Aircraft Decision Tree (acdt)

Professor Caleb Fowler

June 7, 2020

Problem.

This assignment implements a hypothetical troubleshooting tree for possible engine failure in light aircraft. DO NOT FOLLOW IN A REAL EMERGENCY! A decision tree is an Artificial Intelligence method for arriving at a decision based upon a collection of IF statements. The computer asks a question and responds, based upon the answers. The computer only asks the relevant question based upon the user's answers (ie you don't ask all the questions and then try to sort it out.)

Requirements.

This program is menu driven. It presents four menu options, representing 3 potential emergency situations: Communications Failure, Engine Failure, In-flight Icing. Quit the program is the final menu option. See figure 1 for an example menu.

Menu Options.
1. Communications Failure
2. Engine Failure
3. In-Flight Icing
4. Quit

Figure 1: Allowable menu options for this program.

1. Communications Failure
   - This option only leads to 1 element, "Switch to Alternate Radio."
2. Engine Failure
   - This activates the decision tree. This is basically a collection of IF statements arranged in a hierarchy. This is shown on the diagram below.
3. In-Flight Icing
   - Prompt the pilot for the estimated amount of ice on the wings (0 mm to 10 mm). Aircraft will not fly with more than 10 mm of ice (and pilots will not be operating this system).
   - Range check this to make sure only 0 mm to 10 mm can be entered.
   - The amount of ice determines how much power to use to energize the deicing boots on the wings.
     – Less then 1mm, 5% power
     – 1 - 5 mm, 20% power
     – 5.1 - 9 mm, 65% power
     – 9.1 - 10 mm, 100% power

- Only ask questions relevant to the situation and previous answers. Do not ask for all the inputs at once (too stressful for the pilot!
- Display the results of all computations.
- Make your program output as easy to read as possible.

- Don't forget to incorporate these elements from the style guide:
  - Comments
  - Source file header.
  - Program Greeting - this is NOT the same as the menu!
  - Constant variable style if applicable.
  - No global variables or global variable look a-likes.

