prompt, type.
 *
 */
void BackChain::populateVariableList(std::string fileName)
{
    std::string csvLine;

    std::string name;
    std::string prompt = " ";
    int type = 1;
    int startParseLocation = 0;
    int endParseLocation = 0;
    bool isValid = true;

    std::ifstream variableListFile;
    variableListFile.open(fileName);
    int varCount = 0;

    std::cout << "List of variables: ";

    if (variableListFile)
    {
        while (getline(variableListFile, csvLine))
        {
            startParseLocation = 0;
            endParseLocation = (csvLine.find(',', startParseLocation) - startParseLocation);
            if (endParseLocation <= -1)
            {
                isValid = false;
            }
            name = csvLine.substr(startParseLocation, endParseLocation);

            std::cout << name << ", ";
```

```
68.
69.             startParseLocation = endParseLocation + 1;
70.             endParseLocation = (csvLine.find(',', startParseLocation) - startParseLocation);
71.             if (endParseLocation <= -1)
72.             {
73.                 isValid = false;
74.             }
75.             prompt = csvLine.substr(startParseLocation, endParseLocation);
76.
77.             startParseLocation = endParseLocation + 1;
78.             endParseLocation = (csvLine.find(',', startParseLocation) - startParseLocation);
79.             if (endParseLocation <= -1)
80.             {
81.                 isValid = false;
82.             }
83.
84.             if (isValid)
85.             {
86.                 variableList.push_back(VariableListItem(name, false, "", prompt, type));
87.                 varCount++;
88.             }
89.             else
90.             {
91.                 std::cout << "\nInvalid entry, line " << csvLine << " not added." << std::endl;
92.             }
93.         }
94.         std::cout << "\nNumber of variables: " << varCount << std::endl;
95.     }
96.     else
97.     {
98.         std::cout << "Could not find the file" << std::endl;
99.     }
100.    }
101.
102.    /**
103.     * Member Function | BackChain | processPremiseList
```

```
104.      *
105.      * Summary: Processed the premise list of a given statement. This statement
106.      *          can lead to a recursive call if the statement contains a conclusion
107.      *          in its premise list.
108.      *
109.      * Preconditions:   The statement parameter was found to have a valid conclusion.
110.      *
111.      * @param Statement& statement   :   An individual statement taken from the
112.      *                                   knowledge base. It typically includes
113.      *                                   premise clauses that will be processed
114.      *                                   in order to see if they are conclusions,
115.      *                                   premise clauses, or just invalid.
116.      *
117.      */
118.     bool BackChain::processPremiseList(const Statement &statement)
119.     {
120.         int solution = 0;
121.         bool isValid = true;
122.         int location = 0;
123.         int conclusionLocation = 0;
124.         std::string valueToMatch = "";
125.
126.         // Process the premise list for a conclusion that was found to be valid.
127.         for (int premiseIter = 1; (isValid && premiseIter < statement.premiseList.size()); premiseIter++)
128.         {
129.
130.             // Go through and if it is a conclusion on the premise side,
131.             // back chain with it.
132.             // This will cause another recursive call by adding a conclusion
133.             // to the stack. It is this step that allows the removal of the actual
134.             // stack in back chaining.
135.             conclusionLocation = findValidConclusionInStatements(statement.premiseList.at(premiseIter).name, 1,
136.                                                                  statement.premiseList.at(premiseIter).value);
137.
138.             // It is a conclusion but not valid
139.             if (conclusionLocation == -1)
```

```
140.             {
141.                 isValid = false;
142.             }
143.
144.             // It is a conclusion and valid
145.             if (conclusionLocation > 0)
146.             {
147.                 isValid = true;
148.
149.                 // This step is not needed for the backward chaining portion. It is used
150.                 // when farward chaining is to immediately follow backward chaining and
151.                 // use values that have already been instantiated.
152.                 addToIntermediateConclusionList(statement.premiseList.at(premiseIter));
153.             }
154.
155.             // It was not a conclusion. Go to the clause variable list and
156.             // check if it is instantiated as well as what the value was.
157.             if (conclusionLocation == 0)
158.             {
159.                 isValid = instantiatePremiseClause(statement.premiseList.at(premiseIter));
160.             }
161.         }
162.
163.         return isValid;
164.     }
165.
166.     /**
167.      * Member Function | BackChain | instantiatePremiseClause
168.      *
169.      * Summary: Checks to see if the single premise clause passed in has a matching
170.      *          value to what exists in the clause variable list. If not, ask for
171.      *          a value and then check.
172.      *
173.      * Postcondition:   The clause variable list, if it matches, will have the
174.      *                  matching entry instantiated.
175.      *
```

```
176.      * @param  const ClauseItem& clause:   An individual premise clause. Contains
177.      *                                      a name and potentially a value.
178.      *
179.      * @return bool isValid:   Returns if the individual premise clause was found
180.      *                         to be valid by matching the name and value of the
181.      *                         premise clause in the knowledge base to what is
182.      *                         in the clause variable list.
183.      *
184.      */
185.     bool BackChain::instantiatePremiseClause(const ClauseItem &clause)
186.     {
187.         bool isValid = false;
188.         bool isFound = false;
189.
190.         // Go through the entire clause variable list and look for the matching
191.         // Entry. It has to find a match. If not, it could be the case that
192.         // The two are out of sync with eachother (the knowledge base and clause
193.         // variable list).
194.         for (int premiseClauseIter = 1; (!isFound && premiseClauseIter < variableList.size()); premiseClauseIter++)
195.         {
196.             // If the premise clause we are looking to resolve matches, check
197.             // its status.
198.             if (clause.name == variableList.at(premiseClauseIter).name)
199.             {
200.                 isFound = true;
201.
202.                 // This means that it is the first time we encountered this
203.                 // premise. We need more info and will get it in this step.
204.                 if (!variableList.at(premiseClauseIter).instantiated)
205.                 {
206.                     std::cout << variableList.at(premiseClauseIter).description << ": ";
207.                     std::cin >> variableList.at(premiseClauseIter).value;
208.                     std::cout << "\nYou entered: " << variableList.at(premiseClauseIter).value << std::endl;
209.                     variableList.at(premiseClauseIter).instantiated = true;
210.                 }
211.
```

```
212.                 // Clause variable list is guaranteed to be updated here.
213.                 // It can now be safely compared to the incoming premise clause
214.                 // value.
215.                 if (variableList.at(premiseClauseIter).value == clause.value)
216.                 {
217.                     isValid = true;
218.                 }
219.             }
220.         }
221.
222.         // The premise clause we just looked at will either be good or bad.
223.         // One thing to note is that this is for an individual premise.
224.         return isValid;
225.     }
226.
227.     /**
228.      * Member Function | BackChain | findValidConclusionInStatements
229.      *
230.      * Summary: Takes a conclusion name and value and tries to find a statement
231.      *          that matches up to both. If it finds one and the recursive stack
232.      *          is done, that will be the solution. If the stack is not empty,
233.      *          that means that we just completed an intermediate step in the process.
234.      *
235.      * @param string conclusionName: The name of a conclusion to match up to. Used
236.      *                                As the first part in checking if a statement
237.      *                                is valid or not.
238.      * @param int startingIndex: The first index location to begin searching
239.      *                                from. Typically a 1, but can be adjusted.
240.      *          string  stringToMatch:  The value portion to be matched when searching
241.      *                                for a valid conclusion. It will take on
242.      *                                a value when this function is called
243.      *                                recursively.
244.      *
245.      * @return int location:   Specifies the location of a conclusion.
246.      *
247.      */
```

```
248.     int BackChain::findValidConclusionInStatements(std::string conclusionName, int startingIndex, std::string string
   ToMatch)
249.     {
250.         int location = 0;
251.         bool isConclusion = false;
252.         bool isValid = false;
253.
254.         // This loop will go through the knowledge base and look for a matching
255.         // conclusion in all of the statements. It initially is not trying to
256.         // find a match to the conclusion value, as the first inquiry will be
257.         // the open ended question that the user wants the system to solve.
258.         // It also begins at index 1 for the first run.
259.         for (int conclusionIter = startingIndex; (conclusionIter < ruleSystem.kBase.size() && !isValid); conclusionI
   ter++)
260.         {
261.             // Check that the conclusion name matches what the user entered or
262.             // if the back chain is recursing, see if it matches the conclusion
263.             // Next in the list.
264.             if (conclusionName == ruleSystem.kBase.at(conclusionIter).conclusion.name)
265.             {
266.                 // It matched the conclusion name, just that at this point.
267.                 isConclusion = true;
268.
269.                 // Note the DONTCARE here. This allows the initial inquiry to go through
270.                 // Since it is open ended. However, if not DONTCARE, the stringToMatch
271.                 // Parameter that was passed in must match. This is due to the multi
272.                 // purposing of this function.
273.                 if (stringToMatch == ruleSystem.kBase.at(conclusionIter).conclusion.value || stringToMatch == "DONTC
   ARE")
274.                 {
275.                     // It matched the conclusion name (above) and now it also matched the
276.                     // value in the knowledge base. This needs to be fully processed.
277.                     // Process premiseList will do just that for this statement.
278.                     // If everything lines up, we are good.
279.                     isValid = processPremiseList(ruleSystem.kBase.at(conclusionIter));
280.
```

```
281.                     if (isValid)
282.                     {
283.                         // Everything matched up, conclusion name, conclusion value
284.                         // and the premises all were good.
285.                         location = conclusionIter;
286.                     }
287.                 }
288.             }
289.         }
290.
291.         /*
292.         * There are actually three options here. -1 means that the conclusion name
293.         * was found, but it was not valid. This could happen if there is a bad
294.         * knowledge base or perhaps the user entered in a bad value, such as an x
295.         * instead of a y or n.
296.         *
297.         * The second option is 0, which means there was no match, no nothing.
298.         * This can happen if the user enters in a bad inquiry to start.
299.         *
300.         * The third option is the actual index of where a valid conclusion was found.
301.         */
302.         if (isConclusion && !isValid)
303.         {
304.             location = -1;
305.         }
306.
307.         // See comment right above this one for info on this return value.
308.         return location;
309.     }
310.
311.     /**
312.      * Member Function | BackChain | runBackwardChaining
313.      *
314.      * Summary: The entry point for starting the backward chain process. Asks
315.      *          the user to enter the conclusion to solve and then runs.
316.      *
```

