ECS 140A Programming Languages

Spring 2018

Homework #5

Due 11:59pm Tuesday June 5th, 2018

This assignment asks you to complete programming tasks using the Go programming language. This assignment should be worked on individually. Please turn in your solutions electronically via Kodethon or Canvas by the due date.

Getting Started on Kodethon

- Download the project files from Kodethon. Please see this support page[1] for details on downloading the required project files on Kodethon, as well as how to submit your solutions via Kodethon.
- Go to "Switch Environments" from your Kodethon dashboard and choose the "go" execution environment.
- Open the Kodethon Terminal to execute commands. This can be done by selecting the grid icon in the top bar, selecting "CDE Shell", and then clicking the "Terminal" button in the upper-right.

  (**NOTE:** The CDE Shell behaves very differently from the Terminal. Make sure you're using the Terminal!)
- Further questions regarding Kodethon can be directed to the course Piazza forum using the `kodethon` tag.

GOPATH

- You need to set the GOPATH[2] environment variable so that the Go compiler knows how to traverse your project.
- You can do this by using `cd` in your terminal to navigate down to the homework directory, then running `export GOPATH=$(pwd)`.

[1] https://support.kodethon.com/d/38-how-to-use-a-course-as-a-student
[2] https://golang.org/doc/code.html#GOPATH

Test Coverage

- For all parts of this project, you will need to write tests and ensure 100% test coverage of your code. You can generate a coverage profile using the `go test` command. See this post[3] for more on coverage testing.
- To generate a coverage profile for the `Smash` method in the `smash/` package, run `go test smash -run Smash -coverprofile=Smash.cov`.
- You can then run `go tool cover -func=Smash.cov | grep smash.go` to see what the coverage results are.
- You can graphically see which lines of your code are covered by testing using the `go tool cover -html=Smash.cov` command, which opens a new browser window with the results. (On Kodethon, you may need to download the HTML file for local viewing. Add the flag `-o Smash.html` to generate an HTML file, which you can then download from Kodethon.)

Testable Examples in Go

Godoc examples[4] are snippets of Go code that are displayed as package documentation and that are verified by running them as tests. Examples are compiled (and optionally executed) as part of a package's test suite. See also `https://blog.golang.org/examples`.

Parts 2 and 3 in the assignment use such testable examples.

Detecting Race Conditions

- Go includes a race detector,[5] a tool for finding race conditions in Go code.
- The race detector is fully integrated with the Go tool chain. For instance, to enable the race detector for tests simply add the `-race` flag to the command line.
- You might find the race detector useful when debugging the code in Parts 1-2, and when writing your own code for Part 4.

Benchmarking

- The `go test` command supports benchmarking[6] with which functions can be reliably timed.
- `smash_test.go` shows an example of a benchmark, `SmashBenchmark`.
- Add the `-bench` flag to the `go test` command to run the benchmarks.

[3] `https://blog.golang.org/cover`
[4] `https://golang.org/pkg/testing/#hdr-Examples`
[5] `https://blog.golang.org/race-detector`
[6] `https://golang.org/pkg/testing/#hdr-Benchmarks`

- The `-cpu` flag can be used to specify a list of `GOMAXPROCS`[7] values for which the tests or benchmarks should be executed.
- See `https://golang.org/cmd/go/#hdr-Description_of_testing_flags` for a complete description of testing flags.
- You might find it useful to run the `go test -cpu 1,2,4,8 -bench` command to see whether your solution to Parts 4 and 5 exploits parallelism.

Partial Credit

Unlike HW# 2, we do not anticipate giving partial credit for solutions that do not compile or for those do not pass any tests. Partial credit will be given only based on the tests that pass and the code coverage obtained.

The rest of the document describes the four parts of the assignment, and an **extra credit assignment Part 5**.

1 Bug1 (15 points)

The code provided in the package `bug1` contains a bug; it fails the test cases provided in `bug1_test.go`.

- Modify the code in `bug1.go` to fix the bug.
- Write new tests, if needed, to ensure that you get 100% code coverage for your code.

2 Bug2 (20 points)

The code provided in the package `bug2` contains a bug; it fails the test case provided in `bug2_test.go`.

