Some notes and solutions to Russell and Norvig's Artificial Intelligence: A Modern Approach (AIMA, 3rd edition)

August 20, 2012

Contents

1 DONE 1.1

`CLOSED:` *2011-10-10 Mon 03:03*

Intelligence A spontaneous faculty for associating impressions (more general than ideas); synthesizing abstractions from disparate stimuli; deducing conclusions from abstractions.

Intelligence is an emergent property of simples like e.g. neurons.

Artificial intelligence Mechanism for performing association, abstraction, deduction which appears to be spontaneous; may also be an emergent property of bit-pushing.

Agent Self-contained, autonomous input-processing mechanism.

Rationality The appropriate application of or *ratio*; this includes the mechanical process of deduction, as well as an ill-defined notion of common-sense.

Logical reasoning The mechanical aspect of rationality.

2 DONE 1.2

`CLOSED:` *2011-10-10 Mon 03:03*

The Mathematical Objection (3) still holds up: the halting problem; on the other hands, humans are also susceptible to the halting problem, aren't they? If one falls towards the humanity side of the humanity-rationality AI-axis, this deficit is reducible.

Lady Lovelace's Objection (6) is interesting: it denies *ex nihilo*; are genetic algorithms a counter-example?

The Argument from Informality of Behaviour (8) could be solved by fuzzy dispatch.

A modern refutation might be that there are not enough graduate students to make a satisfactory ontology of world-knowledge; thank the gods, then, for mechanical turks and unsupervised learning!

We came pretty damn close to 30% in the 2008 Loebner prize; why not double it to 60% in 2058? Despite Moore's law, let's say that AI proceeds

linearly.

3 DONE 1.3

`CLOSED:` *2011-10-12 Wed 12:58*

Reflex actions are rational in the sense that they are the result of induction on e.g. hot objects and the scientific method (see Turing); though the acquisition may require intelligence (induction, storage), the reflex itself is not intelligent in the sense that it requires no induction: it is immediate.

Reflex actions are not irrational, either, in the sense that someone does a cost-benefit analysis and decides to contravene it; let's call reflex actions *pararational*, therefore: neither rational nor irrational. There's no time to apply a utility function and behave accordingly (or discordingly (sic)).

4 DONE 1.4

Tom Evan's ANALOGY is an ad-hoc geometric solver, and would not therefore program. In people, you might be able to generalize from IQ-tests to success; but not so with domain-specific AI.

5 DONE 1.5

`CLOSED:` *2012-05-28 Mon 21:35*

Aplysia, Eric Kandel

20,000 neurons; memory-updates/second: 10^{-9}; cycle time: 10^15, high end;

Is memory-updates/second merely memory / cycle time? In which case: $20000/10^-9 = 10^5(20000)$ neurons, cycle time: 10^{-3}; memory updates per second? Not sure what the relationship between operations/sec and memory updates/sec; the former is an upper bound, though. Could it be that memory updates/sec is also bounded, somehow, by storage units? There is also the relationship between neuros and synapses.

In humans, 7,000 synapses per neuron; hence 10^{14} from 10^{11}. How many synapses per aplysia-neuron?

From this paper:

> On average, we found 24 contacts per pair of neurons.

Let's say, then, that sea slugs have 10^6 synapses; let's also say that, like humans, this is an upper bound on memory updates per second due to the e.g. refractory period.

That gives 10^6 memory updates per second; which means that a supercomputer houses the potential of 10^8 sea slugs.

6 DONE 1.6

`CLOSED:` *2012-05-28 Mon 21:43*

This post on the limits of introspection posits that:

> Mental processes are the results of opaque preferences, and . . . our own "introspected" goals and preferences are a product of the same machinery that infers goals and preferences in others in order to predict their behavior.

Accordingly, introspection is accurate to the extent that we can infer our own thoughts from the mental model we've extrapolated from watching others.

In other words, the processes which lead to thought are to thought opaque.

7 DONE 1.7

`CLOSED:` *2012-05-31 Thu 02:17*

Bar code scanners should hopefully be a trivial mapping from codes to products; if, on the other hand, you could scan and select similar products someone might be interested in: well, then.

The search engine problem is probably AI-complete; current solutions are some AI-complete-like heuristics.

Voice-activated telephone menus might be artificially intelligent in the sense that they have to recover signal from noise and make sense of it.

Internet routing algorithms are classic agents in the sense that they have environments (connection data), sensors (the ability to peer into network devices) and actuators (the ability to re-route traffic).