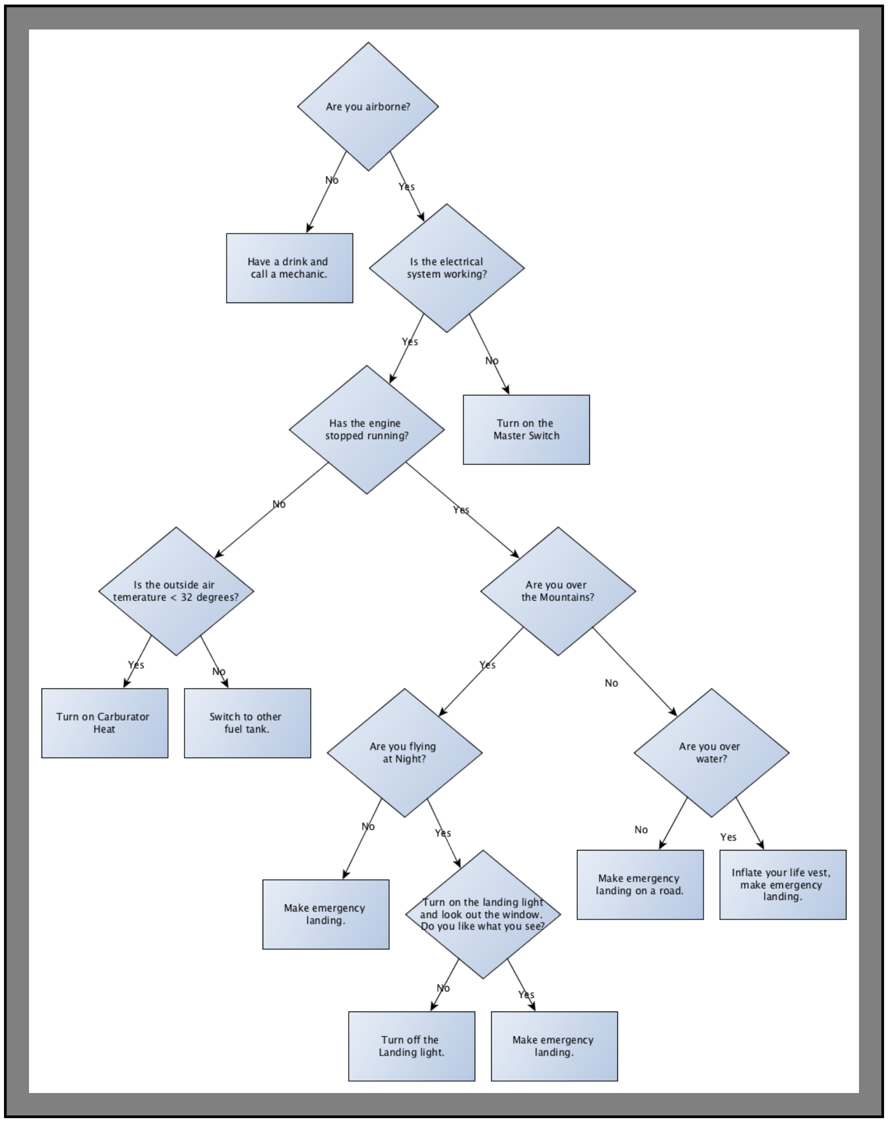

Are you airborne?
No
Yes
Have a drink and call a mechanic.
Is the electrical system working?
Yes
No
Has the engine stopped running?
Turn on the Master Switch
No
Yes
Is the outside air temerature < 32 degrees?
Are you over the Mountains?
Yes
No
Turn on Carburator Heat
Switch to other fuel tank.
Yes
No
Are you flying at Night?
Are you over water?
No
Yes
No
Yes
Make emergency landing.
Turn on the landing light and look out the window. Do you like what you see?
Make emergency landing on a road.
Inflate your life vest, make emergency landing.
No
Yes
Turn off the Landing light.
Make emergency landing.

Figure 2: This is the engine failure menu selection.

Specification Bundles.

Code elements from the specification bundles to control the maximum potential grade you can get for this assignment. The more work you do, the better grade you can get. This is the starting point for your grade.

"C Grade" Specification Bundle.

Code sections corresponding to the 4 menu sections above. Don't forget the description. Make this a number driven menu.

- ☐ // Specification C1 - Communications Option
  This makes the start of the communications option code.
- ☐ // Specification C2 - Engine Failure Option
  This makes the start of the engine failure option code.
- ☐ // Specification C3 - In-Flight Icing Option
  This makes the start of the in-flight icing option code.

"B Grade" Specification Bundle.

- ☐ // Specification B1 - Menu Input Validation
  Only allow the client to enter valid numbers for the main menu. Quit the program with invalid entries. You can just display a general error message for this specification.
- ☐ // Specification B2 - Icing Input Validation
  Only allow a valid range for the de-icing menu. Exit the program with invalid input.
- ☐ // Specification B3 - Date
  On the first line of the main menu, include the current date. You can get the system date and use that, or you can prompt the client for the date. Your choice.

Input validation is a big deal. It's the first line of defense against adversaries. However, we need to balance that against usability. Generally, the more specific the error message you can give the client, the better they will like your program. Usually this means more if tests. You can always add more error tests than specified in the homework.

"A Grade" Specification Bundle.

- ☐ // Specification A1 - Alpha Menu
  Change from a numeric menu to an alphabetic menu. See figure 3 for an example menu. Keep your numeric menu code - just comment it out. I only want 1 menu in operation.
- ☐ // Specification A2 - Menu Input Validation
  Only allow the client to enter valid letters for the main menu. Quit the program with invalid entries. You can just display a general error message for this specification. The replaces specification B1, so also like Specification A1 , just comment out the validation code which doesn't apply to a numeric menu.
- ☐ // Specification A3 - One Function
  I want to see at least one function in an "A" program. I don't care what you do with it, just write one. Don't forget the function prototype.

```
Alphabetic Menu Options.
  <C>ommunications Failure
  <E>ngine Failure
  <I>n-Flight Icing
  <Q>uit
```

Figure 3: Allowable menu options for this program in alphabetic format.

~~*Due Date.*~~

This assignment is due by 11:59 PM on Sunday on the date on the calendar in Canvas. All the assignments are open the first day of class and you can begin working on them immediately. I encourage you to start sooner rather than later with your homework, these always seem to take longer than you think.

