Task 1

(module sn-utils racket

(provide sn-dict-ks-vs

sn-line->entry

sn-list->dict)

;; Returns a dictionary as a list of key-value pairs

(define (sn-dict-ks-vs keys values)

(map cons keys values))

;; Returns a dictionary entry from a string, where the first word is the key and the rest are the value

(define (sn-line->entry ln)

(let ((words (string-split ln)))

(cons (string->symbol (first words)) (rest words))))

;; Returns the input list of key-value pairs

(define (sn-list->dict entries)

entries)

)

Task 2

(module sn-graph scheme

(provide sn-consistent

sn-empty

sn-add-user

sn-users

sn-add-frndshp)

(require scheme/set

scheme/dict)

(define sn-empty empty)

(define (sn-users graph)

(dict-keys graph))

(define (sn-add-user graph user)

(if (dict-ref graph user #f)

graph

(dict-set graph user empty)))

(define (sn-add-frndshp graph u1 u2)

(let ([f1 (dict-ref graph u1 empty)]

[f2 (dict-ref graph u2 empty)])

(dict-set graph

u1

(set-add f1 u2))

(dict-set graph

u2

(set-add f2 u1))))

(define (sn-consistent graph)

(define (consistent-helper [to-check (dict-keys graph)])

(cond

[(empty? to-check) #t]

[else

(let ([current-user (first to-check)]

[remaining-users (rest to-check)])

(define (friend-list-incomplete? user)

(ormap (lambda (friend)

(and (dict-ref graph user #f)

(set-member? (dict-ref graph friend #f) user)))

(dict-ref graph user empty)))

(if (friend-list-incomplete? current-user)

#f

(consistent-helper remaining-users)))]))

(consistent-helper (dict-keys graph))))

Task 3

;; social-network.

;; Easy

;; [(k,v)]| (u,vu) -> vu

(define (sn-ff-for graph user)

(let ([friend-set (dict-ref graph user #f)])

(if friend-set

(set->list friend-set)

'())))

;; Medium

;; [(k,v)]|(u1,f1)|(u2,f2) ->

;; f2 & f3

(define (sn-cmn-frnds-btwn graph user1 user2)

(let ([friend-set1 (dict-ref graph user1 #f)]

[friend-set2 (dict-ref graph user2 #f)])

(if (and friend-set1 friend-set2)

(set->list (set-intersect friend-set1 friend-set2))

'())))

;; Hard

(define (sn-frnd-cnt graph)

(map (lambda (user)

(cons user (set-count (dict-ref graph user empty))))

(dict-keys graph)))

;; pre: length > 0

(define (sn-frndlst-user graph)

(let ([users (dict-keys graph)])

(let\* ([counts (map (lambda (user)

(cons user (set-count (dict-ref graph user empty))))

users)])

(max-by (lambda (x) (cdr x)) counts))))

;; pre: length > 0

(define (sn-unfrndlst-user graph)

(let\* ([counts (map (lambda (user)

(cons user (set-count (dict-ref graph user empty))))

(dict-keys graph))])

(let ([min-count-pair (min-by (lambda (x) (cdr x)) counts)])

min-count-pair)))

;; this is for free. Do not mdify (ROM)

(define (sn-cmn-frnds-ff graph u)

(let\*

([keys (sn-users graph)]

[vals (map

(lambda (key)

(sn-cmn-frnds-btwn graph u key))

keys)]

)

(sn-dict-ks-vs keys vals)))

;; this is for free. Do not mdify (ROM)

(define (sn-cmn-frnds graph )

(let\*

([keys (sn-users graph)]

[vals (map

(lambda (key)

(sn-cmn-frnds-ff graph key))

keys)]

)

(sn-dict-ks-vs keys vals)))

)