8 DONE 1.8

`CLOSED:` *2012-05-31 Thu 02:17*

Isn't it the case that humans do do some kind of implicit calculation? Another example is the ability to catch a ball: there are complex physics at play, and yet the human has evolutionarily honed and ad-hoc facilities to perform the same.

Something like Gaussian blur, in other words, is hard-coded into our neurons; vision system, on the other hand, don't have the advantage of fuzzy connections between analog neurons and have to simulate these biological heuristics with algorithms.

9 DONE 1.9

`CLOSED:` *2012-05-31 Thu 02:19*

Evolution might tend to result in systems that maximize certain utility functions (e.g. propagate genes, to that end: stay alive for a while; &c.); this process is pseudo-rational. Pseudo-rational in the sense that it is not rational for rationality's sake; but accidentally rational as it strives to maximize utility.

Maybe there's no distinction to be drawn there after all: ends justifying means.

10 DONE 1.10

`CLOSED:` *2012-05-31 Thu 02:28*

AI is science in the sense that it benefits from the scientific method (work done, for instance, on the relationship between goals and actions; cooperation; how brains cause minds; &c.) and precise mathematics.

AI is engineering, on the other hand, in the sense that it inheres in the world; it must find solutions in messy situations: solutions which might be approximate but nevertheless useful.

11 DONE 1.11

`CLOSED:` *2012-05-31 Thu 02:35*

"Surely computers . . . can do only what their programmers tell them" might have been the case, if it weren't for the fact that programmers can program machines to do things even they couldn't do (cf. chess programs that outstrip their masters).[1]

This seems like a paradox I don't adequately know how to explain; if it proceeds apace, prepare for the singularity.

12 DONE 1.12

`CLOSED:` *2012-05-31 Thu 02:41*

The relationship between nature and nurture is probably complex; suffice to say: genes might provide an upper bound on the intelligence of an animal that it has to strive to meet. Luck helps; so does discipline.

There is a nature-nuture/code-intelligence analogy only insofar as there is code that adapts to its environment; or a programmer can translate intelligence into code (bounded by the programmer's intelligence, of course).

13 DONE 1.13

`CLOSED:` *2012-05-31 Thu 02:48*

[1] See this article from 2007 on Google's machine translation system: "Using a brute-force statistical approach, the Google machine translation team has developed top performing translation software to and from languages that not even one of the teams members understands, such as Arabic and Chinese."

It's true that animals, humans and computers are bound by the laws of physics; nevertheless, there is this bizarre phenomenon of emergent behavior wherein the sum is more than its whole of parts.

Consciousness, after all, is an emergent behavior from the propagation of current through neurons; and the world-wide-web has emerged from a decentralized connection of web pages.

14 DONE 1.14

`CLOSED:` *2011-10-10 Mon 03:52*

1. The Japanese got this one; just a toy, though.
2. There is at least one driverless car in Cairo; it's not self-controlling, though, but rather remotely driven. Driving in clusterfuck-Cairo (like Athens) is taxing for humans, let alone AI. (Google's making political inroads in Nevada, though.) Sufficiently sensitive sensation of surrounding objects, conditions; physics; navigation; are required.
3. DARPA Grand Challenge
4. This robot fetches a sandwich.
5. Grocery IQ will order groceries; a week's worth, though?
6. Zia Mahmood got clowned once or twice; like poker, though, bridge is probabilistic and psychological.
7. TheoryMine is selling new computer-generated proofs for 15; standard objections apply.
8. The Bulhak-Larios Postmodernism Generator is funny; intentionally so?
9. Hilariously-named SHYSTER: ad-hoc expert system
10. Google Translate
11. Mechanically, but there is a human agent (telemanipulator); see this, though, where "In May 2006 the first AI doctor-conducted unassisted robotic surgery on a 34 year old male to correct heart arythmia."

15 DONE 1.15

`CLOSED:` *2012-05-31 Thu 03:05*

TREC appears to dissolve tracks as they become "solved" (e.g. the spam and terabyte tracks) and take new ones up as they emerge (e.g. the microblog and crowdsourcing tracks).

The Grand Challenge is attempting to solve the problem of driverless transportation (see Google's driverless car); despite recent legislation approving driverless cars (in e.g. California, Nevada, New Jersey), it is still cutting edge.

ICKEPS 2012, for instance, has a track for planning solar array operations on the ISS; seems relevant.