```
317.      */
318.     void BackChain::runBackwardChaining()
319.     {
320.
321.         std::string conclusionToSolve = "";
322.         int conclusionLocation = 0;
323.         bool isSolvedStatement = false;
324.
325.         std::cout << "Please enter a conclusion to solve (values can be: ";
326.         int tmpSetCounter = 0;
327.         for (auto f : ruleSystem.conclusionSet)
328.         {
329.             std::cout << f;
330.             tmpSetCounter++;
331.             if (tmpSetCounter != ruleSystem.conclusionSet.size())
332.             {
333.                  std::cout << ", ";
334.             }
335.         }
336.         std::cout << "): ";
337.         std::cin >> conclusionToSolve;
338.         std::cout << "\nYou entered: " << conclusionToSolve << std::endl;
339.
340.         conclusionLocation = findValidConclusionInStatements(conclusionToSolve, 1, "DONTCARE");
341.
342.         //is a conclusion but not valid
343.         if (conclusionLocation == -1)
344.         {
345.             std::cout << "No conclusion match available. Based on your entries, the results are inconclusive. ";
346.         }
347.
348.         //is a conclusion and valid
349.         if (conclusionLocation > 0)
350.         {
351.             std::cout << "\nResult is: " << ruleSystem.kBase.at(conclusionLocation).conclusion.value << std::endl;
352.             std::cout << "Conclusion is valid. ";
```

```
353.         }
354.
355.         //not a conclusion
356.         if (conclusionLocation == 0)
357.         {
358.             std::cout << "No conclusion. ";
359.         }
360.     }
361.
362.     /**
363.      * Member Function | BackChain | addToIntermediateConclusionList
364.      *
365.      * Summary: This function allows for intermediate conclusion clauses to be added
366.      *          to the forward chaining variable list. This is needed in order to
367.      *          run the forward chaining portion immediately after the backward
368.      *          chaining portion. It allows for all entries including the resolved
369.      *          intermediate conclusion clauses to be preserved. Note that the
370.      *          description field is not specific. This is entered in here only to
371.      *          keep in line with the other entries. It will not actually be seen.
372.      *
373.      */
374.     void BackChain::addToIntermediateConclusionList(const ClauseItem &intermediateConclusion)
375.     {
376.         intermediateConclusionList.push_back(
377.             VariableListItem(intermediateConclusion.name,
378.                              true,
379.                              intermediateConclusion.value,
380.                              (intermediateConclusion.name + "(y/n)"),
381.                              intermediateConclusion.type));
382.     }
383.
```

ForwardChain.hpp

```
1. #ifndef FORWARD_CHAIN_H
2. #define FORWARD_CHAIN_H
3.
4. #include <string>
5. #include <vector>
6. #include <queue>
7.
8. #include "Statement.hpp"
9. #include "VariableListItem.hpp"
10.#include "ClauseItem.hpp"
11.#include "KnowledgeBase.hpp"
12.
13./**
14.* Through forward chaining, we provide repair recommendations to the user based on their input.
15.* We pass a copy of the KnowledgeBase from BackChain to ForwardChain.
16.*/
17.class ForwardChain
18.{
19.public:
20.    void runForwardChaining();
21.    void copyVariableList(const std::vector<VariableListItem>& srcVariableList);
22.    void copyKnowledgeBase(const KnowledgeBase& srcKnowledgeBase);
23.    int getMatchingVariableListEntry(std::string entryName);
24.    void addIntermediateConclusions(const std::vector<VariableListItem>& srcConclusionVariableList);
25.
26.    std::queue<ClauseItem> conclusionVariableQueue;
27.
28.    KnowledgeBase ruleSystem;
29.    std::vector<VariableListItem> variableList;
30.
31.private:
32.    void processStatementIndex(int variableListEntry);
33.    bool instantiatePremiseClause(const ClauseItem& clause);
```

```
34.     bool processPremiseList(std::vector<ClauseItem>& premiseList);
35. };
36.
37. #endif // !FORWARD_CHAIN_H
38.
```

ForwardChain.cpp

```
1.  #include <iostream>
2.
3.  #include "ForwardChain.hpp"
4.
5.  /**
6.   * Member Function | ForwardChain | copyVariableList
7.   *
8.   * Summary: Copies over all values that were used to find the backward chaining
9.   *          conclusion from the clause variable list. This is part one of two.
10.  *          The second part will also bring over the intermediate conclusions.
11.  *
12.  * @param  const vector<VariableListItem>& srcVariableList:    The variable
13.  *                  list to be copied over. Each element is copied with the
14.  *                  current values instantiated by backward chaining.
15.  *
16.  */
17. void ForwardChain::copyVariableList(const std::vector<VariableListItem> &srcVariableList)
18. {
19.     //do start at 0 in this case, might as well copy over the NULL.
20.     for (int varListiter = 0; varListiter < srcVariableList.size(); varListiter++)
21.     {
22.         variableList.push_back(srcVariableList.at(varListiter));
23.         variableList.back().populateStatementIndex(ruleSystem);
24.     }
25. }
26.
```

```
/**
 * Member Function | ForwardChain | addIntermediateConclusions
 *
 * Summary: Takes the intermediate conclusion list that was populated during
 *          backward chaining and adds it to the clause variable list. This is
 *          needed due to forward chaining requiring all premise items to be in
 *          the clause variable list, which is different from backward chaining.
 *
 * @param  const vector<VariableListItem>& srcConclusionVariableList:  The
 *                  additional conclusion variable list items to copy into the
 *                  forward chaining clause variable list.
 *
 */
void ForwardChain::addIntermediateConclusions(const std::vector<VariableListItem> &srcConclusionVariableList)
{
    //do start at 0 in this case, might as well copy over the NULL.
    for (int conclListIter = 0; conclListIter < srcConclusionVariableList.size(); conclListIter++)
    {
        variableList.push_back(srcConclusionVariableList.at(conclListIter));
        variableList.back().populateStatementIndex(ruleSystem);
    }
}

/**
 * Member Function | ForwardChain | copyKnowledgeBase
 *
 * Summary: Copy the internal representation of the knowledge base over to be used
 *          inforward chaining.
 *
 * @param  const KnowledgeBase& srcKnowledgeBase:  The internal representation
 *          of the knowledge base to be used. Similar to what was done for
 *          back chaining.
 */
void ForwardChain::copyKnowledgeBase(const KnowledgeBase &srcKnowledgeBase)
{
    //do start at 0 in this case, might as well copy over the NULL.
```

```
63.     for (int kBaseIter = 0; kBaseIter < srcKnowledgeBase.kBase.size(); kBaseIter++)
64.     {
65.         ruleSystem.kBase.push_back(srcKnowledgeBase.kBase.at(kBaseIter));
66.     }
67. }
68. 
69. /**
70.  * Member Function | ForwardChain | runForwardChaining
71.  *
72.  * Summary: Entry point for running forward chaining. This is expected to run
73.  *          after backward chaining, as part of the suggested fix step.
74.  *
75.  */
76. void ForwardChain::runForwardChaining()
77. {
78.     ClauseItem queueTopPtr;
79.     int variableListEntry;
80.     int initialRepairEntry;
81. 
82.     queueTopPtr.name = "inconclusive";
83.     queueTopPtr.value = "no valid solution.";
84. 
85.     std::cout << std::endl
86.               << std::endl
87.               << "Now running forward chain" << std::endl;
88. 
89.     // Start the chain by looking for the very first prompt, does it have an issue.
90.     // Note that this will also prevent forward chaining from running if the user
91.     // entered in a bad value to resolve while back chaining.
92.     initialRepairEntry = getMatchingVariableListEntry("has_issue");
93. 
94.     if (variableList.at(initialRepairEntry).instantiated)
95.     {
96.         conclusionVariableQueue.push(ClauseItem(variableList.at(initialRepairEntry).name,
97.                                                 variableList.at(initialRepairEntry).value,
98.                                                 variableList.at(initialRepairEntry).type));
```

```
99.     }
100.
101.         while (!conclusionVariableQueue.empty())
102.         {
103.             queueTopPtr = conclusionVariableQueue.front();
104.             std::cout << "Processing " << queueTopPtr.name << std::endl;
105.             // Note that this is the only location where the queue is reduced.
106.             conclusionVariableQueue.pop();
107.
108.             //get the matching entry in the variable list, the value does not matter at this time.
109.             variableListEntry = getMatchingVariableListEntry(queueTopPtr.name);
110.
111.             //go through the variable list's inverted index of statements and push any valid conclusions.
112.             //make sure to prompt for entry of any non instantiated.
113.             if (variableListEntry != -1)
114.             {
115.                 processStatementIndex(variableListEntry);
116.             }
117.         }
118.
119.         std::cout << "The final conclusion is - " << queueTopPtr.name << " - with a value of: " << queueTopPtr.value
    << std::endl;
120.     }
121.
122.     /**
123.      * Member Function | ForwardChain | processStatementIndex
124.      *
125.      * Summary: Runs through an inverted index of the current variable list entry and
126.      *          checks to see which statements are to be processed due to its value.
127.      *          This step is part of the BFS, where each matching item
128.      *          is added to the queue for this particular entry before it is
129.      *          popped off the queue and the next one is processed.
130.      *
131.      *  @param int variableListEntry: A numeric value of which variable list entry
132.      *          is being processed.
133.      *
```

