AIMA for Chicken Scheme

Peter Danenberg <pcd@roxygen.org>

August 13, 2012

Contents

1 AIMA

1.1 aima

Module aima

Description AIMA contains functions common to agents and environments.

Exports

- compose-environments
- debug?
- debug-print

- `default-steps`
- `make-debug-environment`
- `make-step-limited-environment`
- `make-performance-measuring-environment`
- `random-seed`
- `randomize!`
- `simulate`

1.2 debug?

Parameter `#t`

Description Should we print debugging information to stdout?

```
(define debug? (make-parameter #t))
```

1.3 debug-print

Procedure `(debug-print key value)` → `unspecified`
`(debug-print key value out)` → `unspecified`

Description Print key-value pairs if the parameter 'debug?' is true.

Parameters

`key`	The key to print
`value`	The value to print
`out`	The port to print to

```
(define debug-print
  (case-lambda
    ((key value) (debug-print key value #t))
    ((key value out) (if (debug?) (format out "~a: ~a~%" key value)))))
```

1.4 random-seed

Parameter `#f`

Description 'random-seed' is passed to 'randomize!' during 'simulate'.

```
(define random-seed (make-parameter #f))
```

1.5 randomize!

Parameter `randomize`

Description 'randomize!' is called before simulation and is seeded with 'random-seed'.

```
(define randomize! (make-parameter randomize))
```

1.6 simulate

Procedure `(simulate environment)` → `#f`
`(simulate environment randomize! random-seed)` → `#f`

Description Run an environment to completion; an environment is complete when it returns false.

Parameters

`environment`	The environment to simulate
`randomize!`	Function to seed the random-number generator for reproducible results
`random-seed`	Seed to seed the random-number generator

```
(define simulate
  (case-lambda
    ((environment) (simulate environment (randomize!) (random-seed)))
    ((environment randomize! random-seed)
     (if random-seed (randomize! random-seed))
     (loop ((while (environment)))))))
```

1.7 compose-environments

Procedure `(compose-environments . environments)` → `environment`

Description Compose environments into a single environment suitable for 'simulate'.

'compose-environments' effectively 'ands' over its constituent environments every step.

Parameters

`environments`	The environments to be composed

```
(define (compose-environments . environments)
  (lambda ()
    (every identity (map (lambda (environment) (environment)) environments))))
```

1.8 make-performance-measuring-environment

Procedure `(make-performance-measuring-environment measure-performance score-update!)` → `environment`

Description Make an environment that updates a score according to a performance measure.

Parameters

`measure-performance`	A nullary procedure which measures performance
`score-update!`	A function which receives the performance measure and updates the score accordingly

```
(define (make-performance-measuring-environment
         measure-performance
         score-update!)
  (lambda () (score-update! (measure-performance))))
```

1.9 default-steps

Parameter `1000`

Description Default number of steps for the step-limited environment

```
(define default-steps (make-parameter 1000))
```

1.10 make-step-limited-environment

Procedure

`(make-step-limited-environment)`	→	`environment`
`(make-step-limited-environment steps)`	→	`environment`

Description Make an environment that stops simulation after a certain number of steps.

Parameters

`steps`	The number of steps after which to stop simulating

```
(define make-step-limited-environment
  (case-lambda
    (() (make-step-limited-environment (default-steps)))
    ((steps)
     (let ((current-step 0))
       (lambda ()
         (set! current-step (+ current-step 1))
         (< current-step steps))))))
```

1.11 make-debug-environment

Syntax `(make-debug-environment object make-printable-object)` → `environment`

Description Make an environment that prints debugging information (according to 'debug?').

Parameters

`object`	The object to debug
`make-printable-object`	A function which optionally transforms the object before printing

```
(define-syntax
  make-debug-environment
  (er-macro-transformer
    (lambda (expression rename compare)
      (let ((%print (rename 'debug-print)))
        (match expression
               ((_ object) `(lambda () (,%print ',object ,object)))
               ((_ object make-printable-object)
                `(lambda ()
                   (,%print ',object (,make-printable-object ,object)))))))))
```

2 AIMA-Vacuum

2.1 aima-vacuum

Module `aima-vacuum`

Description 'aima-vacuum' has agents and environments for chapter 2: Intelligent Agents.