RoboCup is interesting in the sense that it requires advanced perception and cooperation among autonomous agents; I suspect it does not detract much from new ideas, despite the fact that it is still wrestling with some of the oldest (and unsolved) problems in AI (*vide supra*).

The Loebner Prize, on the other hand, seems a little anachronistic; do people care whether their AI counterparts really act human?

16 DONE 2.1

`CLOSED:` *2012-06-11 Mon 00:51*

It follows directly from the definition of a rational agent, which "maximizes its performance measure, given the evidence provided by the percept sequence" (p. 37), that its action "depends . . . on the time step it has reached."

This is because the lifetime of an agent is measured by the total number of percepts it receives [2] [3].

Let t be the time step the agent has reached; if $t \leq T$, the agent's performence measure depends upon the time step it has reached. If $t > T$, on the

[2] "Let T be the lifetime of the agent (the total number of percepts it will receive)" (p. 47). Percepts are the granularity of time.

[3] The current state of the environment is the same as the current percept, incidentally, if the environment is fully observable.

other hand, the rationality of the agent is undefined; since its performance measure is undefined.

At $t > T$, the agent has become pararational (neither rational nor irrational).

A rational agent's action, therefore, depends upon t only insofar as its performance measure depends upon t.

Take Opportunity, for instance, which had a performance measure of $T =$ 90 sol; as of 2012, it's overstepped T by eight years. If it fails after T to e.g. characterize soil, could you say that it acts rationally? In other words, is Spirit irrational; now that it has failed to meet its original performance measure?

No: by their original performance measure, Opportunity and Spirit are pararational; which is not to say that you couldn't define another performance measure u' depending upon another time T'.

See page 38, by the way, where the authors talk about rationality in terms of expected performance; could it be that an agent transcends T with respect to expected performance?

Example: given a penalty for each move, a reflex agent's expected performance would be just as good as any other's given T = 2; but not when T = 1000 (it would require a state-based agent to realize that the world is clean and stop moving).

17 DONE 2.2

`CLOSED:` *2012-06-11 Mon 17:18*

17.1 DONE a

`CLOSED:` *2012-06-11 Mon 17:18*

Page 38 describes an environment which is partially observable, deterministic and static; as such, the tabular agent in Fig. 2.3 can expect to maximize its utility in no more than four actions (the worst case is A: dirty, B: dirty; which results in either `suck, right, suck, left, ...` or `suck, left, suck, right, ...`).

There is no time to e.g. build a model of dirt, since the dirt doesn't replentish itself.

17.2 DONE b

`CLOSED:` *2012-06-11 Mon 17:18*

The agent does require internal state: it should know, for instance, whether it has cleaned every square; and, if so, should stop.

17.3 DONE c

`CLOSED:` *2012-06-11 Mon 17:18*

It should learn the geography of its environment to avoid wasting time trying to move off of it; it could maintain, furthermore, a dirt-distribution across the grid and favor those squares that tend to get dirty.

18 DONE 2.3

`CLOSED:` *2012-06-13 Wed 05:40*

a False. Page 42 mentions that, even in unobservable environments, "the agent's goals may still be achievable, sometimes with certainty;" the reflexive vacuum agent on page 38 is an example.

b True. In an unknown environment, there is no opportunity for the reflex agent to learn the "laws of physics" of the environment (p. 44); or for the programmer to endow the agent with them *a priori*.

c True. It's possible to imagine a task environment in which there are no decisions to be made: merely existing, for instance, satisfies the performance measure.

d False. According to page 46, the agent program takes the current percept; the agent function, on the other hand, takes the entire percept history.

e False. If the agent function is to e.g. determine whether a program will return an answer or run forever (see p. 8); it is not implementable

by a program/machine combination. Unless, of course, the author (or agent) has solved the halting problem.

f True. Take the performance measure, for instance, where an agent is supposed to simulate the roll of a fair-sided die.

g True. If an agent is a rational NxN tic-tac-toe player, it will perform just as well in a 2x2 as in a 3x3 environment.

h False. See a: page 138 describes a sensorless vacuum agent that knows the geography of its world; it's possible to search its belief space and even coerce the world into certain states.

i False. Even rational poker-playing agents fall prey to luck.

19 DONE 2.4

`CLOSED:` *2012-06-13 Wed 06:47*

19.1 Soccer

Performance measure Score and defend

Environment Field

Actuators Kicking, thrwing, catching

Sensors Topology, ball, agents

Characteristics Fully observable, multiagent, stochastic, sequential, dynamic, continuous, known

19.2 Titan

Performance measure Like TiME for surface lakes, it would determine the presence of biological compounds.

