Project Problems
SNU 4910.210, Fall 2014

Chung-Kil Hur

due: 12/21(Sun), 24:00

Problem 1 (**10%**) "Dust Storm"

The number of Mars exploration robots are more than one hundred now in A.D. 2032, and the number is still growing fast in order to gather rare metals found at the planet.

A satellite is revolving around Mars so that it exchanges signals with the robots which are carrying out their missions on the surface of Mars.

One of the tasks of this satellite is that, when a dust storm at the surface of Mars is expected, it should evacuate all robots to the shelters. The satellite should assign one particular shelter to each robot by transmitting signals to the robot.

Mars Dust Storm May Affect NASA Rovers Opportunity, Curiosity

Posted: 11/24/2012 10:03 am EST Updated: 11/24/2012 10:03 am EST

One shelter can accommodate only one robot at a time. After a robot receives the signal, it immediately moves to the assigned shelter along the shortest (linear) path with its maximum speed.

A problem is, there is a risk of collisions between robots during the evacuation process. The satellite should assign the shelters to the robots without that risk. Write a program `shelterAssign` which does this job, according to the module type below.

```
module type DUSTSTORM =
 sig
  type robot = string                  (* robot's name *)
  type shelter = int                   (* shelta's id number *)
  type location = int * int            (* coordinate *)
  type robot_locs = (robot * location) list
  type shelter_locs = (shelter * location) list
  val shelterAssigcommentatedn: robot_locs -> shelter_locs -> (robot * shelter) list
 end
```

Assume the surface of Mars is a 2-dimensional plane (no donut-shaped or spheral.) The top-left corner is (0,0) and the bottom-right is (100,100) in the coordinate.

Also, write a comment in the program that explains why this program always calculates a right answer within finite time.

- "Right answer": There should be no possibility that any robots collide with others during evacuation.
- "Within finite time": The program always halts with an answer.
- A hint: First, make pairs of robots and shelters in random, and repeat the following: find pairs which are possible to collide and then do $\cdots$.

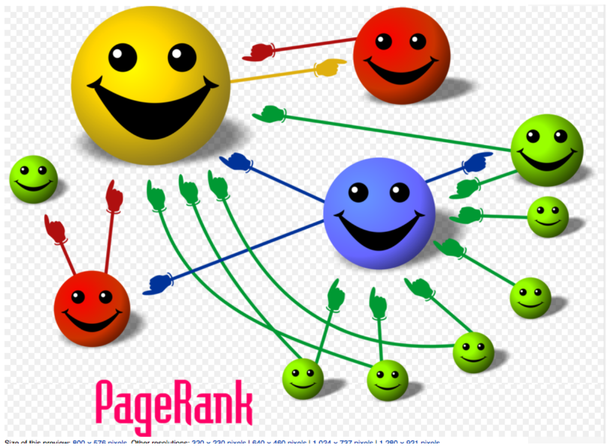
□

Problem 2 (**10%**) "Conjecture of Ranking"

Google's PageRank technology speculates the rank of "importance" of all web pages in the world. The Google search engine first picks web pages containing the phrase that user entered, and then uses this PageRank in order to show the pages to user in the order of the "importance" ranking, hoping that the ranking matches well with the user's want.

So, how can we calculate this "importance"? The idea of the founders of Google, Page and Brin, is like this:

- Assumption: Pages that people visit frequently are "important" pages.
- Then, how can we find out the frequencies of visits?



- Idea: Let's simulate people's behavior while surfing the Internet. People start from one certain page and then move along the links on the page to visit another page. Often they type a URL directly in order to move to a new page, and then start surfing again. If we simulate this behavior to calculate the relative ratio of visits among the web pages, then maybe the ratio represents the "importance" of each page.
- If we assume that, how can we simulate the behavior?
- Answer: We can simulate it by constructing a *Markov chain* model.