~~*Late Work.*~~

If you miss the due date and time specified above, your work is late. Canvas records the date and time your homework upload COMPLETES. Late work is still acceptable, but it suffers a 1 letter grade penalty. You may turn late work in up until MONDAY 11:59 PM AFTER THE ASSIGNMENT WAS DUE. That is, you have 1 day to turn your work in - after that the Canvas drop box closes. Once Canvas closes I will not accept an assignment.

Pro-Tip: Get a bare bones copy of your code running and turn it in. Then go ahead and modify it, fix it and whatnot. Upload it with the same name when you finish. That way, if something unexpected happens, you have some working code turned in. Risk management, class, risk management. [1]

[1] If you really want to go pro, get some sort of version control system running (like Git).

~~*How to Turn in your Homework.*~~

I ONLY accept homework through the Canvas Dropbox. Do not add it to the submission comments or email it to me - I will not accept it. Turn homework in by uploading to the appropriate Canvas Dropbox folder. Save your homework as a .cpp file. Don't zip or otherwise compress your files. Do NOT split your file up into multiple files. I know that is a standard industry practice, but it just get's in the way for this class.

Create a file with the following naming format: W12345678.cpp (your w number). This allows me to sort the class in alphabetical order - don't stand out here! If you are having trouble submitting the assignment, email me immediately. Don't change your filename if you make multiple submissions - Canvas will keep track of them and download the latest one by default.

~~*Style Guide.*~~

All programs you write MUST have the following code and/or comments. Again, I look for these elements with my scripts, you want me to find them.

Comments.

Use white space and comments to make your code more readable. I run a program called cloc (count lines of code) which actually looks for this stuff.

End of line comments are only permitted with variable declarations. Full line comments are used everywhere else.

Specification Comments.

Specifications are bundled into groups: "A", "B", "C". You must meet the specifications of the lowest group before I will count the specifications for the highest group. For example, you must meet the "B" specifications before I will count the "A" specifications. If you miss one element of a specification bundle, that is the grade you will get for the assignment - regardless of how much extra work you do.

Use whole line comments for Specifications. Put the comment on the line above the start of the code implementing the Specification. If the same Specification code appears in more than 1 place, only comment the first place that Specification code appears. Number your Specifications according to the specification bundle and the specific specification you are using, also provide a very short description. DO NOT BUNCH ALL YOUR SPECIFICATIONS AT THE TOP OF THE SOURCE FILE. Example specification comment:

```
// Specification A2 - Display variables
Your code to do this starts here;
```

It's very important to get the specifications down correctly. If your specification code isn't commented, it doesn't count. I use the grep trick to find your specification code. Proper documentation is part of the solution, just like actually coding the solution is.

Compiler Warnings.

Compiler warnings are a potential problem. They are not tolerated in the production environment. In CISP 360 you can have them. I will deduct a small number of points. CISP 400 - I will deduct lots of points if compiler warnings appear. Make sure you compile with -Wall option. This is how you spot them.

C++ Libraries.

We are coding in C++, not C. Therefore, you must use the C++ libraries. The only time you can use the C libraries is if they haven't been ported to C++ (very, very rare).

Non-Standard Language Extensions.

Some compilers support unapproved extensions to the C++ syntax. These extensions are **unacceptable.** Unsupported extensions are compiler specific and non-portable. Do not use them in your programs.

Program Greeting.

Display a program greeting as soon as the program runs. This is a simple description of what the program does. Example:

```
// Program Greeting
cout « "Simple program description text here." « endl;
```

Source File Header.
Start your source file with a program header. This includes the program name, your name, date and this class. I use the grep trick for .cpp (see below) to look for this. I focus on that homework name and display the next 3 lines. Example:

```
// drake.cpp
// Pat Jones, CISP 413
// 12/34/56
```

Specifications and Specification Bundles.
You document specifications like this: // Specification C1 - Some stuff

You do not need to code them in order. You will probably want to because the specifications get harder as you move up in bundles (not THAT much harder). You also don't need to worry about the specification comments appearing in order in your code, either.

However, all of a specification bundle must be coded to reach that bundle grade (ie all C bundle to get a C). Partially completed bundles DO NOT COUNT. Say you code all specifications for a B bundle and only 1 for an A bundle (out of 5 for example). The highest grade you would get would be a B because that's the last bundle you've completed.