```
134.      *  @return    None - note that this is indeed the case since the queue will be
135.      *              added to if there is a valid value. If it is not valid, it is not
136.      *              added.
137.      */
138.     void ForwardChain::processStatementIndex(int variableListEntry)
139.     {
140.         int curStatement = 0;
141.
142.         for (int variableListIter = 1; variableListIter < variableList.at(variableListEntry).statementIndex.size();
    variableListIter++)
143.         {
144.             //           = The matching variable list entry  . The individual statment number
145.             curStatement = variableList.at(variableListEntry).statementIndex.at(variableListIter);
146.             if (true == processPremiseList(ruleSystem.kBase.at(curStatement).premiseList))
147.             {
148.                 // Everything matched up, so move forward on adding it to the queue to be
149.                 // processed.
150.                 conclusionVariableQueue.push(ruleSystem.kBase.at(curStatement).conclusion);
151.             }
152.         }
153.     }
154.
155.     /**
156.      * Member Function | ForwardChain | processPremiseList
157.      *
158.      * Summary: Takes a conclusion name and value and tries to find a statement
159.      *          that matches up to both. If it finds one and the recursive stack
160.      *          is done, that will be the solution. If the stack is not empty,
161.      *          that means that we just completed an intermediate step in the process.
162.      *
163.      * @param  vector<ClauseItem>& premiseList: the premise list of a particular
164.      *                      statement.
165.      *
166.      * @return bool isValid: Specifies if the premise clauses were all found
167.      *                  to be valid for a particular statement..
168.      *
```

```
169.      */
170.     bool ForwardChain::processPremiseList(std::vector<ClauseItem> &premiseList)
171.     {
172.         bool isValid = true;
173.
174.         for (int premiseIter = 1; (isValid && premiseIter < premiseList.size()); premiseIter++)
175.         {
176.             isValid = instantiatePremiseClause(premiseList.at(premiseIter));
177.         }
178.         return isValid;
179.     }
180.
181.     /**
182.      * Member Function | ForwardChain | instantiatePremiseClause
183.      *
184.      * Summary: Takes a single clause and verifies that it has been instantiated and
185.      *          that it matches up to the back chaining portion of diagnostics.
186.      *          Note that this varies slightly from back chaining and the typical
187.      *          behavior of the instantiation step.
188.      *
189.      * Preconditions: Backchaining has been ran and the variable list is populated
190.      *                  with the needed results.
191.      *
192.      * @param  string  conclusionName: The name of a conclusion to match up to. Used
193.      *                                  As the first part in checking if a statement
194.      *                                  is valid or not.
195.      *
196.      * @return isFound:    Specifies if the incoming clause is found within the
197.      *                      variable list.
198.      */
199.     bool ForwardChain::instantiatePremiseClause(const ClauseItem &clause)
200.     {
201.         bool isFound = false;
202.
203.         // Look through the variable list and see if this particular clause has a valid match.
204.         for (int varListIter = 1; (!isFound && varListIter < variableList.size()); varListIter++)
```

```
205.         {
206.             if (variableList.at(varListIter).instantiated && clause.name == variableList.at(varListIter).name && cla
   use.value == variableList.at(varListIter).value)
207.             {
208.                 isFound = true;
209.             }
210.         }
211.
212.         return isFound;
213.     }
214.
215.     /**
216.      * Member Function | ForwardChain | getMatchingVariableListEntry
217.      *
218.      * Summary: Takes a conclusion name that was popped off of the queue and tries to
219.      *          locate it in the variable list. Note that the variable list will only
220.      *          contain unique values.
221.      *
222.      * @param  string entryName: The name of the variable list entry to match up.
223.      *
224.      * @return int matchingEntryIndex: The location of the matching entry. Returns
225.      *                  -1 if it is not found.
226.      *
227.      */
228.     int ForwardChain::getMatchingVariableListEntry(std::string entryName)
229.     {
230.         int matchingEntryIndex = -1;
231.         bool isFound = false;
232.
233.         for (int variableListIter = 1; (!isFound && variableListIter < variableList.size()); variableListIter++)
234.         {
235.             if (entryName == variableList.at(variableListIter).name)
236.             {
237.                 isFound = true;
238.                 matchingEntryIndex = variableListIter;
239.             }
```

```
240.          }
241.
242.          return matchingEntryIndex;
243.      }
244.
```

VariablesList.csv

1. has_issue,Is there an issue with the vehicle? (y/n),STRING
2. is_starting,Does the car start? (y/n),STRING
3. has_fuel,Is there enough fuel? (y/n),STRING
4. has_voltage,Does the battery have enough voltage? (y/n),STRING
5. is_ignition_coil_damaged,Is the ignition coil damaged? (y/n),STRING
6. is_distributor_cap_damaged,Is the distributor cap cracked or broken? (y/n),STRING
7. is_timing_belt_damaged,Is the timing belt damaged? (y/n),STRING
8. is_making_noise,Is the car making an unusual noise? (y/n),STRING
9. is_noisy_while_driving,Does the noise occur when driving? (y/n),STRING
10. is_ticking_noise,Is the car making a ticking noise? (y/n),STRING
11. is_hiccup_noise,Is the car making a rattly noise like a hiccup? (y/n),STRING
12. is_air_filter_dirty,Is the air filter dirty? (y/n),STRING
13. is_exhaust_blocked,Is anything blocking the exhaust? (y/n),STRING
14. are_wheel_bearings_damaged,Are wheel bearings damaged/loose? (y/n),STRING
15. are_tires_bald,Are tires beyond service life (tread depth less than 1/32in)? (y/n),STRING
16. is_truck,Is the vehicle a truck? (y/n),STRING
17. has_items_in_truck_bed,Are there items in the truck bed? (y/n),STRING
18. has_items_on_roof,Does the vehicle have any items latched to the roof? (y/n),STRING
19. is_overheating,Does the temperature gauge show as overheating? (y/n),STRING
20. has_coolant,Is there sufficient coolant in the engine? (y/n),STRING
21. is_water_pump_broken,Is the water pump broken? (y/n),STRING
22. is_oil_low_or_dirty,Is the oil level low or dirty (black in color)? (y/n),STRING
23. has_nonfunctional_electronics,Are any electronics not functional? (y/n),STRING
24. does_ac_power_on,Does the AC power on? (y/n),STRING
25. does_ac_blow_cold,Does the AC blow cold? (y/n),STRING
26. has_nonfunctional_headlights,Are any headlights or lights not turning on? (y/n),STRING

27. is_ac_fuse_intact,Is the AC fuse intact? (y/n),STRING
28. are_ac_wires_connected,Are AC ground and power wires connected? (y/n),STRING
29. are_therm_settings_correct,Are the thermostat settings correct? (y/n),STRING
30. is_evaporator_coil_frozen,Is the evaporator coil frozen? (y/n),STRING
31. is_air_filter_dirty,Is the air filter dirty? (y/n),STRING
32. is_nonfunct_light_fuse_intact,Is the respective non-working light fuse intact? (y/n),STRING
33. are_nonfunct_light_wires_conn,Are ground and power wires connected? (y/n),STRING
34. has_burning_plastic_smell,Do you smell any burning plastic / electric insulation? (y/n),STRING
35. is_radio_working,Does the radio power on? (y/n),STRING
36. is_radio_fuse_intact,Is the radio fuse intact? (y/n),STRING
37. are_radio_wires_connected,Are radio ground and power wires connected? (y/n),STRING
38. does_wheel_turn,Does the steering wheel turn? (y/n),STRING
39. is_power_steering_fuse_intact,Is the power steering fuse intact? (y/n),STRING
40. has_power_steering_fluid,Is there enough power steering fluid? (y/n),STRING
41. are_tires_deflated,Are any of the tires visibly deflated or is the low pressure light on? (y/n),STRING
42. has_piercing_object,Is there a sharp item (e.g. nail) stuck in the deflated tire? (y/n),STRING
43. is_obj_inch_away_from_edge,Is the piercing object more than 1in away from the tire edge? (y/n),STRING
44. is_recent_temp_change,Was there a recent temperature change? (y/n),STRING
45. is_exterior_damaged,Is the exterior of the vehicle damaged? (y/n),STRING
46. is_damage_cosmetic,Is the damage cosmetic paint scratch only? (y/n),STRING

KnowledgeBase.txt

