John Deere Challenges

Daniel Sánchez Domínguez

28-Oct-2021

Contents

Compilation Instructions

In order to automate the environment setup, installation of required tools and the compilation process, a `bash` and a `Makefile` scripts are provided.

The tools that will be installed are:

- The **clang** compiler
- The **make** tool
- The newest **python** version
- Optional: The **Haskell** interpreter, to run the alternative solutions
- Optional: **Gnuplot**, to plot the benchmarks graphs
- Optional: **Pandoc**, to generate an `HTML` of this markup file

In a Linux terminal, navigate to the directory where these files are located and type the following commands:

```
chmod u+x setup.sh  # Change to executable mode
sudo ./setup.sh     # To install packages permission is needed
make run            # Compile and run all the programs
```

Information of the Software used

The `c` files are intended to follow the `C99` standard for the `C` language. It were compiled through the `clang` compiler version `12.0.0` with consistent results.

The `python` version used was `Python 3.9.5`

The machine used for the compilation and benchmarks has `Ubuntu` as Operating System. The programs and scripts were also tested on a `Fedora` distribution with no detected anomalies.

Specifications Summary

```
Ubuntu clang version 12.0.0-3ubuntu1~21.04.2
Target: x86_64-pc-linux-gnu
Thread model: posix

Python 3.9.5

OS: Pop!_OS 21.04 x86_64
Host: Latitude E7470
Kernel: 5.13.0-7614-generic
Shell: fish 3.1.2
CPU: Intel i5-6300U (4) @ 3.000GHz
```

Usage Instructions

Test the solution programs

The compiled programs could be found in the `target` directory. The `make run` command run all the files with default options. No user inputs are required because the lists are generated with random values. Although the requirements of the problems suggested that the lists would be ordered, it was decided to solve the generalization of the problem with unordered lists. But, if it's desired to see results with ordered lists, it's possible to pass `stdin` arguments in both programs. Also, it's possible to set the lists size.

```
# Type this in the terminal to test with custom input
./target/tractors_number.out 12 25  # The first set will have 12
                                    # elements, and the sencond 25.
                                    # Also, they will be unordered.

./target/spayer_path.out 1000 s     # A sorted list with 1000 values

./target/spayer_path.out            # If no input is given, the
                                    # the length will be a random
                                    # number between 10 and 100
```

To test the `python` script, there are two ways of use. First, as a command line program.

```
# For example
./source/search_c.py --path "./" --name "lists_vectors"
```

If no flags are provided, the program displays a prompt asking for the arguments.

```
# For example
./source/search_c.py
Set the search path: /usr/include/
Set the name to search: errno
```

Test the alternate solutions

The alternate solutions are located in the subfolder `alternative`. The file `./alternative/removeIfElem.hs` contains the full solution to the first two challenges. The `Haskell interpreter` is needed.

```
# To load the file in the REPL just type:
$ ghci alternative/removeIfElem.hs
# Example to test the solution of the first challenge
ghci> intersection [1,2,3,5,6] [6,5,4,5]
[5,6]
ghci> intersection [6,7,4,5,3,7,1] [4,5,3,4]
[4,5,3]
# Example to test the solution of the second challenge
ghci> removeClon [4,5,3,4,2,6,7]
[5,3,2,6,7]
# To exist from the REPL
ghci> :q
```

To test the `bash` solution, two flags should be defined, the path and name.

```
./alternative/search_c.sh -p "./" -n "lists_vectors"

./alternative/search_c.sh -h    # display the help message
```

Explanation of each solution

Tractors VIN Number Look up

C solution The aproach to solve this, was using two nested loops to traverse the two sets of data, comparing each value. In pseudo-python:

```
for i in setA:
    for j in setB:
        if setA[i] == setB[j]:
            yield pop(setB)
```

This functionality was programmed in the function `intersection`, located in the library `source/lists_vectors.c`. The principal problems faced were related to memory allocation. To perform less dynamic allocation, the first set of data was stored in a static vector. The second set, was modeled by a list to could remove each element matched and no compare the same element twice.