Exports

- `agent-score`
- `agent-score-set!`
- `agent-location`
- `agent-location-set!`
- `agent-program`
- `agent-program-set!`
- `clean`
- `clean?`
- `compare-graphs`
- `copy-world`
- `cycle`

- cycle?
- connect!
- default-n-nodes
- direction->move
- dirty
- dirty?
- display-world
- display-pdf
- down
- down?
- left
- left?
- location-status
- location-status-set!
- location-neighbors
- location-neighbors-set!
- make-agent
- make-graph
- make-graph-world
- make-linear-world
- make-location
- make-node
- make-performance-measure
- make-preferential-depth-first-world
- make-randomized-graph-agent
- make-reflex-agent
- make-simple-reflex-agent
- make-stateful-reflex-agent
- make-stateful-graph-agent
- make-score-update!
- make-unknown-location
- make-world
- move->direction
- random-start
- reverse-move

- `right`
- `right?`
- `simulate-graph`
- `simulate-graph/animation`
- `simulate-penalizing-vacuum`
- `simulate-vacuum`
- `unknown`
- `unknown?`
- `up`
- `up?`
- `world-location`
- `world-location-set!`
- `write-world-as-pdf`
- `write-world-as-dot`
- `write-world-as-gif`

2.2 Two-square vacuum-world

2.2.1 `display-world`

Procedure `(display-world world)` → `unspecified`

Description Display the two-square vacuum world as a vector.

Parameters `world` The two-square vacuum world to be displayed

```
(define (display-world world)
  (pp (vector-append
        '#(world)
        (vector-map
          (lambda (i location) (if (clean? location) 'clean 'dirty))
          world))))
```

2.2.2 `clean`

Scalar `(make-clean)`

Description A clean square

```
(define clean (make-clean))
```

2.2.3 `dirty`

Scalar `(make-dirty)`

Description A dirty square

```
(define dirty (make-dirty))
```

2.2.4 `unknown`

Scalar `(make-unknown)`

Description An unknown square (either clean or dirty)

```
(define unknown (make-unknown))
```

2.2.5 `left`

Scalar `0`

Description Index of the left square

```
(define left 0)
```

2.2.6 `left?`

Procedure `(left? square)` → `true if it is the left square`

Description Is this the left square?

Parameters `square` The square to be lefted

```
(define left? zero?)
```

2.2.7 `right`

Scalar `1`

Description Index of the right square

```
(define right 1)
```

2.2.8 `right?`

Procedure `(right? square)` → `true if it is the right square`

Description Is this the right square?

Parameters `square` The square to be righted

```
(define right? (cute = <> 1))
```

2.2.9 `make-world`

Procedure `(make-world left right)` → `a two-square vacuum world`

Description Make a two-square vacuum-world.

Parameters

`left`	State of the left square (clean or dirty)
`right`	State of the left square (clean or dirty)

```
(define make-world vector)
```

2.2.10 `world-location`

Procedure `(world-location square)` → `the square-status`

Description Get a square-status (dirty, clean, unknown, &c.) from the two-square vacuum-world.

Parameters

`square`	The square's index ('left' or 'right')

```
(define world-location vector-ref)
```

2.2.11 `world-location-set!`

Procedure `(world-location-set! square status)` → `unspecified`

Description Set the status of a square to dirty, clean, unknown, &c.

Parameters

`square`	The square to be set
`status`	The status to set it to

```
(define world-location-set! vector-set!)
```

2.2.12 `agent`

Record `agent`

Description The fundamental agent-record

Fields

`location`	Where the agent is located
`score`	The agent's score at a given time
`program`	The agent's program: an n-ary procedure where each argument corresponds to a sensor; what is received by the sensors depends on the environments contract with its agents.

```
(define-record agent location score program)
```

2.2.13 simple-agent-program

Procedure `(simple-agent-program location clean?)` → `one of 'left, 'right, 'suck, 'noop`

Description Example of a simple two-square vacuum-agent that merely responds to its percept.