```
1. has_issue = n : issue = No issue
2. issue = No issue : repair = No Repair Required
3. has_issue = y ^ is_starting = n : issue = Failure to Start
4. issue = Failure to Start ^ has_fuel = n : repair = Insufficient Fuel, Add more fuel.
5. issue = Failure to Start ^ has_fuel = y ^ has_voltage = n : repair = Dead Battery, Change the battery.
6. issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = y : repair = Bad Ignition Coil,
   Replace faulty ignition coil.
7. issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = n ^ is_distributor_cap_damaged
   = y : repair = Bad Distributor Cap, Replace faulty distributor cap.
8. issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = n ^ is_distributor_cap_damaged
   = n ^ is_timing_belt_damaged = y : repair = Bad Timing Belt, Replace faulty timing belt.
9. has_issue = y ^ is_starting = y ^ is_making_noise = y : issue = Noise Issue
```

10. issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = y : repair = Faulty bearings, replace wheel bearings
11. issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = y : repair = Tires are worn, replace with new tires
12. issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = n ^ has_items_on_roof = y : repair = Remove items from roof
13. issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = n ^ has_items_on_roof = n ^ is_truck = y ^ has_items_in_truck_bed = y : repair = Remove items from truck bed
14. issue = Noise Issue ^ is_noisy_while_driving = n ^ is_ticking_noise = n ^ is_hiccup_noise = y ^ is_air_filter_dirty = y : repair = Replace dirty air filter with clean one
15. issue = Noise Issue ^ is_noisy_while_driving = n ^ is_ticking_noise = n ^ is_hiccup_noise = y ^ is_air_filter_dirty = n ^ is_exhaust_blocked = y : repair = Remove object obstructing exhaust
16. has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = y : issue = Overheating Issue
17. issue = Overheating Issue ^ has_coolant = n : repair = Low Coolant, add coolant.
18. issue = Overheating Issue ^ has_coolant = y ^ is_water_pump_broken = y : repair = Defective Water Pump, replace water pump.
19. issue = Overheating Issue ^ has_coolant = y ^ is_water_pump_broken = n ^ is_oil_low_or_dirty = y : repair = Low or Dirty Oil, replace oil
20. has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = y : issue = Electronics Issue
21. issue = Electronics Issue ^ does_ac_power_on = n ^ is_ac_fuse_intact = n : repair = Replace defective AC fuse
22. issue = Electronics Issue ^ does_ac_power_on = n ^ is_ac_fuse_intact = y ^ are_ac_wires_connected = n : repair = Connect AC wires properly
23. issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = n : repair = Fix thermostat settings
24. issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = y ^ is_evaporator_coil_frozen = y : repair = Defrost evaporator coil
25. issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = y ^ is_evaporator_coil_frozen = n ^ is_air_filter_dirty = y : repair = Replace dirty air filter with clean one
26. issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = y ^ is_nonfunct_light_fuse_intact = n : repair = Replace defective light fuse
27. issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = y ^ is_nonfunct_light_fuse_intact = y ^ are_nonfunct_light_wires_conn = n : repair = Connect light wires properly
28. issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = y : repair = Serious issue and possible short circuit, replace wiring and fuses.
29. issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = n : repair = Replace defective radio fuse

30. issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = y ^ are_radio_wires_connected = n : repair = Connect radio wires properly
31. issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = y ^ are_radio_wires_connected = y : repair = Wiring or installation issue, replace radio and wiring
32. has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = n ^ does_wheel_turn = n : issue = Steering Issue
33. issue = Steering Issue ^ is_power_steering_fuse_intact = n : repair = Replace power steering fuse
34. issue = Steering Issue ^ is_power_steering_fuse_intact = y ^ has_power_steering_fluid = n : repair = Power steering failure, add power steering fluid
35. has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = n ^ does_wheel_turn = y ^ are_tires_deflated = y : issue = Tire Issue
36. issue = Tire Issue ^ has_piercing_object = y ^ is_obj_inch_away_from_edge = n : repair = Replace deflated tire
37. issue = Tire Issue ^ has_piercing_object = y ^ is_obj_inch_away_from_edge = y : repair = Apply tire patch kit to deflated tire
38. issue = Tire Issue ^ has_piercing_object = n ^ is_recent_temp_change = y : repair = Low tire pressure, add air to deflated tires
39. has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = n ^ does_wheel_turn = y ^ are_tires_deflated = n : issue = General Issue
40. issue = General Issue ^ is_exterior_damaged = y ^ is_damage_cosmetic = y : repair = Repaint the scratched part
41. issue = General Issue ^ is_exterior_damaged = y ^ is_damage_cosmetic = n : repair = Replace the damaged vehicle body part
42. issue = General Issue ^ is_exterior_damaged = n : repair = No damage/fault detected

Appendix C: Complete Sample Output

Sample Run One – Diagnosing Tire Issue

1. Welcome to the Automobile Diagnostic Program.
2. Authors: David Torrente (dat54@txstate.edu), Randall Henderson (rrh93@txstate.edu), Borislav Sabotinov (bss64@txstate.edu).
3.
4.
5. Creating knowledge base instance...

```
Processing:  has_issue = n : issue = No issue
Conclusion is good; Premise list is good!  1 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing:  issue = No issue : repair = No Repair Required
Conclusion is good; Premise list is good!  1 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing:  has_issue = y ^ is_starting = n : issue = Failure to Start
Conclusion is good; Premise list is good!  2 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing:  issue = Failure to Start ^ has_fuel = n : repair = Insufficient Fuel, Add more fuel.
Conclusion is good; Premise list is good!  2 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing:  issue = Failure to Start ^ has_fuel = y ^ has_voltage = n : repair = Dead Battery, Change the battery.
Conclusion is good; Premise list is good!  3 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing:  issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = y : repair = Bad
   Ignition Coil, Replace faulty ignition coil.
Conclusion is good; Premise list is good!  4 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing:  issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = n ^
   is_distributor_cap_damaged = y : repair = Bad Distributor Cap, Replace faulty distributor cap.
Conclusion is good; Premise list is good!  5 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing:  issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = n ^
   is_distributor_cap_damaged = n ^ is_timing_belt_damaged = y : repair = Bad Timing Belt, Replace faulty timing belt.
Conclusion is good; Premise list is good!  6 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing:  has_issue = y ^ is_starting = y ^ is_making_noise = y : issue = Noise Issue
Conclusion is good; Premise list is good!  3 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing:  issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = y : repair = Faulty
   bearings, replace wheel bearings
```

```
43.Conclusion is good; Premise list is good!  3 premise(s) loaded
44.Conclusion and premise(s) are good => List Updated
45.
46.Processing:  issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = y :
   repair = Tires are worn, replace with new tires
47.Conclusion is good; Premise list is good!  4 premise(s) loaded
48.Conclusion and premise(s) are good => List Updated
49.
50.Processing:  issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = n ^
   has_items_on_roof = y : repair = Remove items from roof
51.Conclusion is good; Premise list is good!  5 premise(s) loaded
52.Conclusion and premise(s) are good => List Updated
53.
54.Processing:  issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = n ^
   has_items_on_roof = n ^ is_truck = y ^ has_items_in_truck_bed = y : repair = Remove items from truck bed
55.Conclusion is good; Premise list is good!  7 premise(s) loaded
56.Conclusion and premise(s) are good => List Updated
57.
58.Processing:  issue = Noise Issue ^ is_noisy_while_driving = n ^ is_ticking_noise = n ^ is_hiccup_noise = y ^
   is_air_filter_dirty = y : repair = Replace dirty air filter with clean one
59.Conclusion is good; Premise list is good!  5 premise(s) loaded
60.Conclusion and premise(s) are good => List Updated
61.
62.Processing:  issue = Noise Issue ^ is_noisy_while_driving = n ^ is_ticking_noise = n ^ is_hiccup_noise = y ^
   is_air_filter_dirty = n ^ is_exhaust_blocked = y : repair = Remove object obstructing exhaust
63.Conclusion is good; Premise list is good!  6 premise(s) loaded
64.Conclusion and premise(s) are good => List Updated
65.
66.Processing:  has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = y : issue = Overheating Issue
67.Conclusion is good; Premise list is good!  4 premise(s) loaded
68.Conclusion and premise(s) are good => List Updated
69.
70.Processing:  issue = Overheating Issue ^ has_coolant = n : repair = Low Coolant, add coolant.
71.Conclusion is good; Premise list is good!  2 premise(s) loaded
72.Conclusion and premise(s) are good => List Updated
73.
74.Processing:  issue = Overheating Issue ^ has_coolant = y ^ is_water_pump_broken = y : repair = Defective Water Pump,
   replace water pump.
75.Conclusion is good; Premise list is good!  3 premise(s) loaded
76.Conclusion and premise(s) are good => List Updated
77.
```

78. Processing: issue = Overheating Issue ^ has_coolant = y ^ is_water_pump_broken = n ^ is_oil_low_or_dirty = y : repair = Low or Dirty Oil, replace oil
79. Conclusion is good; Premise list is good! 4 premise(s) loaded
80. Conclusion and premise(s) are good => List Updated
81.
82. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = y : issue = Electronics Issue
83. Conclusion is good; Premise list is good! 5 premise(s) loaded
84. Conclusion and premise(s) are good => List Updated
85.
86. Processing: issue = Electronics Issue ^ does_ac_power_on = n ^ is_ac_fuse_intact = n : repair = Replace defective AC fuse
87. Conclusion is good; Premise list is good! 3 premise(s) loaded
88. Conclusion and premise(s) are good => List Updated
89.
90. Processing: issue = Electronics Issue ^ does_ac_power_on = n ^ is_ac_fuse_intact = y ^ are_ac_wires_connected = n : repair = Connect AC wires properly
91. Conclusion is good; Premise list is good! 4 premise(s) loaded
92. Conclusion and premise(s) are good => List Updated
93.
94. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = n : repair = Fix thermostat settings
95. Conclusion is good; Premise list is good! 4 premise(s) loaded
96. Conclusion and premise(s) are good => List Updated
97.
98. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = y ^ is_evaporator_coil_frozen = y : repair = Defrost evaporator coil
99. Conclusion is good; Premise list is good! 5 premise(s) loaded
100. Conclusion and premise(s) are good => List Updated
101.
102. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = y ^ is_evaporator_coil_frozen = n ^ is_air_filter_dirty = y : repair = Replace dirty air filter with clean one
103. Conclusion is good; Premise list is good! 6 premise(s) loaded
104. Conclusion and premise(s) are good => List Updated
105.
106. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = y ^ is_nonfunct_light_fuse_intact = n : repair = Replace defective light fuse
107. Conclusion is good; Premise list is good! 5 premise(s) loaded
108. Conclusion and premise(s) are good => List Updated
109.

