ECS 140A Programming Languages

Fall 2022

Homework 1

About This Assignment

- This assignment asks you to complete programming tasks using the Go programming language.
- **You are only allowed to use the subset of Go that we have discussed in class. No credit will be given in this assignment if any of the problem solutions use material not discussed in class.** Please use Piazza for any clarifications regarding this issue.
- To complete the assignment (i) download `hw1-handout.zip` from Canvas, (ii) modify the `members.txt` and `.go` files in the `hw1-handout` directory as per the instructions in this document, and (iii) zip the `hw1-handout` directory into `hw1-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 of `hw1-handout`.
- This assignment can be worked on in a group. Even if working in a group, each student has to submit a copy of the work to Canvas to get credit. See the Syllabus for more details.
- List all the names and email addresses of all members of the group in the `members.txt` file in the `hw1-handout` directory, one per line in the format `name <email>`.

  If you are working individually, then only add your name and email to `members.txt`.
- Refer to **Homework 0** for instructions on installing the correct versions of the programming language as well as using CSIF computers.
- 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.
- Keep your homework solution after you submit it. You may need to use it for later assignments.

1 matrix (15 points)

For this part of the assignment, you only need to modify the files `hw1-handout/matrix/matrix.go` and `hw1-handout/matrix/matrix_test.go`.

- A $1 \times m$ matrix of integers can be represented as a slice `[]int` in Go.

  For example, the 1×3 matrix $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ is represented as the slice `[]int{1, 2, 3}`.

- Given a $1 \times m$ matrix mat, the numbers a and b are *adjacent* in m if and only if a occurs immediately to the left or right of b in mat.

  For example, in the 1×3 matrix above, 1 and 2 are adjacent, 2 and 3 are adjacent, and 1 and 3 are NOT adjacent.

- In `hw1-handout/matrix/matrix.go`, implement the `AreAdjacent` function.

  `AreAdjacent(lst, a, b)` returns `true` if and only if the two numbers `a` and `b` are adjacent in the $1 \times m$ matrix represented by `lst`.

- An $n \times m$ matrix of integers can be represented as a slice of slices of integers stored in row-major order. For example, the 2×3 matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ is represented as the slice `[][]int{{1, 2, 3}, {4, 5, 6}}`.

- In `hw1-handout/matrix/matrix.go`, implement the `Transpose` function.

  `Transpose(mat)` returns the transpose of the $n \times m$ matrix represented by `mat`.

  For example, the transpose of the 2×3 matrix above is the 3×2 matrix $\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$, which is represented as the slice `[][]int{{1, 4}, {2, 5}, {3, 6}}`.

- Given a matrix mat, we say that the numbers a and b are *neighbors* in mat if b occurs to the immediate left, right, top, or bottom of a in mat.

  For example, in the 2×3 matrix above, 1 and 2 are neighbors, 2 and 5 are neighbors, 5 and 6 are neighbors, and 2 and 6 are NOT neighbors.

- In `hw1-handout/matrix/matrix.go`, implement the `AreNeighbors` function.

  `AreNeighbors(m, a, b)` returns `true` if and only if the two numbers `a` and `b` are neighbors in the $n \times m$ matrix represented by `m`.

- Unit tests for these functions can be found in `hw1-handout/matrix/matrix_test.go`. From the `hw1-handout/matrix` directory, run the `go test` command to run the unit tests.

- If needed, write new tests, in `hw1-handout/matrix/matrix_test.go` to ensure that you get 100% code coverage for your code.

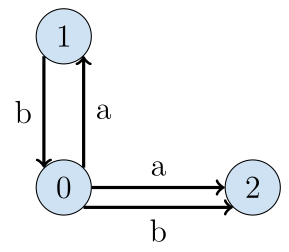
From the `hw1-handout/matrix` directory, run the `go test -cover` command to see the current code coverage.

From the `hw1-handout/matrix` 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 nfa (10 points)

A non-deterministic 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 *ababa*ba.

- 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 `hw1-handout/nfa/nfa_test.go`. Some unit tests have also been given to you in `hw1-handout/nfa/nfa_test.go`. From the `hw1-handout/nfa` directory, run the `go test` command to run the unit tests.