Parameters

`location`	The location of the agent
`clean?`	Whether or not this square is clean

```
(define (simple-agent-program location clean?)
  (if clean? (if (left? location) 'right 'left) 'suck))
```

2.2.14 make-stateful-agent-program

Procedure `(make-stateful-agent-program)` → `stateful agent program`

Description Make an agent program that models the two-square vacuum-world, and stops cleaning.

```
(define (make-stateful-agent-program)
  (let ((world (make-world unknown unknown)))
    (lambda (location clean?)
      (if clean?
        (begin
          (vector-set! world location clean)
          (if (all-clean? world) 'noop (if (right? location) 'left 'right)))
        'suck))))
```

2.2.15 make-reflex-agent

Procedure

`(make-reflex-agent location)`	→	`unspecified`
`(make-reflex-agent location program)`	→	`unspecified`

Description Make a stateless agent that merely responds to its current percept.

Parameters

`location`	Where does the agent start? 'left' or 'right'
`program`	The agent's program; should be a binary procedure that takes a location and whether that location is clean. See 'simple-agent-program'.

```
(define make-reflex-agent
  (case-lambda
    ((location) (make-reflex-agent location (default-agent-program)))
    ((location program) (make-agent location 0 program))))
```

2.2.16 `make-simple-reflex-agent`

Procedure `(make-simple-reflex-agent location)` → `a simple reflex agent`

Description Make a simple reflex agent and place it in the given location.

Parameters `location` Where to place the agent: 'left' or 'right'

```
(define (make-simple-reflex-agent location)
  (make-reflex-agent location simple-agent-program))
```

2.2.17 `make-stateful-reflex-agent`

Procedure `(make-stateful-reflex-agent location)` → `a stateful reflex agent`

Description Make a stateful reflex agent and place it in the given location.

Parameters `location` Where to place the agent: 'left' or 'right'

```
(define (make-stateful-reflex-agent location)
  (make-reflex-agent location (make-stateful-agent-program)))
```

2.2.18 `make-performance-measure`

Procedure `(make-performance-measure world)` → `environment`

Description Make a performance measure that awards one point for every clean square.

```
(define (make-performance-measure world)
  (lambda () (vector-count (lambda (i square) (clean? square)) world)))
```

2.2.19 `make-score-update!`

Procedure `(make-score-update! agent)` → `a monadic procedure that takes the score to add`

Description Make a score-updater that adds score to the score of an agent.

Parameters `agent` The agent whose score to add to

```
(define (make-score-update! agent)
  (lambda (score) (agent-score-set! agent (+ (agent-score agent) score))))
```

2.2.20 `simulate-vacuum`

Procedure `(simulate-vacuum world agent)` → `the agent-score`
`(simulate-vacuum world agent steps)` → `the agent-score`
`(simulate-vacuum world agent steps make-environment)` → `the agent-score`

Description Simulate the two-square vacuum-world.

Parameters

`world`	The two-square vacuum world (see 'make-world')
`agent`	The agent to inhabit the world
`steps`	The number of steps to simulate (default: 1000)
`make-environment`	The environment constructor (default: 'make-environment')

```
(define simulate-vacuum
  (case-lambda
    ((world agent) (simulate-vacuum world agent (default-steps)))
    ((world agent steps) (simulate-vacuum world agent steps make-environment))
    ((world agent steps make-environment)
     (simulate
       (compose-environments
         (make-step-limited-environment steps)
         (make-performance-measuring-environment
           (make-performance-measure world)
           (make-score-update! agent))
         (make-debug-environment
           agent
           (lambda (agent)
             (vector
               (let ((location (agent-location agent)))
                 (if (left? location) 'left 'right))
               (agent-score agent))))
         (make-debug-environment world)
         (make-environment world agent)))
     (agent-score agent))))
```

2.2.21 `simulate-penalizing-vacuum`

Procedure `(simulate-penalizing-vacuum world agent)` → `the agent-score`
`(simulate-penalizing-vacuum world agent steps)` → `the agent-score`

Description Like 'simulate-vacuum', but penalizes agents for every movement.