Example 1 One example of *Markov chain* modelling

Every year, the movement of populations between Seoul and Sejong follows this: 5% of Seoul's people move to Sejong, and 15% of Sejong's people move to Seoul. The population of Seoul S_n and that of Sejong J_n satisfy the following relation.

$$\begin{aligned} S_n &= 0.95 \times S_{n-1} + 0.15 \times J_{n-1} \\ J_n &= 0.05 \times S_{n-1} + 0.85 \times J_{n-1} \end{aligned}$$

in other words,

$$\begin{pmatrix} S_n \\ J_n \end{pmatrix} = \begin{pmatrix} 0.95 & 0.15 \\ 0.05 & 0.85 \end{pmatrix} \begin{pmatrix} S_{n-1} \\ J_{n-1} \end{pmatrix}$$

Let S_0 and J_0 be the populations of the two cities at the beginning. Then we can obtain the final populations of them by calculating the equations above – or the 'recurrence formulas' – over and over.

Like the matrix above, we call a matrix a 'Markov matrix' if the sum of elements in each column is exactly 1. We call the sequence of values (such as (S_n, J_n) for all n) produced by the recurrence formulas of it a *Markov chain*.

□

Back to the PageRank problem, we can model the ratio of visits for each page like this *Markov chain*:

- Suppose that there are only 3 web pages A, B, and C. Each page has some links that point to other pages. Assume that the ratio that A is visited is proportional to the number of links from A, B, and C to A. For example, if B has 3 links and one of them points to A, then a person who is currently in B will visit A at the next time with the probability of $1/3$. For example: (there are 2 links in A, 3 links in B, and 1 link in A.)

$$\begin{aligned} A_n &= \tfrac{1}{2} \times A_{n-1} + \tfrac{1}{3} \times B_{n-1} + \tfrac{0}{1} \times C_{n-1} \\ B_n &= \tfrac{1}{2} \times A_{n-1} + \tfrac{1}{3} \times B_{n-1} + \tfrac{1}{1} \times C_{n-1} \\ C_n &= \tfrac{0}{2} \times A_{n-1} + \tfrac{1}{3} \times B_{n-1} + \tfrac{0}{1} \times C_{n-1} \end{aligned}$$

  In other words,

$$\begin{pmatrix} A_n \\ B_n \\ C_n \end{pmatrix} = \begin{pmatrix} 1/2 & 1/3 & 0 \\ 1/2 & 1/3 & 1 \\ 0 & 1/3 & 0 \end{pmatrix} \begin{pmatrix} A_{n-1} \\ B_{n-1} \\ C_{n-1} \end{pmatrix}$$

- Generally, for N pages the $N \times N$ Markov matrix $\mathcal{M}$ is like this.
  - $x_{ij} \in \mathcal{M}$: If page i has a link to page j then $1/r_i$ (r_i = the number of links in page i) otherwise 0.
  - Then, the change of ratio of pages along the time is expressed with this recurrence formulas:

$$S_n = \mathcal{M} S_{n-1}$$

  - Goal: We want to calculate the eventual ratio, in other words, we want to calculate the limit of S.

$$S = \lim_{i \to \infty} S_i$$

    We can let the first vector S_0 have $1/N$ for each element. (i.e. We suppose that the probability of each page to be the start page is uniform.)

– Problem: does the limit of the *Markov chain* exist uniquely? And does the program that calculates the limit by repeatedly solving the recurrence formulas always halt within finite time?

– hint: Perron-Frobenius Theorem

Find the answer for the questions above, and write a function `markov_limit` that converts the input matrix into a Markov matrix with the unique limit. It should satisfy the module type below. (The functions `row`, `column`, `add_row`, `add_column`, `size`, `ij` are the functions for constructing/using matrices.)

```
module type MARKOV =
 sig
    type matrix
    val row: float list -> matrix
    val column: float list -> matrix
    val add_row: float list -> matrix -> matrix
    val add_column: float list -> matrix -> matrix
    val size: matrix -> int * int    (* numbers of columns and rows *)
    val ij: matrix -> int -> int -> float
    (*
      Given a Markov matrix and an initial column,
      markov_limit returns the limit of the Markov chain.
    *)
    val markov_limit: matrix -> matrix -> matrix
 end
```

□

Problem 3 (**10%**) "Transformers"

Let's create a "Transformer" that transforms objects in one world into objects in another world. The objects in our transformation are programs which handle integers.