```
110.     Processing:  issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^
   has_nonfunctional_headlights = y ^ is_nonfunct_light_fuse_intact = y ^ are_nonfunct_light_wires_conn = n : repair =
   Connect light wires properly
111.     Conclusion is good; Premise list is good!  6 premise(s) loaded
112.     Conclusion and premise(s) are good => List Updated
113.
114.     Processing:  issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^
   has_nonfunctional_headlights = n ^ has_burning_plastic_smell = y : repair = Serious issue and possible short circuit,
   replace wiring and fuses.
115.     Conclusion is good; Premise list is good!  5 premise(s) loaded
116.     Conclusion and premise(s) are good => List Updated
117.
118.     Processing:  issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^
   has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = n :
   repair = Replace defective radio fuse
119.     Conclusion is good; Premise list is good!  7 premise(s) loaded
120.     Conclusion and premise(s) are good => List Updated
121.
122.     Processing:  issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^
   has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = y ^
   are_radio_wires_connected = n : repair = Connect radio wires properly
123.     Conclusion is good; Premise list is good!  8 premise(s) loaded
124.     Conclusion and premise(s) are good => List Updated
125.
126.     Processing:  issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^
   has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = y ^
   are_radio_wires_connected = y : repair = Wiring or installation issue, replace radio and wiring
127.     Conclusion is good; Premise list is good!  8 premise(s) loaded
128.     Conclusion and premise(s) are good => List Updated
129.
130.     Processing:  has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^
   has_nonfunctional_electronics = n ^ does_wheel_turn = n : issue = Steering Issue
131.     Conclusion is good; Premise list is good!  6 premise(s) loaded
132.     Conclusion and premise(s) are good => List Updated
133.
134.     Processing:  issue = Steering Issue ^ is_power_steering_fuse_intact = n : repair = Replace power steering fuse
135.     Conclusion is good; Premise list is good!  2 premise(s) loaded
136.     Conclusion and premise(s) are good => List Updated
137.
138.     Processing:  issue = Steering Issue ^ is_power_steering_fuse_intact = y ^ has_power_steering_fluid = n : repair
   = Power steering failure, add power steering fluid
```

```
139.     Conclusion is good; Premise list is good!  3 premise(s) loaded
140.     Conclusion and premise(s) are good => List Updated
141.
142.     Processing:  has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^
   has_nonfunctional_electronics = n ^ does_wheel_turn = y ^ are_tires_deflated = y : issue = Tire Issue
143.     Conclusion is good; Premise list is good!  7 premise(s) loaded
144.     Conclusion and premise(s) are good => List Updated
145.
146.     Processing:  issue = Tire Issue ^ has_piercing_object = y ^ is_obj_inch_away_from_edge = n : repair = Replace
   deflated tire
147.     Conclusion is good; Premise list is good!  3 premise(s) loaded
148.     Conclusion and premise(s) are good => List Updated
149.
150.     Processing:  issue = Tire Issue ^ has_piercing_object = y ^ is_obj_inch_away_from_edge = y : repair = Apply tire
   patch kit to deflated tire
151.     Conclusion is good; Premise list is good!  3 premise(s) loaded
152.     Conclusion and premise(s) are good => List Updated
153.
154.     Processing:  issue = Tire Issue ^ has_piercing_object = n ^ is_recent_temp_change = y : repair = Low tire
   pressure, add air to deflated tires
155.     Conclusion is good; Premise list is good!  3 premise(s) loaded
156.     Conclusion and premise(s) are good => List Updated
157.
158.     Processing:  has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^
   has_nonfunctional_electronics = n ^ does_wheel_turn = y ^ are_tires_deflated = n : issue = General Issue
159.     Conclusion is good; Premise list is good!  7 premise(s) loaded
160.     Conclusion and premise(s) are good => List Updated
161.
162.     Processing:  issue = General Issue ^ is_exterior_damaged = y ^ is_damage_cosmetic = y : repair = Repaint the
   scratched part
163.     Conclusion is good; Premise list is good!  3 premise(s) loaded
164.     Conclusion and premise(s) are good => List Updated
165.
166.     Processing:  issue = General Issue ^ is_exterior_damaged = y ^ is_damage_cosmetic = n : repair = Replace the
   damaged vehicle body part
167.     Conclusion is good; Premise list is good!  3 premise(s) loaded
168.     Conclusion and premise(s) are good => List Updated
169.
170.     Processing:  issue = General Issue ^ is_exterior_damaged = n : repair = No damage/fault detected
171.     Conclusion is good; Premise list is good!  2 premise(s) loaded
172.     Conclusion and premise(s) are good => List Updated
```

```
173.
174.
175.      Knowledge Base finished Loading.
176.      42 items were loaded into the KnowledgeBase
177.      <CR/Enter> to continue  List of variables: has_issue, is_starting, has_fuel, has_voltage,
   is_ignition_coil_damaged, is_distributor_cap_damaged, is_timing_belt_damaged, is_making_noise, is_noisy_while_driving,
   is_ticking_noise, is_hiccup_noise, is_air_filter_dirty, is_exhaust_blocked, are_wheel_bearings_damaged,
   are_tires_bald, is_truck, has_items_in_truck_bed, has_items_on_roof, is_overheating, has_coolant,
   is_water_pump_broken, is_oil_low_or_dirty, has_nonfunctional_electronics, does_ac_power_on, does_ac_blow_cold,
   has_nonfunctional_headlights, is_ac_fuse_intact, are_ac_wires_connected, are_therm_settings_correct,
   is_evaporator_coil_frozen, is_air_filter_dirty, is_nonfunct_light_fuse_intact, are_nonfunct_light_wires_conn,
   has_burning_plastic_smell, is_radio_working, is_radio_fuse_intact, are_radio_wires_connected, does_wheel_turn,
   is_power_steering_fuse_intact, has_power_steering_fluid, are_tires_deflated, has_piercing_object,
   is_obj_inch_away_from_edge, is_recent_temp_change, is_exterior_damaged, is_damage_cosmetic,
178.      Number of variables: 46
179.      Do you want to display the knowledge base (y/n)? 1. IF has_issue THEN issue
180.      2. IF issue THEN repair
181.      3. IF has_issue AND is_starting THEN issue
182.      4. IF issue AND has_fuel THEN repair
183.      5. IF issue AND has_fuel AND has_voltage THEN repair
184.      6. IF issue AND has_fuel AND has_voltage AND is_ignition_coil_damaged THEN repair
185.      7. IF issue AND has_fuel AND has_voltage AND is_ignition_coil_damaged AND is_distributor_cap_damaged THEN repair
186.      8. IF issue AND has_fuel AND has_voltage AND is_ignition_coil_damaged AND is_distributor_cap_damaged AND
   is_timing_belt_damaged THEN repair
187.      9. IF has_issue AND is_starting AND is_making_noise THEN issue
188.      10. IF issue AND is_noisy_while_driving AND are_wheel_bearings_damaged THEN repair
189.      11. IF issue AND is_noisy_while_driving AND are_wheel_bearings_damaged AND are_tires_bald THEN repair
190.      12. IF issue AND is_noisy_while_driving AND are_wheel_bearings_damaged AND are_tires_bald AND has_items_on_roof
   THEN repair
191.      13. IF issue AND is_noisy_while_driving AND are_wheel_bearings_damaged AND are_tires_bald AND has_items_on_roof
   AND is_truck AND has_items_in_truck_bed THEN repair
192.      14. IF issue AND is_noisy_while_driving AND is_ticking_noise AND is_hiccup_noise AND is_air_filter_dirty THEN
   repair
193.      15. IF issue AND is_noisy_while_driving AND is_ticking_noise AND is_hiccup_noise AND is_air_filter_dirty AND
   is_exhaust_blocked THEN repair
194.      16. IF has_issue AND is_starting AND is_making_noise AND is_overheating THEN issue
195.      17. IF issue AND has_coolant THEN repair
196.      18. IF issue AND has_coolant AND is_water_pump_broken THEN repair
197.      19. IF issue AND has_coolant AND is_water_pump_broken AND is_oil_low_or_dirty THEN repair
198.      20. IF has_issue AND is_starting AND is_making_noise AND is_overheating AND has_nonfunctional_electronics THEN
   issue
```

```
199.     21. IF issue AND does_ac_power_on AND is_ac_fuse_intact THEN repair
200.     22. IF issue AND does_ac_power_on AND is_ac_fuse_intact AND are_ac_wires_connected THEN repair
201.     23. IF issue AND does_ac_power_on AND does_ac_blow_cold AND are_therm_settings_correct THEN repair
202.     24. IF issue AND does_ac_power_on AND does_ac_blow_cold AND are_therm_settings_correct AND
   is_evaporator_coil_frozen THEN repair
203.     25. IF issue AND does_ac_power_on AND does_ac_blow_cold AND are_therm_settings_correct AND
   is_evaporator_coil_frozen AND is_air_filter_dirty THEN repair
204.     26. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND
   is_nonfunct_light_fuse_intact THEN repair
205.     27. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND
   is_nonfunct_light_fuse_intact AND are_nonfunct_light_wires_conn THEN repair
206.     28. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND
   has_burning_plastic_smell THEN repair
207.     29. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND
   has_burning_plastic_smell AND is_radio_working AND is_radio_fuse_intact THEN repair
208.     30. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND
   has_burning_plastic_smell AND is_radio_working AND is_radio_fuse_intact AND are_radio_wires_connected THEN repair
209.     31. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND
   has_burning_plastic_smell AND is_radio_working AND is_radio_fuse_intact AND are_radio_wires_connected THEN repair
210.     32. IF has_issue AND is_starting AND is_making_noise AND is_overheating AND has_nonfunctional_electronics AND
   does_wheel_turn THEN issue
211.     33. IF issue AND is_power_steering_fuse_intact THEN repair
212.     34. IF issue AND is_power_steering_fuse_intact AND has_power_steering_fluid THEN repair
213.     35. IF has_issue AND is_starting AND is_making_noise AND is_overheating AND has_nonfunctional_electronics AND
   does_wheel_turn AND are_tires_deflated THEN issue
214.     36. IF issue AND has_piercing_object AND is_obj_inch_away_from_edge THEN repair
215.     37. IF issue AND has_piercing_object AND is_obj_inch_away_from_edge THEN repair
216.     38. IF issue AND has_piercing_object AND is_recent_temp_change THEN repair
217.     39. IF has_issue AND is_starting AND is_making_noise AND is_overheating AND has_nonfunctional_electronics AND
   does_wheel_turn AND are_tires_deflated THEN issue
218.     40. IF issue AND is_exterior_damaged AND is_damage_cosmetic THEN repair
219.     41. IF issue AND is_exterior_damaged AND is_damage_cosmetic THEN repair
220.     42. IF issue AND is_exterior_damaged THEN repair
221.     Please enter a conclusion to solve (values can be: issue, repair):
222.     You entered: issue
223.     Is there an issue with the vehicle? (y/n):
224.     You entered: y
225.     Does the car start? (y/n):
226.     You entered: y
227.     Is the car making an unusual noise? (y/n):
228.     You entered: n
```

```
Does the temperature gauge show as overheating? (y/n):
You entered: n
Are any electronics not functional? (y/n):
You entered: n
Does the steering wheel turn? (y/n):
You entered: y
Are any of the tires visibly deflated or is the low pressure light on? (y/n):
You entered: y

Result is: Tire Issue
Conclusion is valid.
Creating knowledge base instance...