Parameters

`world`	The two-square vacuum world (see 'make-world')
`agent`	The agent to inhabit the world
`steps`	The number of steps to simulate (default: 1000)

```
(define simulate-penalizing-vacuum
  (case-lambda
    ((world agent) (simulate-penalizing-vacuum world agent (default-steps)))
    ((world agent steps)
     (simulate-vacuum world agent steps make-penalizing-environment))))
```

2.3 Graph-based vacuum-world

2.3.1 `make-graph`

Procedure `(make-graph)` → `graph`

Description Make a hash-table-based adjacency list.

```
(define make-graph make-hash-table)
```

2.3.2 `up`

Scalar `2`

Description Index of the up square

```
(define up 2)
```

2.3.3 `up?`

Procedure `(up?)` → `true if it is the up square`

Description Is this the up square?

```
(define up? (cute = <> 2))
```

2.3.4 `down`

Scalar `3`

Description Index of the down square

```
(define down 3)
```

2.3.5 `down?`

Procedure `(down?)` → `true if this is the down square`

Description Is this the down square?

```
(define down? (cute = <> 3))
```

2.3.6 `location`

Record `location`

Description Location-records describing the status (e.g. clean, dirty) of the square and its neighbors at 'left', 'right', 'down', 'up'.

'neighbors' is a ternary vector indexed by relative directions.

```
(define-record location status neighbors)
```

2.3.7 `copy-world`

Procedure `(copy-world world)` → `graph-world`

Description Make a deep copy of a graph-world.

Parameters `world` The world to copy

```
(define (copy-world world)
  (let ((world (hash-table-copy world)))
    (hash-table-walk
      world
      (lambda (name location) (hash-table-update! world name copy-location)))
    world))
```

2.3.8 `make-node`

Procedure `(make-node)` → `symbol`

Description Make a unique symbol suitable for a node-name.

```
(define make-node gensym)
```

2.3.9 `connect!`

Procedure `(connect! world connectend connector direction)` → `unspecified`

Description Bi-connect two locations over a direction and its inverse.

Parameters

`world`	The graph-world within which to connect
`connectend`	The node to be connected
`connector`	The connecting node
`direction`	The relative direction to connect over

```
(define (connect! world connectend connector direction)
  (hash-table-update!/default
    world
    connectend
    (lambda (location)
      (vector-set! (location-neighbors location) direction connector)
      location)
    (make-dirty-location))
  (hash-table-update!/default
    world
    connector
    (lambda (location)
      (vector-set!
        (location-neighbors location)
        (reverse-direction direction)
        connectend)
      location)
    (make-dirty-location)))
```

2.3.10 `random-start`

Procedure `(random-start world)` → `symbol`

Description Find a random starting node in the given world.

Parameters `world` The world to search

```
(define (random-start world)
  (let ((nodes (hash-table-keys world)))
    (list-ref nodes (bsd-random-integer (length nodes)))))
```

2.3.11 `make-randomized-graph-agent`

Procedure `(make-randomized-graph-agent start)` → `agent`

Description Make a simply reflex agent that randomly searches the graph and cleans dirty squares.

Parameters `start` Starting square (see 'random-start')

```
(define (make-randomized-graph-agent start)
  (make-reflex-agent
    start
    (lambda (location clean?)
      (if clean? (list-ref '(left right up down) (random-direction)) 'suck))))
```

2.3.12 `default-n-nodes`

Parameter `20`

Description Default number of nodes for a graph

```
(define default-n-nodes (make-parameter 20))
```

2.3.13 `make-linear-world`

Procedure `(make-linear-world)` → `graph`
`(make-linear-world n-nodes)` → `graph`

Description Make a world that consists of a line of nodes (for testing pathological cases.

Parameters `n-nodes` Number of nodes in the graph (default: (default-n-nodes))

```
(define make-linear-world
  (case-lambda
    (() (make-linear-world (default-n-nodes)))
    ((n-nodes)
     (let ((world (make-graph))
           (nodes (list-tabulate n-nodes (lambda i (make-node)))))
       (for-each
         (lambda (node1 node2) (connect! world node1 node2 right))
         (drop nodes 1)
         (drop-right nodes 1))
       world))))
```

2.3.14 `make-preferential-depth-first-world`

Procedure `(make-preferential-depth-first-world)` → `graph`
`(make-preferential-depth-first-world n-nodes)` → `graph`

Description Create a random-graph using depth-first search that nevertheless shows preference for connected nodes (á la Barabási-Albert).

