ECS 140A Programming Languages

WINTER 2019

Homework #5

About This Assignment

- This assignment asks you to complete programming tasks using the Go programming language.
- To complete the assignment (i) download `hw5-handout.zip` from Canvas, (ii) modify the `.go` files in the `hw5-handout` directory as per the instructions in this document, and (iii) zip the `hw5-handout` directory into `hw5-handout.zip` and upload this `zip` file to Canvas by the due date.

  Do not change the file names, create new files, or change the directory structure in `hw5-handout`.
- This assignment has to be worked on individually.
- We will be using Go version 1.11.4, which can be downloaded from https://golang.org/dl/

  Run the command `go version` to verify that you have the correct version installed:

  ```
  $ go version
  go version go1.11.4 <other output>
  ```
- Go 1.11.4 is installed on all *CSIF machines* in the directory `/usr/local/go/`; the `go` binary is, thus, at `/usr/local/go/bin/go`.

  Consider adding `/usr/local/go/bin` to your `PATH`.
- Information about using CSIF computers, such as how to remotely login to CSIF computers from home and how to copy files to/from the CSIF computers using your personal computer, can be found at http://csifdocs.cs.ucdavis.edu/about-us/csif-general-faq.
- Begin working on the homework early.
- Apart from the description in this document, look at the unit tests provided to understand the requirements for the code you have to write.
- Post questions on piazza if you require any further clarifications. Use private posts if your question contains part of the solution to the homework.

1 Bug1 (5 points)

- Modify code in `hw5-handout/bug1/bug1.go` to fix the concurrency bug.
- Some unit tests are provided for you in `hw5-handout/bug1/bug1_test.go`.

  From the `hw5-handout/bug1` directory, run the `go test` command to run the unit tests.

  Run the `go test -race` command to run the unit tests with the race detector,[1] a tool for finding race conditions in Go code.
- From the `hw5-handout/bug1` directory, run `go test -cover` command to see the current code coverage.
- If needed, add more unit tests in `hw5-handout/bug1/bug1_test.go` to get 100% code coverage for the code in `hw5-handout/bug1/bug1.go`.
- From the `hw5-handout/bug1` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  $ go tool cover -html=temp.cov
  ```

2 Bug2 (5 points)

- Modify code in `hw5-handout/bug2/bug2.go` to fix the concurrency bug.
- Some unit tests are provided for you in `hw5-handout/bug2/bug2_test.go`.

  From the `hw5-handout/bug2` directory, run the `go test` command to run the unit tests.

  Run the `go test -race` command to run the unit tests with the race detector,[2] a tool for finding race conditions in Go code.
- **Removing the use of concurrency is not a valid way to fix the bug.**
- From the `hw5-handout/bug2` directory, run `go test -cover` command to see the current code coverage.
- If needed, add more unit tests in `hw5-handout/bug2/bug2_test.go` to get 100% code coverage for the code in `hw5-handout/bug2/bug2.go`.
- From the `hw5-handout/bug2` directory, run the following two commands to see which statements are covered by the unit tests:

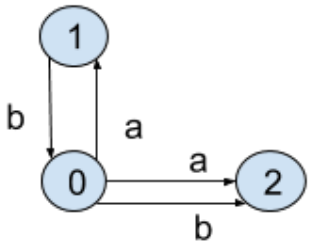
  ```
  $ go test -coverprofile=temp.cov
  $ go tool cover -html=temp.cov
  ```

[1] https://blog.golang.org/race-detector
[2] https://blog.golang.org/race-detector

3 NFA (10 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*.

The transition function for the NFA described above is represented by the expTransitions function in `hw5-handout/nfa/nfa_test.go`. Some unit tests have also been given to you in `hw5-handout/nfa/nfa_test.go`. From the `hw5-handout/nfa` directory, run the `go test` command to run the unit tests.

- Write a **concurrent** implementation of the `Reachable` function in `hw5-handout/nfa/nfa.go` that returns `true` if a final state is reachable from the start state after reading an input sequence of symbols, and `false`, otherwise.
- If needed, write new tests, in `hw5-handout/nfa/nfa_test.go` to ensure that you get 100% code coverage for your code.

  From the `hw5-handout/nfa` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw5-handout/nfa` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  $ go tool cover -html=temp.cov
  ```
- Implementing a sequential version of `Reachable` will get you no points.
- **OPTIONAL**: From the `hw5-handout/nfa` directory, run the following command:

  ```
  $ go test -cpu 1,2,4,8 -bench=.
  ```

  to benchmark[3] your concurrent implementation of `Reachable` and see how your implementation scales with increasing number of CPUs.

4 Smash (10 points)

- Write a **concurrent** implementation of `Smash` in `hw5-handout/smash/smash.go`.
- The `Smash` function takes the following inputs:
  - `io.Reader`[4] 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.
- 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`. Words in a string are separated by whitespace and newline.
- 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`.

[3] https://golang.org/pkg/testing/#hdr-Benchmarks
[4] https://golang.org/pkg/io/#Reader

- Some unit tests are provided for you in `hw5-handout/smash/smash_test.go`.

  From the `hw5-handout/smash/` directory, run the `go test` command to run the unit tests.

- If needed, write more unit tests in `hw5-handout/smash/smash_test.go` to get 100% code coverage for the code in `hw5-handout/smash/smash.go`.

- From the `hw5-handout/smash` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  $ go tool cover -html=temp.cov
  ```

- You can look into using bufio.Scanner[5] to read data from the `io.Reader`.[6]

- You might want to use strings.Fields[7] to split a string into words.

- Implementing a sequential version of `Smash` will get you no points.

- **OPTIONAL**: From the `hw5-handout/smash` directory, run the following command:

  ```
  $ go test -cpu 1,2,4,8 -bench=.
  ```

  to benchmark[8] your concurrent implementation of `Smash` and see how your implementation scales with increasing number of CPUs.

[5] https://golang.org/pkg/bufio/#Scanner
[6] https://golang.org/src/bufio/example_test.go
[7] https://golang.org/pkg/strings/#Fields
[8] https://golang.org/pkg/testing/#hdr-Benchmarks