- The objects that we are going to transform are integer expressions which are constructed according to these rules:

$$
\begin{array}{rcll}
E & \to & n & \text{integer} \\
  & | & E\texttt{+}E & \text{addition expression} \\
  & | & \texttt{-}E & \text{sign-change expression} \\
  & | & \texttt{read} & \text{input to be read from outside} \\
  & | & \texttt{if}\ E\ E\ E & \text{conditional expression} \\
  & | & \texttt{repeat}\ E\ E & \text{repeated expression}
\end{array}
$$

  The input expression `read` is an integer which is entered from outside. The range of input is from -100 to 100. The result of the conditional expression

$$\texttt{if}\ E_1\ E_2\ E_3$$

  is determined according to the value of E_1: if E_1's value is not 0 then the result becomes the value of E_2, and otherwise, the result becomes the value of E_3. The value of the repeated expression

$$\texttt{repeat}\ E_1\ E_2$$

  is defined only when the value of E_1 is non-negative, and then the result value is by adding the value of E_2 in E_1 times. In other words, it is (E_1's value) $\times$ (E_2's value). Therefore, when the value of E_1 is 0, the result is 0.

- The resulting objects from the transformation are imperative programs constructed by these rules:

C	$\rightarrow$	x `has` n	variable x has integer n
	$\vert$	x `has` x	variable has value of another variable
	$\vert$	x `has` x`+`x	variable has the result of addition of two variables
	$\vert$	x `has` x`-`x	has result of subtraction of two variables
	$\vert$	x `has read`	has integer from input
	$\vert$	`say` x	print value of variable
	$\vert$	`goto` ℓ `on` x	jump according to value of variable
	$\vert$	ℓ`:` C	tagged command
	$\vert$	C `;` C	commands executed sequencially

  As you see, variables play a crucial role in these programs. All values are always stored in variables, and all calculations are executed with variables. Before using variables, they should contain values.

  The command

  x `has read`

  gets the input from outside and store it at x. As we saw earlier, the value of the input is between -100 and 100. The command

  `goto` ℓ `on` x

  terminates without any operation when the value of x is 0, and otherwise it jumps to the command to which the tag ℓ is attached.

  For example, the following program prints 1:

```
    x has 1 ;
    y has 2 ;
    x has x+y ;
    goto L on x ;
    x has 0 ;
L:  z has x-y ;
    say z
```

  The following program cannot be executed because the variable `z` is used while it has no value.

```
x has 1 ;
y has x+x ;
z has y+z ;
say z
```

- The transformer outputs an imperative program that does the same thing with the input integer expression. The output program should print the same value with the value that the input expression is evaluated to.

- Check before transformation: there exist some "meaningless" expressions that the transformer should reject before transformation. There is only one case for the "meaningless" programs, which happens when the repeated expression

  $$\texttt{repeat}\ E_1\ E_2$$

  has a negative value of E_1. In this case the calculation is not defined.

  The transformer should check whether the input expression contains this meaningless repeated expression. Before transformation, it should predict the value of E_1 in order to check whether it is negative.

- Check after transformation: you should check whether the resulting program C works correctly.

  In fact, you should check whether the input and output programs do the same thing, but here we are going to check these two conditions:

  – All values should contain values before use.

  – The jump statements should have well-defined target commands, which means, for each jump statement there should be only one command which has the corresponding tag.

The transformer `transform` and the checkers, `check_exp` and `check_cmd`, should satisfy the following module type:

```
module type TRANS =
 sig
  type exp = Num of int
           | Add of exp * exp
           | Minus of exp
           | Read
           | If of exp * exp * exp
           | Repeat of exp * exp
  type var = string
  type tag = string
  type cmd = HasNum of var * int
           | HasVar of var * var
           | HasSum of var * var * var
           | HasSub of var * var * var
           | HasRead of var
           | Say of var
           | Goto of tag * var
           | Tag of tag * cmd
           | Seq of cmd * cmd
  val transform: exp -> cmd
  val check_exp: exp -> bool
  val check_cmd: cmd -> bool
 end
```

□