- Add code to `bug2.go` to fix the bug. Removing the use of concurrency is not a valid way to fix the bug.
- Write new tests, if needed, to ensure that you get 100% code coverage for your code.

3 Bug3 (25 points)

The code provided in the package `bug3` contains a bug; it fails the test case provided in `bug3_test.go`.

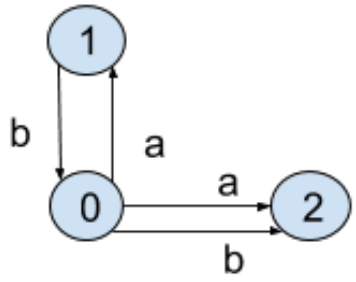
- Add code to `bug3.go` to fix the bug.
- Write new tests, if needed, to ensure that you get 100% code coverage for your code.

[7] `https://golang.org/pkg/runtime/#GOMAXPROCS`

4 NFA (40 points)

A nondeterministic finite automaton (NFA) is defined by a set of states, symbols in an alphabet, and a transition function. A state is represented by an integer. A symbol is represented by a rune, i.e., a character. Given a state and a symbol, a transition function returns the set of states that the NFA can transition to after reading the given symbol. This set of next states could be empty.

A graphical representation of an NFA is shown below:



In this example, $\{0, 1, 2\}$ are the set of states, $\{a, b\}$ are the set of symbols, and the transition function is represented by labelled arrows between states.

- If the NFA is in state 0 and it reads the symbol a, then it can transition to either state 1 or to state 2.
- If the NFA is in state 0 and it reads the symbol b, then it can only transition to state 2.
- If the NFA is in state 1 and it reads the symbol b, then it can only transition to state 0.
- If the NFA is in state 1 and it reads the symbol a, it cannot make any transitions.
- If the NFA is in state 2 and it reads the symbol a or b, it cannot make any transitions.

A given final state is said to be reachable from a given start state via a given input sequence of symbols if there exists a sequence of transitions such that if the NFA starts at the start state it would reach the final state after reading the entire sequence of input symbols.

In the example NFA above,

- The state 1 is reachable from the state 0 via the input sequence *abababa*.
- The state 1 is *not* reachable from the state 0 via the input sequence *ababab*.
- The state 2 is reachable from state 0 via the input sequence *abababa*.

For this part of the assignment you are expected to do the following:

- Write a concurrent implementation of the `Reachable` function in `nfa.go` that returns true if a final state is reachable from the start state after reading an input sequence of symbols.
- Write new tests, if needed, in `nfa_test.go` to ensure that you get 100% code coverage for your code.

Benchmark your code to check whether your implementation benefits from parallelism.

5 Smash (20 points)

This is an optional extra-credit part of the assignment. Points earned in this part of the assignment will be added to Homeworks 2-5.

In this assignment, you have to write a concurrent implementation of the `Smash` function whose inputs are

- `io.Reader`[8] to read text data, and
- a `smasher` function that returns a `uint32` given a `word`. `smasher` may return the same output `uint32` value for different input words.

Words in a string are separated by whitespace and newline. The output of `Smash` is a `map[uint32]uint` that stores the count of the number of words that are mapped to the same value by `smasher`.

As an example, suppose `smasher` maps a `word` to its length. Then for the input `a c d ab abc bac abcd dcba`, `smash` will return the map `{1: 3, 2: 1, 3: 2, 4: 2}`.

On the other hand, if the given `smasher` were to map each word to unique output, then `Smash` would return the count of each word in the input `io.Reader`.

You can look into using bufio.Scanner[9] to read data from the `io.Reader`.[10] You might want to use strings.Fields[11] to split a string into words.

- Write a concurrent implementation of `smash` in `smash.go`. There are tests provided in `smash_test.go`.
- If needed, write your own tests in `smash_test.go` to ensure the tests provide 100% code coverage of the code you write.

Benchmark your code to check whether your implementation benefits from parallelism.

[8] https://golang.org/pkg/io/#Reader
[9] https://golang.org/pkg/bufio/#Scanner
[10] https://golang.org/src/bufio/example_test.go
[11] https://golang.org/pkg/strings/#Fields