CS 305: How to engineer faster solutions
Fall 2019

This activity's goal is to summarize all data-structure and programming trade-offs we learned about in this class under the concept of code optimization. Code optimization uses engineering trade-offs between different data-structures and programming concepts to make the code base faster at executing while minimizing the resource use (memory, disk I/O, etc). The activity is divided into three sections: (1) code profiling to identify the code parts that take-up most execution time, (2) optimization and analysis of alternative solutions using optimized code, and (3) code optimization using the compiler. Work in groups of 3-4

Your names: _____

_____

Problem statement:

It is not always obvious which part of the coded solution is the slowest or which data-structure alterations would result in the best speed improvements.

Objectives

Upon completion of this activity, you will be able to do the following:

- profile the code execution and identify the code base performance bottlenecks
- analyze the optimized code using alternative data-structures: arrays, linked lists, hashes.
- use compiler optimization to translate the C language codebase to fast machine code that is optimized for fastest execution possible but not necessarily the best memory use

Part 1: Code profiling – same for every group.

1. Log into your Linux VDI (the terminal emulator like Mobaxterm won't work for this activity as it does not have all tools necessary). Go into your course folder and cerate. Make a new `lab12` directory:

```
cd /drives/p/cs305
mkdir lab12
```

2. Download the `routeFinder_Original.zip` from moodle to your lab12 folder. This is one of the solutions for the airport flight route finding that was assigned as your last homework. Create a subfolder `original` and un-compress the archive you downloaded into this subfolder.

3. In the command line (terminal), navigate to the project folder and compile the code using the `make` tool.

4. The folder has an extra file called `searches.txt`

    a. what does this file contain (useful commands: cat, head, tail, wc,...)

b. why so many entries in the `searches.txt` file?

c. how can we use this file since the `routeFinder` now takes only a single argument `<file name>` that contains airports and routes. You might want to quickly examine `main.c` to see what the code does. Write the command line directive to execute the `routeFinder` with `AllUSRoutes` file and `searches`.

5. measure the execution time of the `routeFinder` program that is loaded with `AllUSRoutes` airports and routes and executes all searches in `searches.txt` file. Use the `time` utility to measure the performance as such:
`/usr/bin/time -v <the rest of the routeFinder command goes here>`

What are the values reported on the lines:

```
Command being timed: "./routeFinder AllUSRoutes.txt"
User time (seconds): _____________________________
System time (seconds): ___________________________
Percent of CPU this job got: _____________________
Elapsed (wall clock) time (h:mm:ss or m:ss): _____________________
```

Which one do we care about the most? Why do these times differ?

6. The last step in the profiling task is to do the code base audit manually for its efficiency. Identify any and all lines of code that could be simplified (merging loops), code removed, data-structure simplified, data structure swapped for a higher efficiency data-structure.

Summarize the major code optimizations you would make:

Draw a diagram of an alternative datastructure to represent the data for increased efficiency (faster execution).

Report out part 1.

Part 2: Understand the optimized code.

At this point, download one of the code bases that you are assigned to investigate. This code base contains an alternative solution for the same problem of finding the shortest flight route from any airport to all other airports.

1. What is the code base you were assigned to analyze?

File name you downloaded and is:_____

2. Unpack the downloaded the code base archive, into a new subfolder in the main activity folder. Open the source files in the IDE and compare this implementation with the original solution that you analyzed in the Part 1. Answering the following answers will help you with understanding how is the alternative solution different from the original code base.

a. How are the structs defined in the .h file different in this version from the original?

b. What happened to the `char** jump` in the `graph` struct? Alternatively, how does the solution print the airport labels now (where is the label stored/accessed)?

c. In the `main.c` after `dijkstra` is called, the shortest route is printed. How is this different from the original implementation? The question above should be helpful.

d. The `dijkstra` does all the work, but it mainly relies on getMin and isEmpty. How do these functions (and the way they are called) differ from the original solution?