Environment Titan

Actuators Drill, satellite, landing gear

Sensors Mass spectrometer, camera

Characteristics Partially observable, multiagent? stochastic, sequential, dynamic, continuous, known

19.3 Shopping on the internet

Performance measure Finding used AI books

Environment The internet

Actuators Form completion, HTTP request, cookie storage

Sensors HTML parser

Characteristics Partially observable, multiagent, stochastic, sequential, dynamic, continuous, known

19.4 Playing a tennis match

Performance measure Winning the match

Environment Tennis court

Actuators Tennis racket

Sensors Location, trajectory of ball, opponent; topology

Characteristics Fully observable, multiagent, stochastic, sequential, dynamic, continuous, known

19.5 Practicing tennis against a wall

Performance measure Length of rally

Environment Half-court with wall

Actuators See above.

Sensors See above (sans opponent).

Characteristics Fully observable, single agent, stochastic, sequential, dynamic, continuous, known

19.6 Performing a high jump

Performance measure Height jumped

Environment Measuring stick

Actuators Spring

Sensors Balance

Characteristics Fully observable, single agent, deterministic, episodic, static, continuous, known

19.7 Knitting a sweater

Performance measure Consistency of stitch, conformance to the recipient's body

Environment Yarn, recipient's body

Actuators Needle

Sensors Yarn on needle

Characteristics Fully observable, single agent, deterministic, sequential, static, continuous, known

19.8 Bidding on an item

Performance measure Win, save cash

Environment Auction

Actuators Signify bid

Sensors See the artifact, understand the auctioneer

Characteristics Partially observable [4], stochastic, sequential, dynamic, continuous, known

[4] The agent's minds are unobservable: we have to operate in belief space.

20 DONE 2.5

CLOSED: *2012-06-14 Thu 06:44*

Agent An agent is a black box with inputs and outputs that conspires to perform something

Agent function The agent function maps inputs to outputs.

Agent program The agent program implements the agent function.

Rationality Rationality usually means the application of reason; but because the authors have given up on AI as "thinking humanly" (p. 2), it has been cheapened to mean: "act in accordance with this performance measure we've set up."

Autonomy Autonomy is the ability of an agent to select actions beyond the *a priori* programming of its maker.

Reflex agent A reflex agent acts according to the immediate percept; it has amnesia.

Model-based agent A model-based agent acts according to a model of the world it has synthesized from percepts.

Goal-based agent Not merely reacting to the environment (or its model thereof), the goal-based agent has a Vorhaben (so to speak) that can inform sequences of actions.

Utility-based agent Utility-based agents have internalized their own performance measure; and, as such, are able to decide between conflicting goals.

Learning agent Learning agents hone their sense of appropriate actions by modifying the weights associated with environmental features.

21 DONE 2.6

CLOSED: *2012-06-14 Thu 12:06*

a There are infinite agent programs that implement a given agent function; take, for instance, an agent that perceives flashes of light and maps them to some output (say, an integer).

The percept sequence could be mapped as an integer encoded as a bit-string of light and dark moments; or a bit-array representing the same thing.

b Yes; an agent function whose performance measure is to determine whether a program stops or not cannot be implemented as a progrm (unless one first solves the Halting Problem).

c Yes; which is to say: a program implements a mapping from percepts to actions; to change the mapping, you have to change the program.

d There would be 2^n possible agent programs on an n-bit machine (not all of them functional).

(According to this, there are a^{2^n} possible programs; 2^n possible states and a choices for each state. I don't think they're factoring the program into the storage, are they?)

e Speeding up the agent program does not change the agent function; they are orthogonal: the former is concrete, the latter abstract.

If they don't behave like Hegelian dialectic, they are at least Platonic forms and instantiations.

22 DONE 2.7

`CLOSED:` *2012-06-20 Wed 03:52*

Clever: the goal-based agent mutates belief-space based on its best guess; acts accordingly.

