Grand Theft Wumpus [1]

Eric Bailey

December 14, 2017 [2]

Contents

[1] Conrad Barski. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!*, chapter 8, pages 129–152. No Starch Press, 2010. ISBN 9781593273491. URL http://landoflisp.com

[2] Last updated December 14, 2017

`src/wumpus.lisp`:

1 ⟨* 1⟩≡
```
(in-package :cl-user)
(defpackage lol.wumpus
  (:use :cl :prove))
(in-package :lol.wumpus)
```

This definition is continued in chunks 2–7.
Root chunk (not used in this document).
Defines:
`lol.wumpus`, used in chunk 11.

Defining the Edges of Congestion City

Load the code from the previous chapter

2a ⟨* 1⟩+≡
```
;; TODO: (load "graph-util")
```

2b ⟨* 1⟩+≡
```
(defparameter *congestion-city-nodes* nil)
(defparameter *congestion-city-edges* nil)
(defparameter *visited-nodes* nil)
(defparameter *node-num* 30 )
(defparameter *edge-num* 45)
(defparameter *worm-num* 3)
(defparameter *cop-odds* 15)
```

Defines:
congestion-city-edges, never used.
congestion-city-nodes, never used.
cop-odds, used in chunk 5a.
edge-num, used in chunk 3a.
node-num, used in chunks 2c, 5a, and 7b.
visited-nodes, never used.
worm-num, used in chunk 7b.

Generating Random Edges

Generate a random number between 1 and *node-num*.

2c ⟨* 1⟩+≡
```
(defun random-node ()
  (1+ (random *node-num*)))
```

Defines:
random-node, used in chunks 3a and 7b.
Uses *node-num* 2b.

Describe edge-pair

2d ⟨* 1⟩+≡
```
(defun edge-pair (a b)
  (unless (eql a b)
    (list (cons a b) (cons b a))))
```

Defines:
edge-pair, used in chunks 3a, 4b, and 6a.

Describe make-edge-list

3a ⟨* 1⟩+≡
```
(defun make-edge-list ()
  (apply #'append (loop repeat *edge-num*
                        collect (edge-pair (random-node) (random-node)))))
```

Defines:
  `make-edge-list`, used in chunk 5a.
Uses `*edge-num*` 2b, `edge-pair` 2d, and `random-node` 2c.

Preventing Islands

Describe direct-edges

3b ⟨* 1⟩+≡
```
(defun direct-edges (node edge-list)
  (remove-if-not (lambda (x)
                   (eql (car x) node))
                 edge-list))
```

Defines:
  `direct-edges`, used in chunks 3c and 5b.

Describe get-connected

3c ⟨* 1⟩+≡
```
(defun get-connected (node edge-list)
  (let ((visited nil))
    (labels ((traverse (node)
               (unless (member node visited)
                 (push node visited)
                 (mapc (lambda (edge)
                         (traverse (cdr edge)))
                       (direct-edges node edge-list)))))
      (traverse node))
    visited))
```

Defines:
  `get-connected`, used in chunk 4a.
Uses `direct-edges` 3b.

Describe find-islands

4a ⟨* 1⟩+≡
```
(defun find-islands (nodes edge-list)
  (let ((islands nil))
    (labels ((find-island (nodes)
               (let* ((connected (get-connected (car nodes) edge-list))
                      (unconnected (set-difference nodes connected)))
                 (push connected islands)
                 (when unconnected
                   (find-island unconnected)))))
      (find-island nodes))
    islands))
```

Defines:
`find-islands`, used in chunk 4c.
Uses `get-connected` 3c.

Describe connect-with-bridges

4b ⟨* 1⟩+≡
```
(defun connect-with-bridges (islands)
  (when (cdr islands)
    (append (edge-pair (caar islands) (caadr islands))
            (connect-with-bridges (cdr islands)))))
```

Defines:
`connect-with-bridges`, used in chunk 4c.
Uses `edge-pair` 2d.