You can stop at any bundle you want, you just can't get a higher grade (ex, you code all specifications for bundle B - the best you can get for this homework is a B). This is designed to mirror the work word, the more features your code has, usually, the happier your clients are. This also gives you some control over your grade.

This style guide has more information on the specifics of these comments.

Variables.
Constant variables - anytime you have a value which is not supposed to change, that's a constant. We make it read only with the const keyword and signify it with the ALL CAPS style: const PI = 3.14; We prefer using constants because they make the code easier to read. There are a few situations where we do not usually use them, such as starting a loop at zero. However, if we have that loop end at, say, 33, then it's a magic number. What's 33? Who knows? If we use const SIZE = 33; we know what 33 is.

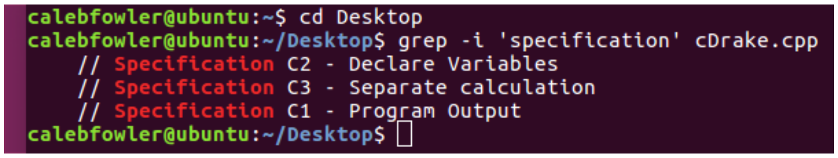
When we have numeric literals appearing in the program we call these magic numbers. We don;t know what they are, but if we change them, the program breaks. hence, magic. Magic numbers are generally frowned upon.

~~*Grep Trick.*~~
Always run your code immediately before your turn it in. I can't tell you how many times students make 'one small change' and turn in broken code. It kills me whenever I see this. Don't kill me.

You can check to see if I will find your specification and feature comments by executing the following command from the command line. If you see your comments on the terminal, then I will see them. If not, I will NOT see them and you will NOT get credit for them. The following will check to see if you have commented your specifications:

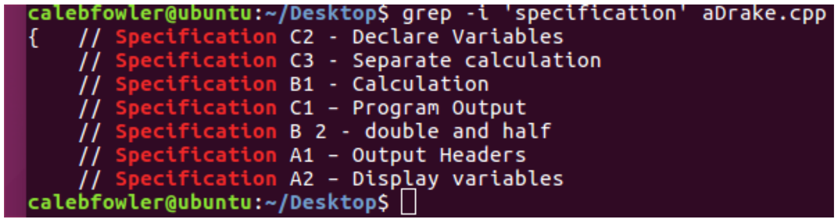
```
grep -i 'specification' homework.cpp
```

This will generate the following output. Notice the specifications are numbered to match the specification number in the assignment. This is what I would expect to see for a 'C' Drake assignment. Note the cd Desktop changes the file location to the desktop - which is where the source file is located.



```
calebfowler@ubuntu:~$ cd Desktop
calebfowler@ubuntu:~/Desktop$ grep -i 'specification' cDrake.cpp
    // Specification C2 - Declare Variables
    // Specification C3 - Separate calculation
    // Specification C1 - Program Output
calebfowler@ubuntu:~/Desktop$ 
```

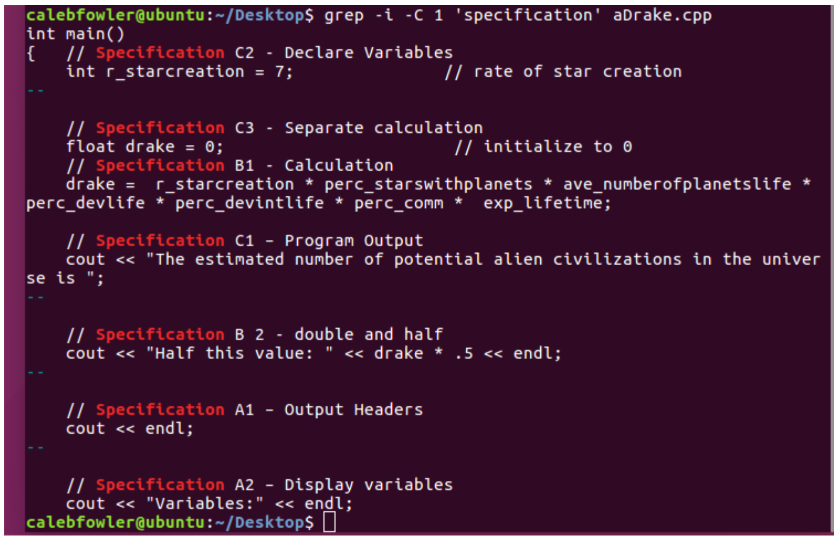
This is what I would expect to see for an 'A' level Drake assignment.



```
calebfowler@ubuntu:~/Desktop$ grep -i 'specification' aDrake.cpp
{   // Specification C2 - Declare Variables
    // Specification C3 - Separate calculation
    // Specification B1 - Calculation
    // Specification C1 - Program Output
    // Specification B 2 - double and half
    // Specification A1 - Output Headers
    // Specification A2 - Display variables
calebfowler@ubuntu:~/Desktop$ 
```