22.1 Goal-based agent

```
- let
  - state
  - model
  - goals
  - action
    - define (goal-based-agent percept)
      - set! state (update-state state action percept model)
      - let
```

```
        # Shouldn't we distinguish between many different
        action-sequences; and, if so, how to do so without a utility
        function: evaluate them against the performance measure?
        - action-sequence (search goals state)
          - return (first action-sequence)
```

22.2 Utility-based agent

```
- let
  - state
  - model
  - goals
  - action
    - define (utility-based-agent percept)
      - set! state (update-state state action percept model)
      - let
        - probabilities (map probability goals)
        - utilities (map utility goals)
          - let
            - expected-utilities (map * probabilities utilities)
            - goal-of-maximum-expected-utility (max goals expected-utilities)
            - action-sequence (search goal-of-maximum-expected-utility state)
              - return (first action-sequence)
```

23 DONE 2.8

CLOSED: *2012-06-21 Thu 04:39*

- CLOSING NOTE *2012-06-21 Thu 04:39*
  See aima-chicken.

```
(use debug
     foof-loop
     lolevel
     srfi-1
     srfi-8
     srfi-13
     srfi-69
     vector-lib)
```

```
(define (simulate environment)
  (loop ((while (environment)))))

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

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

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

;;; What about pairs of objects and optional display things.
(define make-debug-environment
  (case-lambda
   ((object) (make-debug-environment object pp))
   ((object display)
    (lambda () (display object)))))

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

(define clean #t)
(define clean? identity)

(define dirty #f)
```

```
(define dirty? (complement clean?))

(define left 0)
(define left? zero?)

(define right 1)
(define right? (complement zero?))

(define make-vacuum-world vector)

(define vacuum-world-location vector-ref)

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

(define-record vacuum-agent
  location
  score
  program)

(define-record-printer vacuum-agent
  (lambda (vacuum-agent output)
    (format output
            "#(agent ~a ~a)"
            (if (left? (vacuum-agent-location vacuum-agent))
                'left
                'right)
            (vacuum-agent-score vacuum-agent))))

(define (make-vacuum-environment world agent)
  (lambda ()
    (let* ((location (vacuum-agent-location agent))
           (action ((vacuum-agent-program agent)
                    location
                    (vacuum-world-location world location))))
      (case action
        ((left) (vacuum-agent-location-set! agent left))
        ((right) (vacuum-agent-location-set! agent right))
        ((suck) (vacuum-world-location-set! world location clean))
        (else (error (string-join
                      "make-vacuum-environment --"
```

```
                     "Unknown action")
                    action))))))

(define (reflex-vacuum-agent-program location clean?)
  (if clean?
      (if (left? location)
          'right
          'left)
      'suck))

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

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

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

(define simulate-vacuum
  (case-lambda
   ((world agent) (simulate-vacuum world agent 1000))
   ((world agent steps)
    (simulate
     (compose-environments
      (make-step-limited-environment steps)
      (make-performance-measuring-environment
       (make-vacuum-performance-measure world)
       (make-vacuum-score-update! agent))
      (make-debug-environment agent)
```

```
      (make-debug-environment world vacuum-world-display)
      (make-vacuum-environment world agent)))
    (vacuum-agent-score agent))))

(simulate-vacuum (make-vacuum-world dirty clean)
                 (make-reflex-vacuum-agent
                  left
                  (lambda (location clean?)
                    'right))
                 10)
```

I want environmental combinators, incidentally; such that I can compose an e.g. step-limited environment with an agent with a vacuum one.

We can compose steps; but how do you compose score: do you have to specify a reducer of some kind; e.g. addition? Is it really environment reduction we're talking about here?

I'm beginning to suspect that the performance score is a property of the agent, not the environment; this is consistent with the book's use of "reward" and "penalty." It also makes sense in a multi-agent environment.

On page 37, however, the authors state that:

> This notion of desirability [for a sequence of actions leading to a sequence of states] is captured by a **performance measure** that evaluates any given sequence of environment states.

I suspect that, whereas the environment is an arbiter of the performance score (i.e. applies the performance measure), the score inheres in the agents.

This is corroborated by the following:

> Notice that we said *environment* states, not *agent* states. If we define success in terms of agent's opinion of its own performance, an agent could achieve perfect rationality simply by deluding itself that its performance was perfect.

Since only the environment has access to its true states, it alone can measure performance. Is this problematic in cases where we don't have an omniscient environment that directly communicates performance scores? In such cases, we'd have to rely on the imperfect self-judgement of the agent; and attempt to converge on rationality by internal coherence.

What I'm calling environments, incidentally, are now just functions: step-functions, at that; and can be reduced by `every`.

Agent combinators are a little tough, though; the performance measure has to be aware of the combined features. Can we use some kind of message-passing mechanism?

What stops us, for instance, as modelling the agents as lambdas; too? Part of the problem is the inversion of control: we'd have to pass a message to the agent to store its score, as opposed to manipulating the score directly.