- Write an implementation of the `Reachable` function in `hw1-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 `hw1-handout/nfa/nfa_test.go` to ensure that you get 100% code coverage for your code.

  From the `hw1-handout/nfa` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw1-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
  ```

3 disjointset (20 points)

For this part of the assignment, you only need to modify `hw1-handout/disjointset/disjointset.go` and `hw1-handout/disjointset/disjointset_test.go`.

- A *disjoint-set data structure* maintains a collection $\mathcal{S}$ of disjoint dynamic sets $S_1, S_2, \ldots, S_k$; viz., $S_i \cap S_j = \emptyset, i \neq j$.
- For the purpose of this assignment, assume that initially $\mathcal{S}$ contains a singleton set for each integer; that is, $\mathcal{S} = \{\ldots, \{-2\}, \{-1\}, \{0\}, \{1\}, \{2\}, \{3\}, \ldots\}$.
- Each dynamic set S_i is identified by a *representative*.

  The only requirement for this representative is that it is an element of S_i, and that the representative remains the same if S_i has not been modified.

  The `FindSet`(u) returns the representative of the (unique) set containing u.

  For example, given the initial value of $\mathcal{S}$ stated above, we have `FindSet`$(1) = 1$, `FindSet`$(2) = 2$, and so on.
- The collection $\mathcal{S}$ can be modified using the `UnionSet` operation.

  Let S_u and S_v be the dynamic sets containing u and v, respectively. The operation `UnionSet`(u, v) merges the dynamic sets S_u and S_v. The resulting set $S_u \cup S_v$ is added to $\mathcal{S}$, while S_u and S_v are removed from $\mathcal{S}$.

For example, performing `UnionSet`$(1, 2)$ on the initial value of $\mathcal{S}$ modifies $\mathcal{S}$ to be $\mathcal{S} = \{\dots, \{-2\}, \{-1\}, \{0\}, \{1, 2\}, \{3\}, \dots\}$. Now `FindSet`$(1) =$ `FindSet`(2).

Performing `UnionSet`$(0, -1)$ on $\mathcal{S}$ modifies $\mathcal{S}$ to be $\mathcal{S} = \{\dots, \{-2\}, \{-1, 0\}, \{1, 2\}, \{3\}, \dots\}$. Now `FindSet`$(-1) =$ `FindSet`(0).

Performing `UnionSet`$(2, -1)$ modifies $\mathcal{S}$ to be $\mathcal{S} = \{\dots, \{-2\}, \{-1, 0, 1, 2\}, \{3\}, \dots\}$. Now `FindSet`$(-1) =$ `FindSet`(0) =`FindSet`(1) =`FindSet`(2).

- The `DisjointSet` interface type specifying the `FindSet` and `UnionSet` functions is defined in `hw1-handout/disjointset/disjointset.go`. **Do not change this interface type**.

- Modify `hw1-handout/disjointset/disjoint.go` to define a `struct` type that implements the `DisjointSet` interface. Modify the `NewDisjointSet` function to create an instance of this type.

- Unit tests have been given to you in `hw1-handout/disjointset/disjointset_test.go`. From the `hw1-handout/disjointset` directory, run the `go test` command to run the unit tests.

- If needed, write new tests, in `hw1-handout/disjointset/disjointset_test.go` to ensure that you get 100% code coverage for your code.

  From the `hw1-handout/disjointset` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw1-handout/disjointset` 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
  ```

- The assignment does NOT *require* you to use a specific algorithm when implementing the `DisjointSet` interface; you are free to implement it in any way you want.

  However, you are *encouraged* to implement the data structure as a disjoint-set forest: each disjoint set is represented as a rooted tree with the root of the tree containing the representative. The *path compression* and *union by rank* heuristics ensure that the height of these rooted trees do not grow too large.

  Some of you might have been taught this algorithm in ECS 60 or ECS 122A. You will also find this approach described in a standard algorithms textbook, such as

  Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.

  or in this Wikipedia article.

- We will be using a **timeout of 100 seconds** when running the tests. Our reference solution completes the tests in around 1 second.