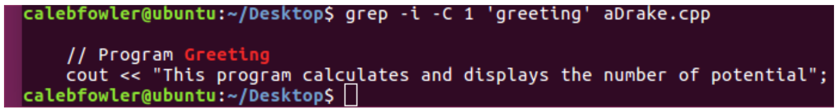
We can also look at the line(s) after the grep statement. I do this to pay attention to code segments.

```
grep -i -C 1 'specification' aDrake.cpp
```



```
calebfowler@ubuntu:~/Desktop$ grep -i -C 1 'specification' aDrake.cpp
int main()
{   // Specification C2 - Declare Variables
    int r_starcreation = 7;              // rate of star creation
--

    // Specification C3 - Separate calculation
    float drake = 0;                      // initialize to 0
    // Specification B1 - Calculation
    drake =  r_starcreation * perc_starswithplanets * ave_numberofplanetslife *
perc_devlife * perc_devintlife * perc_comm *  exp_lifetime;

    // Specification C1 - Program Output
    cout << "The estimated number of potential alien civilizations in the univer
se is ";
--

    // Specification B 2 - double and half
    cout << "Half this value: " << drake * .5 << endl;
--

    // Specification A1 - Output Headers
    cout << endl;
--

    // Specification A2 - Display variables
    cout << "Variables:" << endl;
calebfowler@ubuntu:~/Desktop$ 
```

We can also use this to look for other sections of your code. The grep command searches for anything withing the single quotes ", and the -i option makes it case insensitive. This is how I will look for your program greeting:



```
calebfowler@ubuntu:~/Desktop$ grep -i -C 1 'greeting' aDrake.cpp

    // Program Greeting
    cout << "This program calculates and displays the number of potential";
calebfowler@ubuntu:~/Desktop$ 
```

The grep trick is extremely powerful. Use it often, especially right before you turn in your code. This is the best way I can think of for you to be sure you met all the requirements of the assignment.

Client System.

Your code must compile and run on the client's system. That will be Ubuntu Desktop Linux, version 18.04. Remember, sourcefile.cpp is YOUR program's name. I will type the following command to compile your code:

```
g++ -std=c++14 -g -Wall sourcefile.cpp
```

If you do not follow this standard it is likely I will detect errors you miss - and grade accordingly. If you choose to develop on another system there is a high likelihood your program will fail to compile. You have been warned.

Using the Work of Others.

This is an individual assignment, you may use the Internet and your text to research it, but I expect you to work alone. You **may** discuss code and the assignment. Copying code from someone else and turning it in as your own is plagiarism. I also consider isomorphic homework to be plagiarism. You are ultimately responsible for your homework, regardless of who may have helped you on it.

ProTip: Get a bare bones copy of your code running and turn it in. Then go ahead and modify it with bonuses and whatnot. Upload it with the same name so it replaces your previous homework. This way, if something comes up or you can't finish your homework for some reason, you still have something turned in. A "C" is better than a zero. Risk management class, risk management.

Canvas has a built in plagiarism detector. You should strive to generate a green color box. If you submit it and the score is too high, delete it, change your code and resubmit. You are still subject to the due date, however. This does not apply if I have already graded your homework.

Often, you will not be able to change the code to lower the score. In this case, include as a comment with your homework, what you did and why you thought it was ineffective in lowering your score. This shows me something very important - you are paying attention to what you are doing and you are mindful of your plagiarism score.