Now running forward chain
Processing has_issue
Processing issue
The final conclusion is - issue - with a value of: Tire Issue
```

Sample Run Two - vehicle repair recommendation with dead battery.

1. Welcome to the Automobile Diagnostic Program.
2. Authors: David Torrente (dat54@txstate.edu), Randall Henderson (rrh93@txstate.edu), Borislav Sabotinov (bss64@txstate.edu).
3.
4.
5. Creating knowledge base instance...
6. Processing: has_issue = n : issue = No issue
7. Conclusion is good; Premise list is good! 1 premise(s) loaded
8. Conclusion and premise(s) are good => List Updated
9.
10. Processing: issue = No issue : repair = No Repair Required
11. Conclusion is good; Premise list is good! 1 premise(s) loaded
12. Conclusion and premise(s) are good => List Updated
13.
14. Processing: has_issue = y ^ is_starting = n : issue = Failure to Start
15. Conclusion is good; Premise list is good! 2 premise(s) loaded
16. Conclusion and premise(s) are good => List Updated
17.
18. Processing: issue = Failure to Start ^ has_fuel = n : repair = Insufficient Fuel, Add more fuel.
19. Conclusion is good; Premise list is good! 2 premise(s) loaded
20. Conclusion and premise(s) are good => List Updated
21.
22. Processing: issue = Failure to Start ^ has_fuel = y ^ has_voltage = n : repair = Dead Battery, Change the battery.

23. Conclusion is good; Premise list is good! 3 premise(s) loaded
24. Conclusion and premise(s) are good => List Updated
25.
26. Processing: issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = y : repair = Bad Ignition Coil, Replace faulty ignition coil.
27. Conclusion is good; Premise list is good! 4 premise(s) loaded
28. Conclusion and premise(s) are good => List Updated
29.
30. Processing: issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = n ^ is_distributor_cap_damaged = y : repair = Bad Distributor Cap, Replace faulty distributor cap.
31. Conclusion is good; Premise list is good! 5 premise(s) loaded
32. Conclusion and premise(s) are good => List Updated
33.
34. Processing: issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = n ^ is_distributor_cap_damaged = n ^ is_timing_belt_damaged = y : repair = Bad Timing Belt, Replace faulty timing belt.
35. Conclusion is good; Premise list is good! 6 premise(s) loaded
36. Conclusion and premise(s) are good => List Updated
37.
38. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = y : issue = Noise Issue
39. Conclusion is good; Premise list is good! 3 premise(s) loaded
40. Conclusion and premise(s) are good => List Updated
41.
42. Processing: issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = y : repair = Faulty bearings, replace wheel bearings
43. Conclusion is good; Premise list is good! 3 premise(s) loaded

44. Conclusion and premise(s) are good => List Updated
45.
46. Processing: issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = y : repair = Tires are worn, replace with new tires
47. Conclusion is good; Premise list is good! 4 premise(s) loaded
48. Conclusion and premise(s) are good => List Updated
49.
50. Processing: issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = n ^ has_items_on_roof = y : repair = Remove items from roof
51. Conclusion is good; Premise list is good! 5 premise(s) loaded
52. Conclusion and premise(s) are good => List Updated
53.
54. Processing: issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = n ^ has_items_on_roof = n ^ is_truck = y ^ has_items_in_truck_bed = y : repair = Remove items from truck bed
55. Conclusion is good; Premise list is good! 7 premise(s) loaded
56. Conclusion and premise(s) are good => List Updated
57.
58. Processing: issue = Noise Issue ^ is_noisy_while_driving = n ^ is_ticking_noise = n ^ is_hiccup_noise = y ^ is_air_filter_dirty = y : repair = Replace dirty air filter with clean one
59. Conclusion is good; Premise list is good! 5 premise(s) loaded
60. Conclusion and premise(s) are good => List Updated
61.
62. Processing: issue = Noise Issue ^ is_noisy_while_driving = n ^ is_ticking_noise = n ^ is_hiccup_noise = y ^ is_air_filter_dirty = n ^ is_exhaust_blocked = y : repair = Remove object obstructing exhaust

63. Conclusion is good; Premise list is good! 6 premise(s) loaded
64. Conclusion and premise(s) are good => List Updated
65.
66. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = y : issue = Overheating Issue
67. Conclusion is good; Premise list is good! 4 premise(s) loaded
68. Conclusion and premise(s) are good => List Updated
69.
70. Processing: issue = Overheating Issue ^ has_coolant = n : repair = Low Coolant, add coolant.
71. Conclusion is good; Premise list is good! 2 premise(s) loaded
72. Conclusion and premise(s) are good => List Updated
73.
74. Processing: issue = Overheating Issue ^ has_coolant = y ^ is_water_pump_broken = y : repair = Defective Water Pump, replace water pump.
75. Conclusion is good; Premise list is good! 3 premise(s) loaded
76. Conclusion and premise(s) are good => List Updated
77.
78. Processing: issue = Overheating Issue ^ has_coolant = y ^ is_water_pump_broken = n ^ is_oil_low_or_dirty = y : repair = Low or Dirty Oil, replace oil
79. Conclusion is good; Premise list is good! 4 premise(s) loaded
80. Conclusion and premise(s) are good => List Updated
81.
82. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = y : issue = Electronics Issue
83. Conclusion is good; Premise list is good! 5 premise(s) loaded
84. Conclusion and premise(s) are good => List Updated
85.

86. Processing: issue = Electronics Issue ^ does_ac_power_on = n ^ is_ac_fuse_intact = n : repair = Replace defective AC fuse
87. Conclusion is good; Premise list is good! 3 premise(s) loaded
88. Conclusion and premise(s) are good => List Updated
89.
90. Processing: issue = Electronics Issue ^ does_ac_power_on = n ^ is_ac_fuse_intact = y ^ are_ac_wires_connected = n : repair = Connect AC wires properly
91. Conclusion is good; Premise list is good! 4 premise(s) loaded
92. Conclusion and premise(s) are good => List Updated
93.
94. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = n : repair = Fix thermostat settings
95. Conclusion is good; Premise list is good! 4 premise(s) loaded
96. Conclusion and premise(s) are good => List Updated
97.
98. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = y ^ is_evaporator_coil_frozen = y : repair = Defrost evaporator coil
99. Conclusion is good; Premise list is good! 5 premise(s) loaded
100. Conclusion and premise(s) are good => List Updated
101.
102. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = y ^ is_evaporator_coil_frozen = n ^ is_air_filter_dirty = y : repair = Replace dirty air filter with clean one
103. Conclusion is good; Premise list is good! 6 premise(s) loaded
104. Conclusion and premise(s) are good => List Updated
105.

106. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = y ^ is_nonfunct_light_fuse_intact = n : repair = Replace defective light fuse

107. Conclusion is good; Premise list is good! 5 premise(s) loaded

108. Conclusion and premise(s) are good => List Updated

109.

110. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = y ^ is_nonfunct_light_fuse_intact = y ^ are_nonfunct_light_wires_conn = n : repair = Connect light wires properly

111. Conclusion is good; Premise list is good! 6 premise(s) loaded

112. Conclusion and premise(s) are good => List Updated

113.

114. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = y : repair = Serious issue and possible short circuit, replace wiring and fuses.

115. Conclusion is good; Premise list is good! 5 premise(s) loaded

116. Conclusion and premise(s) are good => List Updated

117.

118. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = n : repair = Replace defective radio fuse

119. Conclusion is good; Premise list is good! 7 premise(s) loaded

120. Conclusion and premise(s) are good => List Updated

121.

122. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = y ^ are_radio_wires_connected = n : repair = Connect radio wires properly

123. Conclusion is good; Premise list is good! 8 premise(s) loaded

124. Conclusion and premise(s) are good => List Updated

125.

126. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = y ^ are_radio_wires_connected = y : repair = Wiring or installation issue, replace radio and wiring

127. Conclusion is good; Premise list is good! 8 premise(s) loaded

128. Conclusion and premise(s) are good => List Updated

129.

130. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = n ^ does_wheel_turn = n : issue = Steering Issue

131. Conclusion is good; Premise list is good! 6 premise(s) loaded

132. Conclusion and premise(s) are good => List Updated

133.

134. Processing: issue = Steering Issue ^ is_power_steering_fuse_intact = n : repair = Replace power steering fuse

135. Conclusion is good; Premise list is good! 2 premise(s) loaded

136. Conclusion and premise(s) are good => List Updated

137.

138. Processing: issue = Steering Issue ^ is_power_steering_fuse_intact = y ^ has_power_steering_fluid = n : repair = Power steering failure, add power steering fluid

139. Conclusion is good; Premise list is good! 3 premise(s) loaded

140. Conclusion and premise(s) are good => List Updated

141.

142. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = n ^ does_wheel_turn = y ^ are_tires_deflated = y : issue = Tire Issue

143. Conclusion is good; Premise list is good! 7 premise(s) loaded

144. Conclusion and premise(s) are good => List Updated

145.

146. Processing: issue = Tire Issue ^ has_piercing_object = y ^ is_obj_inch_away_from_edge = n : repair = Replace deflated tire

147. Conclusion is good; Premise list is good! 3 premise(s) loaded

148. Conclusion and premise(s) are good => List Updated

149.

150. Processing: issue = Tire Issue ^ has_piercing_object = y ^ is_obj_inch_away_from_edge = y : repair = Apply tire patch kit to deflated tire

151. Conclusion is good; Premise list is good! 3 premise(s) loaded

152. Conclusion and premise(s) are good => List Updated

153.

154. Processing: issue = Tire Issue ^ has_piercing_object = n ^ is_recent_temp_change = y : repair = Low tire pressure, add air to deflated tires