The graph has no cycles.

Parameters `n-nodes` The number of nodes in the graph (default: (default-n-nodes))

```
(define make-preferential-depth-first-world
  (case-lambda
    (() (make-preferential-depth-first-world (default-n-nodes)))
    ((n-nodes)
     (let* ((world (make-seed-world)) (start (random-start world)))
       (let iter ((node start)
```

```
                  (n-nodes (max 0 (- n-nodes (count-nodes world))))
                  (n-degrees (count-degrees world)))
          (if (zero? n-nodes)
            world
            (let ((location
                    (hash-table-ref/default world node (make-dirty-location))))
              (let ((n-neighbors (n-neighbors location)))
                (if (and (< n-neighbors 4)
                         (< (bsd-random-real) (/ n-neighbors n-degrees)))
                  (let* ((new-directions
                          (vector-fold
                            (lambda (direction directions neighbor)
                              (if (no-passage? neighbor)
                                (cons direction directions)
                                directions))
                            '()
                            (location-neighbors location)))
                         (new-direction
                           (list-ref
                             new-directions
                             (bsd-random (length new-directions)))))
                    (let ((new-node (make-node)))
                      (connect! world node new-node new-direction)
                      (iter new-node (- n-nodes 1) (+ n-degrees 2))))
                  (let* ((neighbors
                          (vector-fold
                            (lambda (direction neighbors neighbor)
                              (if (passage? neighbor)
                                (cons neighbor neighbors)
                                neighbors))
                            '()
                            (location-neighbors location)))
                         (neighbor
                           (list-ref
                             neighbors
                             (bsd-random (length neighbors)))))
                    (iter neighbor n-nodes n-degrees)))))))))))
```

2.3.15 `make-graph-world`

Procedure `(make-graph-world n-nodes)` → `graph`

Description Make a random graph.

Parameters `n-nodes` The number of nodes in the graph (default: (default-n-nodes))

```
(define make-graph-world make-preferential-depth-first-world)
```

2.3.16 `write-world-as-dot`

Procedure `(write-world-as-dot world agent)` → `unspeci`
`(write-world-as-dot world agent step)` → `unspeci`
`(write-world-as-dot world agent step width height font-size title)` → `unspeci`

Description Output the graph-world as in dot-notation (i.e. Graphviz).

Parameters

`world`	The graph-world to output
`agent`	The agent inhabiting the graph-world
`step`	The current step or false
`width`	Width of the output
`height`	Height of the output
`font-size`	Font-size of the output
`title`	Title of the output

```
(define write-world-as-dot
  (case-lambda
    ((world agent) (write-world-as-dot world agent #f))
    ((world agent step)
     (write-world-as-dot
       world
       agent
       step
       (default-width)
       (default-height)
       (default-font-size)
       (default-title)))
    ((world agent step width height font-size title)
     (write-dot-preamble agent step width height font-size title)
     (write-dot-nodes world agent)
     (write-dot-edges world)
     (write-dot-postscript))))
```

2.3.17 `write-world-as-pdf`

Procedure `(write-world-as-pdf world agent pdf)` → `unspecified`

Description Output the graph-world as a pdf via graphviz.

Parameters

`world`	The world to output
`agent`	The agent that inhabits the world
`pdf`	The file to write to

```
(define (write-world-as-pdf world agent pdf)
  (receive
    (input output id)
    (process "neato" `("-Tpdf" "-o" ,pdf))
```

```
    (with-output-to-port
      output
      (lambda () (write-world-as-dot world agent #f #f #f #f #f)))
    (flush-output output)
    (close-output-port output)
    (close-input-port input)))
```

2.3.18 write-world-as-gif

Procedure `(write-world-as-gif world agent frame gif)` → un
`(write-world-as-gif world agent frame gif width height font-size title)` → un

Description Output the graph-world as gif via Graphviz (useful for e.g. animations).