Describe connect-all-islands

4c ⟨* 1⟩+≡
```
(defun connect-all-islands (nodes edge-list)
  (append (connect-with-bridges (find-islands nodes edge-list)) edge-list))
```

Defines:
`connect-all-islands`, used in chunk 5a.
Uses `connect-with-bridges` 4b and `find-islands` 4a.

Building the Final Edges for Congestion City

Describe make-city-edges

5a ⟨* 1⟩+≡

```
(defun make-city-edges ()
  (let* ((nodes (loop for i from 1 to *node-num*
                      collect i))
         (edge-list (connect-all-islands nodes (make-edge-list)))
         (cops (remove-if-not (lambda (x)
                                (zerop (random *cop-odds*)))
                              edge-list)))
    (add-cops (edges-to-alist edge-list) cops)))
```

Defines:
  make-city-edges, never used.
Uses *cop-odds* 2b, *node-num* 2b, add-cops 6a, connect-all-islands 4c,
  edges-to-alist 5b, and make-edge-list 3a.

Describe edges-to-alist

5b ⟨* 1⟩+≡

```
(defun edges-to-alist (edge-list)
  (mapcar (lambda (node1)
            (cons node1
                  (mapcar (lambda (edge)
                            (list (cdr edge)))
                          (remove-duplicates (direct-edges node1 edge-list)
                                             :test #'equal))))
          (remove-duplicates (mapcar #'car edge-list))))
```

Defines:
  edges-to-alist, used in chunk 5a.
Uses direct-edges 3b.

Describe add-cops

6a ⟨* 1⟩+≡
```
(defun add-cops (edge-alist edges-with-cops)
  (mapcar (lambda (x)
            (let ((node1 (car x))
                  (node1-edges (cdr x)))
              (cons node1
                    (mapcar (lambda (edge)
                              (let ((node2 (car edge)))
                                (if (intersection (edge-pair node1 node2)
                                                  edges-with-cops
                                                  :test #'equal)
                                    (list node2 'cops)
                                    edge)))
                            node1-edges))))
          edge-alist))
```

Defines:
 add-cops, used in chunk 5a.
Uses edge-pair 2d.

Building the Nodes for Congestion City

Describe neighbors

6b ⟨* 1⟩+≡
```
(defun neighbors (node edge-alist)
  (mapcar #'car (cdr (assoc node edge-alist))))
```

Defines:
 neighbors, used in chunks 6c and 7a.

Describe within-one

6c ⟨* 1⟩+≡
```
(defun within-one (a b edge-alist)
  (member b (neighbors a edge-alist)))
```

Defines:
 within-one, used in chunk 7.
Uses neighbors 6b.

Describe within-two

7a ⟨* 1⟩+≡

```
(defun within-two (a b edge-alist)
  (or (within-one a b edge-alist)
      (some (lambda (x)
              (within-one x b edge-alist))
            (neighbors a edge-alist))))
```

Defines:
  within-two, used in chunk 7b.
Uses neighbors 6b and within-one 6c.

Describe make-city-nodes

7b ⟨* 1⟩+≡

```
(defun make-city-nodes (edge-alist)
  (let ((wumpus (random-node))
        (glow-worms (loop for i below *worm-num*
                          collect (random-node))))
    (loop for n from 1 to *node-num*
          collect (append (list n)
                          (cond ((eql n wumpus) '(wumpus))
                                ((within-two n wumpus edge-alist) '(blood!)))
                          (cond ((member n glow-worms)
                                 '(glow-worm))
                                ((some (lambda (worm)
                                         (within-one n worm edge-alist))
                                       glow-worms)
                                  '(lights!)))
                          (when (some #'cdr (cdr (assoc n edge-alist)))
                            '(sirens!))))))
```

Defines:
  make-city-nodes, never used.
Uses *node-num* 2b, *worm-num* 2b, random-node 2c, within-one 6c, and within-two 7a.