3. Compile the code using the make utility. Repeat Part 1, Step 5 and record your findings.
Measure the execution time of the `routeFinder` program that is loaded with `AllUSRoutes` airports and routes and executes all searches in `searches.txt` file.
Use the `time` utility to measure the performance as such:
`/usr/bin/time -v <the rest of the routeFinder command goes here>`

What are the values reported on the lines:

```
Command being timed: "./routeFinder AllUSRoutes.txt"
User time (seconds): _____________________________
System time (seconds): ___________________________
Percent of CPU this job got: _____________________
Elapsed (wall clock) time (h:mm:ss or m:ss): _____________________
```

Report out part 2.

Part 3: Implementing functions

GCC compiler and many other mature C compilers can help out with the optimization of how the source code is translated to the C code. Let' see how much it can help us with the code base

1. Check out the gcc compiler documentation at: https://gcc.gnu.org/onlinedocs/gcc-4.6.4/gcc/

a. Which sections of the manual are relevant to the code base optimization?

_____ and _____

In detail read the first, read the first half page from https://gcc.gnu.org/onlinedocs/gcc-4.6.4/gcc/Optimize-Options.html
Which flags can we use in the makefile to force the gcc to optimize the code translation?

_____

b. Let's use the above flag to make our code (more more more) optimized. Navigate to the `original` source code sub-folder, open makefile for editing and add the optimization flag after the "`gcc -g` " directive. Save the makefile and recompile the project suing "`make clean; make`".

Repeat the Part 1 Step 5 and report the results here:
Measure the execution time of the `routeFinder` program that is loaded with `AllUSRoutes` airports and routes and executes all searches in `searches.txt` file. Use the `time` utility to measure the performance as such:
`/usr/bin/time -v <the rest of the routeFinder command goes here>`

What are the values reported on the lines:

```
Command being timed: "./routeFinder AllUSRoutes.txt"
User time (seconds): _____
System time (seconds): _____
Percent of CPU this job got: _____
Elapsed (wall clock) time (h:mm:ss or m:ss): _____
```

Did the optimization make a dent in the execution time? If so, by how much?

c. repeat the step above, but optimize the codebase with the alternative solution you worked on in Part 2. (add the optimization flag to the make file of the alternative solution, recompile the project, re-run the project and report the results

What are the values reported on the lines:

```
Command being timed: "./routeFinder AllUSRoutes.txt"
User time (seconds): _____________________
System time (seconds): _____________________
Percent of CPU this job got: _____________________
Elapsed (wall clock) time (h:mm:ss or m:ss): _____________________
```

Did the optimization make a dent in the execution time? If so, by how much?

2. Profile the code execution from the above using `valgrind`. Navigate to the sub-folder `original` that contains the original solution. Use the following `valgrind` flag to record all access times (while the command runs work on step 7):
`valgrind –tool=callgrind <the rest of the executable command here>`

a. list the content of your working directory, there is a new file what is it's name and what does it have (useful commands: head -200, cat, ls)

b. open `kcachegrind` with the new file you have in your directory after running `valgrind`. The file name's is likely prefixed by "`callgrind.out…`". From the command line use the following: "`kcachegrind <file callgrind.out… here>`" What are the names of the names of functions that took 90% or more of the program's execution (listed in the left hand column). What functionality do they do when the program executed?

c. Select (click on) the `dijkstra` function in the left hand column and on the right hand column's tab select "collee map". What other functions were called from the `dijkstra` function?

d. Number of times dijkstra function and getMin function were called:
dijkstra function: _____

getMin function: _____

e. Navigate to the sub-folder with the alternative solution. Repeat the profiled execution and report:
--What are the names of the names of functions that took 70% or more time.

--What other functions were called from the `dijkstra` function?

--Number of times the dijkstra function and its “helper” functions were called

Report out Part 3.

Final discussion:
Under which circumstances would the above tested alternative solutions improve the execution time?
Where else should we look to make speed improvements? (Part 3, Section 2 should give you some ideas).
Why hash did not perform as well as expected?