155. Conclusion is good; Premise list is good! 3 premise(s) loaded

156. Conclusion and premise(s) are good => List Updated

157.

158. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = n ^ does_wheel_turn = y ^ are_tires_deflated = n : issue = General Issue

159. Conclusion is good; Premise list is good! 7 premise(s) loaded

160. Conclusion and premise(s) are good => List Updated

161.

162. Processing: issue = General Issue ^ is_exterior_damaged = y ^ is_damage_cosmetic = y : repair = Repaint the scratched part

163. Conclusion is good; Premise list is good! 3 premise(s) loaded

164. Conclusion and premise(s) are good => List Updated

165.

166. Processing: issue = General Issue ^ is_exterior_damaged = y ^ is_damage_cosmetic = n : repair = Replace the damaged vehicle body part

167. Conclusion is good; Premise list is good! 3 premise(s) loaded

168. Conclusion and premise(s) are good => List Updated

169.

170. Processing: issue = General Issue ^ is_exterior_damaged = n : repair = No damage/fault detected

171. Conclusion is good; Premise list is good! 2 premise(s) loaded

172. Conclusion and premise(s) are good => List Updated

173.

174.

175. Knowledge Base finished Loading.

176. 42 items were loaded into the KnowledgeBase

177. <CR/Enter> to continue List of variables: has_issue, is_starting, has_fuel, has_voltage, is_ignition_coil_damaged, is_distributor_cap_damaged, is_timing_belt_damaged, is_making_noise, is_noisy_while_driving, is_ticking_noise, is_hiccup_noise, is_air_filter_dirty, is_exhaust_blocked, are_wheel_bearings_damaged, are_tires_bald, is_truck, has_items_in_truck_bed, has_items_on_roof, is_overheating, has_coolant, is_water_pump_broken, is_oil_low_or_dirty, has_nonfunctional_electronics, does_ac_power_on, does_ac_blow_cold, has_nonfunctional_headlights, is_ac_fuse_intact, are_ac_wires_connected, are_therm_settings_correct, is_evaporator_coil_frozen, is_air_filter_dirty, is_nonfunct_light_fuse_intact, are_nonfunct_light_wires_conn, has_burning_plastic_smell, is_radio_working, is_radio_fuse_intact, are_radio_wires_connected, does_wheel_turn, is_power_steering_fuse_intact, has_power_steering_fluid, are_tires_deflated, has_piercing_object, is_obj_inch_away_from_edge, is_recent_temp_change, is_exterior_damaged, is_damage_cosmetic,

178. Number of variables: 46

179. Do you want to display the knowledge base (y/n)? Please enter a conclusion to solve (values can be: issue, repair):

180. You entered: repair

181. Is there an issue with the vehicle? (y/n):

182. You entered: y

183. Does the car start? (y/n):

184. You entered: n

185. Is there enough fuel? (y/n):

186. You entered: y

187. Does the battery have enough voltage? (y/n):

188. You entered: n

189.

190. Result is: Dead Battery, Change the battery.

191. Conclusion is valid.

192. Creating knowledge base instance...

193.

194.

195. Now running forward chain

196. Processing has_issue

197. Processing issue

198. Processing repair

199. The final conclusion is - repair - with a value of: Dead Battery, Change the battery.

Sample Run Three – Replace Water Pump Repair Recommendation

1. Welcome to the Automobile Diagnostic Program.
2. Authors: David Torrente (dat54@txstate.edu), Randall Henderson (rrh93@txstate.edu), Borislav Sabotinov (bss64@txstate.edu).

3. Creating knowledge base instance...
4. Processing: has_issue = n : issue = No issue
5. Conclusion is good; Premise list is good! 1 premise(s) loaded
6. Conclusion and premise(s) are good => List Updated

7. Processing: issue = No issue : repair = No Repair Required
8. Conclusion is good; Premise list is good! 1 premise(s) loaded
9. Conclusion and premise(s) are good => List Updated

10. Processing: has_issue = y ^ is_starting = n : issue = Failure to Start
11. Conclusion is good; Premise list is good! 2 premise(s) loaded
12. Conclusion and premise(s) are good => List Updated

13. Processing: issue = Failure to Start ^ has_fuel = n : repair = Insufficient Fuel, Add more fuel.
14. Conclusion is good; Premise list is good! 2 premise(s) loaded
15. Conclusion and premise(s) are good => List Updated

16. Processing: issue = Failure to Start ^ has_fuel = y ^ has_voltage = n : repair = Dead Battery, Change the battery.
17. Conclusion is good; Premise list is good! 3 premise(s) loaded
18. Conclusion and premise(s) are good => List Updated

19. Processing: issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = y : repair = Bad Ignition Coil, Replace faulty ignition coil.
20. Conclusion is good; Premise list is good! 4 premise(s) loaded
21. Conclusion and premise(s) are good => List Updated

22. Processing: issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = n ^ is_distributor_cap_damaged = y : repair = Bad Distributor Cap, Replace faulty distributor cap.
23. Conclusion is good; Premise list is good! 5 premise(s) loaded
24. Conclusion and premise(s) are good => List Updated

25. Processing: issue = Failure to Start ^ has_fuel = y ^ has_voltage = y ^ is_ignition_coil_damaged = n ^ is_distributor_cap_damaged = n ^ is_timing_belt_damaged = y : repair = Bad Timing Belt, Replace faulty timing belt.
26. Conclusion is good; Premise list is good! 6 premise(s) loaded
27. Conclusion and premise(s) are good => List Updated

28. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = y : issue = Noise Issue
29. Conclusion is good; Premise list is good! 3 premise(s) loaded
30. Conclusion and premise(s) are good => List Updated

31. Processing: issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = y : repair = Faulty bearings, replace wheel bearings
32. Conclusion is good; Premise list is good! 3 premise(s) loaded
33. Conclusion and premise(s) are good => List Updated

34. Processing: issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = y : repair = Tires are worn, replace with new tires
35. Conclusion is good; Premise list is good! 4 premise(s) loaded
36. Conclusion and premise(s) are good => List Updated

37. Processing: issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = n ^ has_items_on_roof = y : repair = Remove items from roof
38. Conclusion is good; Premise list is good! 5 premise(s) loaded
39. Conclusion and premise(s) are good => List Updated

40. Processing: issue = Noise Issue ^ is_noisy_while_driving = y ^ are_wheel_bearings_damaged = n ^ are_tires_bald = n ^ has_items_on_roof = n ^ is_truck = y ^ has_items_in_truck_bed = y : repair = Remove items from truck bed
41. Conclusion is good; Premise list is good! 7 premise(s) loaded
42. Conclusion and premise(s) are good => List Updated

43. Processing: issue = Noise Issue ^ is_noisy_while_driving = n ^ is_ticking_noise = n ^ is_hiccup_noise = y ^ is_air_filter_dirty = y : repair = Replace dirty air filter with clean one
44. Conclusion is good; Premise list is good! 5 premise(s) loaded
45. Conclusion and premise(s) are good => List Updated

46. Processing: issue = Noise Issue ^ is_noisy_while_driving = n ^ is_ticking_noise = n ^ is_hiccup_noise = y ^ is_air_filter_dirty = n ^ is_exhaust_blocked = y : repair = Remove object obstructing exhaust
47. Conclusion is good; Premise list is good! 6 premise(s) loaded
48. Conclusion and premise(s) are good => List Updated

49. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = y : issue = Overheating Issue
50. Conclusion is good; Premise list is good! 4 premise(s) loaded
51. Conclusion and premise(s) are good => List Updated

52. Processing: issue = Overheating Issue ^ has_coolant = n : repair = Low Coolant, add coolant.
53. Conclusion is good; Premise list is good! 2 premise(s) loaded
54. Conclusion and premise(s) are good => List Updated

55. Processing: issue = Overheating Issue ^ has_coolant = y ^ is_water_pump_broken = y : repair = Defective Water Pump, replace water pump.
56. Conclusion is good; Premise list is good! 3 premise(s) loaded
57. Conclusion and premise(s) are good => List Updated

58. Processing: issue = Overheating Issue ^ has_coolant = y ^ is_water_pump_broken = n ^ is_oil_low_or_dirty = y : repair = Low or Dirty Oil, replace oil
59. Conclusion is good; Premise list is good! 4 premise(s) loaded
60. Conclusion and premise(s) are good => List Updated

61. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = y : issue = Electronics Issue
62. Conclusion is good; Premise list is good! 5 premise(s) loaded
63. Conclusion and premise(s) are good => List Updated

64. Processing: issue = Electronics Issue ^ does_ac_power_on = n ^ is_ac_fuse_intact = n : repair = Replace defective AC fuse
65. Conclusion is good; Premise list is good! 3 premise(s) loaded
66. Conclusion and premise(s) are good => List Updated

67. Processing: issue = Electronics Issue ^ does_ac_power_on = n ^ is_ac_fuse_intact = y ^ are_ac_wires_connected = n : repair = Connect AC wires properly
68. Conclusion is good; Premise list is good! 4 premise(s) loaded
69. Conclusion and premise(s) are good => List Updated

70. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = n : repair = Fix thermostat settings
71. Conclusion is good; Premise list is good! 4 premise(s) loaded
72. Conclusion and premise(s) are good => List Updated

73. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = y ^ is_evaporator_coil_frozen = y : repair = Defrost evaporator coil
74. Conclusion is good; Premise list is good! 5 premise(s) loaded

75. Conclusion and premise(s) are good => List Updated

76. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = n ^ are_therm_settings_correct = y ^ is_evaporator_coil_frozen = n ^ is_air_filter_dirty = y : repair = Replace dirty air filter with clean one
77. Conclusion is good; Premise list is good! 6 premise(s) loaded
78. Conclusion and premise(s) are good => List Updated

79. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = y ^ is_nonfunct_light_fuse_intact = n : repair = Replace defective light fuse
80. Conclusion is good; Premise list is good! 5 premise(s) loaded
81. Conclusion and premise(s) are good => List Updated

82. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = y ^ is_nonfunct_light_fuse_intact = y ^ are_nonfunct_light_wires_conn = n : repair = Connect light wires properly