Alternate solution Before writing the `C` solution, I solved it with `Haskell`. It is a language that fully embraces the functional paradigm and allows to write algorithms in a very neat way.

```haskell
intersection :: (Eq a) => [a] -> [a] -> [a]
intersection _ [] = []      -- The first lines granted that there
intersection [] _ = []      -- are no empty lists
intersection (x:xs) ys
    | state     = x : intersection xs ys'   -- decisive line
    | otherwise = intersection xs ys'
    where (state, ys') = ifElemRemove (False, x) ys
```

The `decisive line` means that if the list has not changed after trying to remove subsequent occurrences of the x value, then the x value is included in the solution list. The implementation of `ifElemRemove` is as follows:

```haskell
ifElemRemove :: (Eq a) => (Bool, a) -> [a] -> (Bool, [a])
ifElemRemove (c, e) = auxLoop (False, e) where
    auxLoop (state, _) [] = (state, [])
    auxLoop (state, e) (x:xs)
        | e /= x    = (state', x:xs') -- decisive line
        | otherwise = if c then auxLoop (True, e) xs
                           else (True, xs)
        where (state', xs') = auxLoop (state, e) xs
```

The purpose of the `state` variable is to keep track of the possible change in the list. That was needed because in `haskell` all the data is immutable and all the functions are point-free.

Sprayer Path Planning

C solution This was the first solved successfully. And most of the functions in `lists_vectors` were designed with this challenge in mind. If the list functions solve this problem, I was very sure that the first challenge would be solved more easily by combining these functions.

```c
list deleteClones(list ls) {
    node *head = ls.head;
    int value;
    list new_ls = initList();
    while (head != NULL) {
```

```
        value = head->value;
        if (numberOfOccurrences(ls, value) == 1) {
            new_ls = prependData(new_ls, value);
        }
        head = getNextNode(head);
    }
    return new_ls;
}
```

This functions could be improved. One difficulty was that if an element had to be removed, it meant that at that moment was pointed to by the header, so when deleting it, the rest of the list was usually lost. That's why the `numberOfOccurrences` function was used.

Alternate solution The `haskell` solution is very similar to the previous one. The `removeClone` function is exactly the same described before. If the `state` is `true` meaning that at least another occurrence of the value were found after his first position, then that value is ignored.

```
removeClon :: (Eq a) => [a] -> [a]
removeClon []   = []
removeClon (x:xs)
    | state     = removeClon xs'
    | otherwise = x : removeClon xs'
    where (state, xs') = ifElemRemove (True, x) xs
```

Feature Search in File

Python solution This was the easiest challenge of the three. I faced a little bit problem trying to list recursively the contents of a directory.

```
def get_files(path):
    subfolders, files = [], []

    for f in os.scandir(path):          # In the first for, the
        if f.is_dir():                  # inner folders are catched
            subfolders.append(f.path)
        if f.is_file():
            files.append(f.path)

    for sub_dir in list(subfolders):    # The base case is reached
        sf, f = get_files(sub_dir)      # in a folder with no
        subfolders.extend(sf)           # subfolders
        files.extend(f)
    return subfolders, files
```

The rest of the program was more easy, basically give a simple command line interface and filter the files with a `regex`.

Bash solution I wanted to give a `Bash` solution because problems of searching files are almost trivially solved through commands. Here, `find` does all the job listing recursively the directories.

```
# The regex in Bash
find $path -type f -name "$name*\.[ch]$"

# The regex in Python
re.search(f'{name}*\.[ch]$', file):
```

Even though they are basically the same `regex`, the Python implementation gives more false negatives than its Bash counterpart. In the check tests, there were no reported false positives in neither of both.

Header `lists_vectors`

Given the conceptual similarly between the first two challenges, it was decided to write a common library. Below are brief explanations of some of the most important functions.

- `parseInt`: His purpose is handle errors by incorrect user inputs. First, the `strtol` tries to convert the input string to a `int`. After, the the global variable `errno` is analized to classify the possible errors. Only if there are no errors, the integer is returned.
- `deleteThisNode`: Is used by `intersection` to pop the node pointed by `head` if the values match, through the iterations. It granted to not incour in a segmentation fault.
- `prependData`: Even when a working `append` function was possible to program, the `prepend` operation is preferred to not traverse all the list with each new element.

Shortcomings Summary

- The temporal complexity of challenges 1 and 2 is $O(n^2)$. In a couple of papers, an algorithm was found that reduced the complexity to $O(log(n^2))$ using hash tables, however it could not be implemented at the moment.
- The `target/tractors_number.out` when executed with the `sort` option, generates two equal data sets and the intersection only includes the first 3 elements.
- False negatives in the Python implementation of challenge 3.
- Due to time constraints, the benchmarks test suite could not be successfully completed. An observation from the preliminary tests is that the first two programs showed linear growth. The cause is unknown at the moment. Perhaps more thorough testing can lead to a precise conclusion.

Conclusion

Possibly a more optimal implementation could have been achieved using only vectors. Also, more optimizations could have been applied if the ordered list guarantee was used. On the other hand, with the actual approach, one of the reasons that led to more checks and perhaps less optimal functions, was the attempt to avoid segmentation faults.