Full Listing

```
(in-package :cl-user)
(defpackage lol.wumpus
  (:use :cl :prove))
(in-package :lol.wumpus)


;; TODO: (load ”graph-util”)


(defparameter *congestion-city-nodes* nil)
(defparameter *congestion-city-edges* nil)
(defparameter *visited-nodes* nil)
(defparameter *node-num* 30 )
(defparameter *edge-num* 45)
(defparameter *worm-num* 3)
(defparameter *cop-odds* 15)


(defun random-node ()
  (1+ (random *node-num*)))


(defun edge-pair (a b)
  (unless (eql a b)
    (list (cons a b) (cons b a))))


(defun make-edge-list ()
  (apply #'append (loop repeat *edge-num*
                        collect (edge-pair (random-node) (random-node)))))


(defun direct-edges (node edge-list)
  (remove-if-not (lambda (x)
                   (eql (car x) node))
                 edge-list))


(defun get-connected (node edge-list)
  (let ((visited nil))
    (labels ((traverse (node)
               (unless (member node visited)
                 (push node visited)
                 (mapc (lambda (edge)
                         (traverse (cdr edge)))
                       (direct-edges node edge-list)))))
      (traverse node))
    visited))
```

```
(defun find-islands (nodes edge-list)
  (let ((islands nil))
    (labels ((find-island (nodes)
               (let* ((connected (get-connected (car nodes) edge-list))
                      (unconnected (set-difference nodes connected)))
                 (push connected islands)
                 (when unconnected
                   (find-island unconnected)))))
      (find-island nodes))
    islands))


(defun connect-with-bridges (islands)
  (when (cdr islands)
    (append (edge-pair (caar islands) (caadr islands))
            (connect-with-bridges (cdr islands)))))


(defun connect-all-islands (nodes edge-list)
  (append (connect-with-bridges (find-islands nodes edge-list)) edge-list))


(defun make-city-edges ()
  (let* ((nodes (loop for i from 1 to *node-num*
                      collect i))
         (edge-list (connect-all-islands nodes (make-edge-list)))
         (cops (remove-if-not (lambda (x)
                                (zerop (random *cop-odds*)))
                              edge-list)))
    (add-cops (edges-to-alist edge-list) cops)))


(defun edges-to-alist (edge-list)
  (mapcar (lambda (node1)
            (cons node1
                  (mapcar (lambda (edge)
                            (list (cdr edge)))
                          (remove-duplicates (direct-edges node1 edge-list)
                                             :test #'equal))))
          (remove-duplicates (mapcar #'car edge-list))))
```

```
(defun add-cops (edge-alist edges-with-cops)
  (mapcar (lambda (x)
            (let ((node1 (car x))
                  (node1-edges (cdr x)))
              (cons node1
                    (mapcar (lambda (edge)
                              (let ((node2 (car edge)))
                                (if (intersection (edge-pair node1 node2)
                                                  edges-with-cops
                                                  :test #'equal)
                                    (list node2 'cops)
                                    edge)))
                            node1-edges))))
          edge-alist))


(defun neighbors (node edge-alist)
  (mapcar #'car (cdr (assoc node edge-alist))))


(defun within-one (a b edge-alist)
  (member b (neighbors a edge-alist)))


(defun within-two (a b edge-alist)
  (or (within-one a b edge-alist)
      (some (lambda (x)
              (within-one x b edge-alist))
            (neighbors a edge-alist))))


(defun make-city-nodes (edge-alist)
  (let ((wumpus (random-node))
        (glow-worms (loop for i below *worm-num*
                          collect (random-node))))
    (loop for n from 1 to *node-num*
          collect (append (list n)
                          (cond ((eql n wumpus) '(wumpus))
                                ((within-two n wumpus edge-alist) '(blood!)))
                          (cond ((member n glow-worms)
                                 '(glow-worm))
                                ((some (lambda (worm)
                                         (within-one n worm edge-alist))
                                       glow-worms)
                                 '(lights!)))
                          (when (some #'cdr (cdr (assoc n edge-alist)))
                            '(sirens!))))))
```

Tests

Implement tests

11 ⟨*test/wumpus.lisp* 11⟩≡
```
  (in-package :lol.wumpus)
```

Root chunk (not used in this document).
Uses `lol.wumpus` 1.

References

Conrad Barski. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!*, chapter 8, pages 129–152. No Starch Press, 2010. ISBN 9781593273491. URL `http://landoflisp.com`.

To-Do