83. Conclusion is good; Premise list is good! 6 premise(s) loaded
84. Conclusion and premise(s) are good => List Updated

85. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = y : repair = Serious issue and possible short circuit, replace wiring and fuses.
86. Conclusion is good; Premise list is good! 5 premise(s) loaded
87. Conclusion and premise(s) are good => List Updated

88. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = n : repair = Replace defective radio fuse
89. Conclusion is good; Premise list is good! 7 premise(s) loaded
90. Conclusion and premise(s) are good => List Updated

91. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = y ^ are_radio_wires_connected = n : repair = Connect radio wires properly
92. Conclusion is good; Premise list is good! 8 premise(s) loaded
93. Conclusion and premise(s) are good => List Updated

94. Processing: issue = Electronics Issue ^ does_ac_power_on = y ^ does_ac_blow_cold = y ^ has_nonfunctional_headlights = n ^ has_burning_plastic_smell = n ^ is_radio_working = n ^ is_radio_fuse_intact = y ^ are_radio_wires_connected = y : repair = Wiring or installation issue, replace radio and wiring
95. Conclusion is good; Premise list is good! 8 premise(s) loaded
96. Conclusion and premise(s) are good => List Updated

97. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = n ^ does_wheel_turn = n : issue = Steering Issue
98. Conclusion is good; Premise list is good! 6 premise(s) loaded
99. Conclusion and premise(s) are good => List Updated

100. Processing: issue = Steering Issue ^ is_power_steering_fuse_intact = n : repair = Replace power steering fuse
101. Conclusion is good; Premise list is good! 2 premise(s) loaded
102. Conclusion and premise(s) are good => List Updated

103. Processing: issue = Steering Issue ^ is_power_steering_fuse_intact = y ^ has_power_steering_fluid = n : repair = Power steering failure, add power steering fluid
104. Conclusion is good; Premise list is good! 3 premise(s) loaded
105. Conclusion and premise(s) are good => List Updated

106. Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = n ^ does_wheel_turn = y ^ are_tires_deflated = y : issue = Tire Issue
107. Conclusion is good; Premise list is good! 7 premise(s) loaded
108. Conclusion and premise(s) are good => List Updated

Processing: issue = Tire Issue ^ has_piercing_object = y ^ is_obj_inch_away_from_edge = n : repair = Replace deflated tire
Conclusion is good; Premise list is good! 3 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing: issue = Tire Issue ^ has_piercing_object = y ^ is_obj_inch_away_from_edge = y : repair = Apply tire patch kit to deflated tire
Conclusion is good; Premise list is good! 3 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing: issue = Tire Issue ^ has_piercing_object = n ^ is_recent_temp_change = y : repair = Low tire pressure, add air to deflated tires
Conclusion is good; Premise list is good! 3 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing: has_issue = y ^ is_starting = y ^ is_making_noise = n ^ is_overheating = n ^ has_nonfunctional_electronics = n ^ does_wheel_turn = y ^ are_tires_deflated = n : issue = General Issue
Conclusion is good; Premise list is good! 7 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing: issue = General Issue ^ is_exterior_damaged = y ^ is_damage_cosmetic = y : repair = Repaint the scratched part
Conclusion is good; Premise list is good! 3 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing: issue = General Issue ^ is_exterior_damaged = y ^ is_damage_cosmetic = n : repair = Replace the damaged vehicle body part
Conclusion is good; Premise list is good! 3 premise(s) loaded
Conclusion and premise(s) are good => List Updated

Processing: issue = General Issue ^ is_exterior_damaged = n : repair = No damage/fault detected
Conclusion is good; Premise list is good! 2 premise(s) loaded
Conclusion and premise(s) are good => List Updated

130. Knowledge Base finished Loading.
131. 42 items were loaded into the KnowledgeBase
132. <CR/Enter> to continue List of variables: has_issue, is_starting, has_fuel, has_voltage, is_ignition_coil_damaged, is_distributor_cap_damaged, is_timing_belt_damaged, is_making_noise, is_noisy_while_driving, is_ticking_noise, is_hiccup_noise, is_air_filter_dirty, is_exhaust_blocked, are_wheel_bearings_damaged, are_tires_bald, is_truck, has_items_in_truck_bed, has_items_on_roof, is_overheating, has_coolant, is_water_pump_broken, is_oil_low_or_dirty, has_nonfunctional_electronics, does_ac_power_on, does_ac_blow_cold, has_nonfunctional_headlights, is_ac_fuse_intact, are_ac_wires_connected, are_therm_settings_correct, is_evaporator_coil_frozen, is_air_filter_dirty, is_nonfunct_light_fuse_intact, are_nonfunct_light_wires_conn, has_burning_plastic_smell, is_radio_working, is_radio_fuse_intact, are_radio_wires_connected, does_wheel_turn, is_power_steering_fuse_intact, has_power_steering_fluid, are_tires_deflated, has_piercing_object, is_obj_inch_away_from_edge, is_recent_temp_change, is_exterior_damaged, is_damage_cosmetic,
133. Number of variables: 46
134. Do you want to display the knowledge base (y/n)? 1. IF has_issue THEN issue
3. IF issue THEN repair
4. IF has_issue AND is_starting THEN issue
5. IF issue AND has_fuel THEN repair
6. IF issue AND has_fuel AND has_voltage THEN repair
7. IF issue AND has_fuel AND has_voltage AND is_ignition_coil_damaged THEN repair
8. IF issue AND has_fuel AND has_voltage AND is_ignition_coil_damaged AND is_distributor_cap_damaged THEN repair
9. IF issue AND has_fuel AND has_voltage AND is_ignition_coil_damaged AND is_distributor_cap_damaged AND is_timing_belt_damaged THEN repair
10. IF has_issue AND is_starting AND is_making_noise THEN issue
11. IF issue AND is_noisy_while_driving AND are_wheel_bearings_damaged THEN repair
12. IF issue AND is_noisy_while_driving AND are_wheel_bearings_damaged AND are_tires_bald THEN repair
13. IF issue AND is_noisy_while_driving AND are_wheel_bearings_damaged AND are_tires_bald AND has_items_on_roof THEN repair
14. IF issue AND is_noisy_while_driving AND are_wheel_bearings_damaged AND are_tires_bald AND has_items_on_roof AND is_truck AND has_items_in_truck_bed THEN repair
15. IF issue AND is_noisy_while_driving AND is_ticking_noise AND is_hiccup_noise AND is_air_filter_dirty THEN repair
16. IF issue AND is_noisy_while_driving AND is_ticking_noise AND is_hiccup_noise AND is_air_filter_dirty AND is_exhaust_blocked THEN repair
17. IF has_issue AND is_starting AND is_making_noise AND is_overheating THEN issue
18. IF issue AND has_coolant THEN repair
19. IF issue AND has_coolant AND is_water_pump_broken THEN repair
20. IF issue AND has_coolant AND is_water_pump_broken AND is_oil_low_or_dirty THEN repair
21. IF has_issue AND is_starting AND is_making_noise AND is_overheating AND has_nonfunctional_electronics THEN issue

22. IF issue AND does_ac_power_on AND is_ac_fuse_intact THEN repair
23. IF issue AND does_ac_power_on AND is_ac_fuse_intact AND are_ac_wires_connected THEN repair
24. IF issue AND does_ac_power_on AND does_ac_blow_cold AND are_therm_settings_correct THEN repair
25. IF issue AND does_ac_power_on AND does_ac_blow_cold AND are_therm_settings_correct AND is_evaporator_coil_frozen THEN repair
26. IF issue AND does_ac_power_on AND does_ac_blow_cold AND are_therm_settings_correct AND is_evaporator_coil_frozen AND is_air_filter_dirty THEN repair
27. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND is_nonfunct_light_fuse_intact THEN repair
28. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND is_nonfunct_light_fuse_intact AND are_nonfunct_light_wires_conn THEN repair
29. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND has_burning_plastic_smell THEN repair
30. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND has_burning_plastic_smell AND is_radio_working AND is_radio_fuse_intact THEN repair
31. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND has_burning_plastic_smell AND is_radio_working AND is_radio_fuse_intact AND are_radio_wires_connected THEN repair
32. IF issue AND does_ac_power_on AND does_ac_blow_cold AND has_nonfunctional_headlights AND has_burning_plastic_smell AND is_radio_working AND is_radio_fuse_intact AND are_radio_wires_connected THEN repair
33. IF has_issue AND is_starting AND is_making_noise AND is_overheating AND has_nonfunctional_electronics AND does_wheel_turn THEN issue
34. IF issue AND is_power_steering_fuse_intact THEN repair
35. IF issue AND is_power_steering_fuse_intact AND has_power_steering_fluid THEN repair
36. IF has_issue AND is_starting AND is_making_noise AND is_overheating AND has_nonfunctional_electronics AND does_wheel_turn AND are_tires_deflated THEN issue
37. IF issue AND has_piercing_object AND is_obj_inch_away_from_edge THEN repair
38. IF issue AND has_piercing_object AND is_obj_inch_away_from_edge THEN repair
39. IF issue AND has_piercing_object AND is_recent_temp_change THEN repair
40. IF has_issue AND is_starting AND is_making_noise AND is_overheating AND has_nonfunctional_electronics AND does_wheel_turn AND are_tires_deflated THEN issue
41. IF issue AND is_exterior_damaged AND is_damage_cosmetic THEN repair
42. IF issue AND is_exterior_damaged AND is_damage_cosmetic THEN repair
43. IF issue AND is_exterior_damaged THEN repair

135. Please enter a conclusion to solve (values can be: issue, repair):
136. You entered: repair
137. Is there an issue with the vehicle? (y/n):
138. You entered: y
139. Does the car start? (y/n):
140. You entered: y

Is the car making an unusual noise? (y/n):
You entered: n
Does the temperature gauge show as overheating? (y/n):
You entered: y
Is there sufficient coolant in the engine? (y/n):
You entered: y
Is the water pump broken? (y/n):
You entered: y

Result is: Defective Water Pump, replace water pump.
Conclusion is valid.
Creating knowledge base instance...

Now running forward chain
Processing has_issue
Processing issue
Processing repair
The final conclusion is - repair - with a value of: Defective Water Pump, replace water pump.