Parameters

`world`	The graph-world to output
`agent`	The agent inhabiting the graph-world
`frame`	The frame-number
`gif`	The base-name of the gif to write to
`width`	Width of the output
`height`	Height of the output
`font-size`	Font-size of the output
`title`	Title of the output

```
(define write-world-as-gif
  (case-lambda
    ((world agent frame gif)
     (write-world-as-gif
       world
       agent
       frame
       gif
       (default-width)
       (default-height)
       (default-font-size)
       (default-title)))
    ((world agent frame gif width height font-size title)
     (receive
       (input output id)
       (process "neato" `("-Tgif" "-o" ,gif))
       (with-output-to-port
         output
         (lambda ()
           (write-world-as-dot
             world
             agent
```

```
              frame
              width
              height
              font-size
              title)))
        (flush-output output)
        (close-output-port output)
        (close-input-port input)))))
```

2.3.19 `make-unknown-location`

Procedure `(make-unknown-location clean?)` → `location`

Description Make a graph-location whose neighbors are all unknown.

Parameters `clean?` Is the graph-location clean?

```
(define (make-unknown-location clean?)
  (make-location
    (if clean? clean dirty)
    (vector unknown unknown unknown unknown)))
```

2.3.20 `reverse-move`

Procedure `(reverse-move move)` → `direction`

Description Reverse the relative direction.

Parameters `move` The relative direction to reverse

```
(define (reverse-move move)
  (case move ((left) 'right) ((right) 'left) ((up) 'down) ((down) 'up)))
```

2.3.21 `direction->move`

Procedure `(direction->move direction)` → `relative direction`

Description Convert a neighbor-index into a relative direction.

Parameters `direction` The index to convert

```
(define (direction->move direction) (list-ref '(left right up down) direction))
```

2.3.22 `move->direction`

Procedure `(move->direction move)` → `index`

Description Convert a relative direction into a neighbor index.

Parameters `move` The relative direction to convert

```
(define (move->direction move)
  (case move ((left) left) ((right) right) ((up) up) ((down) down)))
```

2.3.23 `make-stateful-graph-agent`

Procedure `(make-stateful-graph-agent start)` → `agent`

Description Make a graph-traversal agent that models the graph and searches it thoroughly, stopping when the world is clean.

The agent can detect cycles.

Parameters `start` Starting position of the agent (see 'random-start')

```
(define (make-stateful-graph-agent start)
  (make-reflex-agent
    start
    (let ((world (make-hash-table))
          (nodes (list->stack (list start)))
          (moves (make-stack)))
      (lambda (node clean?)
        (if (stack-empty? nodes)
          'noop
          (if (not clean?)
            'suck
            (let ((location
                    (hash-table-ref/default
                      world
                      node
                      (make-unknown-location clean?))))
              (if (stack-empty? moves)
                (hash-table-set! world node location)
                (let ((last-move (stack-peek moves)))
                  (if (eq? last-move 'backtrack)
                    (stack-pop! moves)
                    (if (eq? (stack-peek nodes) node)
                      (let ((last-move (stack-pop! moves)))
                        (vector-set!
                          (location-neighbors location)
                          (move->direction last-move)
```

```
                        no-passage))
                    (let* ((last-node (stack-peek nodes))
                           (last-location (hash-table-ref world last-node)))
                      (if (hash-table-exists? world node)
                        (stack-push! nodes cycle)
                        (begin
                          (hash-table-set! world node location)
                          (stack-push! nodes node)))
                      (vector-set!
                        (location-neighbors location)
                        (move->direction (reverse-move last-move))
                        last-node)
                      (vector-set!
                        (location-neighbors last-location)
                        (move->direction last-move)
                        node))))))
              (let ((new-moves
                      (map direction->move
                           (undiscovered-directions location))))
                (if (or (cycle? (stack-peek nodes)) (null? new-moves))
                  (begin
                    (stack-pop! nodes)
                    (if (stack-empty? moves)
                      'noop
                      (let ((move (stack-pop! moves)))
                        (stack-push! moves 'backtrack)
                        (reverse-move move))))
                  (let ((move (list-ref
                                new-moves
                                (bsd-random (length new-moves)))))
                    (stack-push! moves move)
                    move)))))))))
```

2.3.24 `simulate-graph`

Procedure `(simulate-graph world agent)` → `unspecified`
`(simulate-graph world agent steps)` → `unspecified`

Description Simulate the graph world.

Parameters

`world`	The world to simulate
`agent`	The agent to inhabit the world
`steps`	The steps to simulate (default: (default-steps))

```
(define simulate-graph
  (case-lambda
```

```
    ((world agent) (simulate-graph world agent (default-steps)))
    ((world agent steps)
     (parameterize
       ((randomize! bsd-randomize))
       (simulate
         (compose-environments
           (make-step-limited-environment steps)
           (make-debug-environment agent)
           (make-graph-environment world agent)
           (make-graph-performance-measure world agent)))))))
```

2.3.25 `simulate-graph/animation`

Procedure `(simulate-graph/animation world agent file)`
`(simulate-graph/animation world agent file steps)`
`(simulate-graph/animation world agent file steps width height font-size title)`

Description Simulate the graph world, creating an animation along the way; see, for instance, <http://youtu.be/EvZvyxAoNdo>.

Requires Graphviz.

Parameters

`world`	The world to simulate
`agent`	The agent that inhabits the world
`file`	The base-name of the animation file
`steps`	The steps to simulation (default: '(default-steps)'
`width`	Width of the animation in pixels
`hight`	Height of the animation in pixels
`font-size`	Font-size of the animation in points
`title`	Title of the animation

```
(define simulate-graph/animation
  (case-lambda
    ((world agent file)
     (simulate-graph/animation world agent file (default-steps)))
    ((world agent file steps)
     (simulate-graph/animation
       world
       agent
       file
       steps
       (default-width)
       (default-height)
       (default-font-size)
       (default-title)))
    ((world agent file steps width height font-size title)
     (let ((directory (create-temporary-directory)))
```

```
        (parameterize
          ((randomize! bsd-randomize))
          (simulate
            (compose-environments
              (make-step-limited-environment steps)
              (make-graph-animating-environment
                world
                agent
                directory
                width
                height
                font-size
                title)
              (make-finalizing-environment
                (make-animation-finalizer directory file)
                steps)
              (make-debug-environment agent)
              (make-graph-environment world agent)
              (make-graph-performance-measure world agent))))
        directory))))
```

2.3.26 `compare-graphs`

Procedure `(compare-graphs world agent-one title-one agent-two title-two composite-file)`
`(compare-graphs world agent-one title-one agent-two title-two composite-file ste`

Description Simulate two agents in a given world and animate their progress side-by-side; see, for instance, <http://youtu.be/B28ay_zSnoY>.

Requires Graphviz.

Parameters

`world`	The world to simulate
`agent-one`	The first inhabiting agent
`title-one`	Title of the first agent
`agent-two`	The second inhabiting agent
`title-two`	Title of the second agent
`composite-file`	Base-name of the composite animation

```
(define compare-graphs
  (case-lambda
    ((world agent-one title-one agent-two title-two composite-file)
     (compare-graphs
       world
       agent-one
       title-one
       agent-two
       title-two
```

```
        composite-file
        (default-steps)
        (/ (default-width) 2)
        (default-height)
        (/ (default-font-size) 2)))
      ((world agent-one
              title-one
              agent-two
              title-two
              composite-file
              steps
              width
              height
              font-size)
       (let ((directory-one
              (simulate-comparatively
                (copy-world world)
                agent-one
                steps
                width
                height
                font-size
                title-one))
             (directory-two
              (simulate-comparatively
                world
                agent-two
                steps
                width
                height
                font-size
                title-two)))
         (let ((composite-directory (create-temporary-directory)))
           (system*
             "cd ~a && for i in *; do echo $i; convert +append $i ~a/$i ~a/$i; done"
             directory-one
             directory-two
             composite-directory)
           ((make-animation-finalizer composite-directory composite-file)))))))
```