Every agent would be a dispatch-mechanism that would manage its own meta-variables (including score and e.g. location) on the basis of messages. Is it problematic, however, to have agents managing their own score? Could we have an agent $\rightarrow$ score mapping in the environment itself? That way, agents only maintain state according to its percepts.

Score, for instance, is not a percept in the vacuum world; location, however, is. Agents, then, are functions with closures; functions which take as many parameters as their percepts have components. The performance-measuring-environment, therefore, maintains an `agent->score` table. Yes!

Problem is, though, that we'd have to break the nice contract we have: environments are niladic lambdas. To maintain the performance measure table, we'd have to receive the agent and the new score.

How to make the performance measure part of the environment, so that we can relieve the agent from metadata?

By taking the metadata out of the agent, we have to maintain agent $\rightarrow$ metadata mappings in the environment; this is kind of a pain in the ass.

By maintaining agents-as-lambda, we get a certain flexibility; on the other hand, we shunt some complexity onto the environment: as it has to maintain agent-metadata: score, location, &c.

Is this an acceptable tradeoff? The alternative, where I need to guess what agents need (program, score, location) seems onerous; for some reason. In practice, however, it may be simpler. We haven't even solved the agent-hashing-problem, for instance (wherein hashing fails if we mutate a field).

Can we hash closures?

I want to follow this environment-maintains-agent-¿metadata-mapping thing and see how far it goes. (I see now why objects are interesting; closures, of

course, do the same thing.)

If `make-*-environment` returned multiple values: the thunk followed by e.g. `agent->score`, `agent->location`; you can ignore the latter values, if you want to.

Or, we can demand that the user furnish them; better yet, we can give the user the option of furnishing and ignoring them.

Also, shouldn't we be able to name agents at some point? This would also have to fall within an external data structure. Maybe the record solution isn't problematic if we create ad-hoc agents for each problem.

If we really need to decouple the program from the agent metadata (do we?), one solution is to have an agent-¿metadata table in the environment; the metadata would be a record containing location, score, name, &c.

This metadata table, on the other hand, would have to be passed to each subenvironment for composition. Seems like a pain.

We found that, since environments consist of a step function, we could reduce them to a lambda; let's see if this continues to be the case. For the time being, however, I think using agent-records is simplifying.

I wouldn't mind agents being lambdas with closures; problem is: can't access the closure without some kind of message passing. (Message passing simulates records.) We could possibly do it with some kind of multiple-return-values hack, in which the subsequent values are ignored (the agent effectively does a state dump every time its program is invoked). The problem with that is that I have to pass a percept in to access its state, or store its state some other way.

To avoid namespacing everything (like e.g. `vacuum-agent`, &c.), I'd like to have separate modules; that way, if we need to, we can import with a prefix.

For learning purposes, we should allow the student to specify no more than the agent program; worry about all the bootstrapping on the back end.

We may have to copy worlds, incidentally, to compare how e.g. reflex- vs. state-agents behave; thank goodness for `vector-copy`. (Copy by default?)

To give feedback to students, should have an e.g. `environment-print` that we can pass around (this sort of function-passing, incidentally, is what Norvig sought to avoid); `environment-print` might happen at every step in e.g. `simulate`. Oh, `make-debugging-environment`.

24 DONE 2.9

CLOSED: *2012-06-28 Thu 12:47*

Using the aima-chicken framework:

```
(use aima-vacuum
     test)

(let ((worlds
       (list (make-world clean clean)
             (make-world clean clean)
             (make-world clean dirty)
             (make-world clean dirty)
             (make-world dirty clean)
             (make-world dirty clean)
             (make-world dirty dirty)
             (make-world dirty dirty)))
      (agents
       (list (make-reflex-agent left)
             (make-reflex-agent right)
             (make-reflex-agent left)
             (make-reflex-agent right)
             (make-reflex-agent left)
             (make-reflex-agent right)
             (make-reflex-agent left)
             (make-reflex-agent right))))
  (let* ((scores (map simulate-vacuum worlds agents))
         (average-score (/ (apply + scores) 8)))
    (test
     "Scores for each configuration"
     scores
     '(2000 2000 1998 1999 1999 1998 1996 1996))
    (test
     "Average overall score"
     1998.25
     average-score)))
```

25 DONE 2.10

CLOSED: *2012-06-29 Fri 17:52*

25.1 DONE a

CLOSED: *2012-06-29 Fri 13:29*

With a partially observable environment, a simple reflex agent will not be rational (in the sense that its expected performance is not as good as any other's); in other words, it should be scoring about twice as much as this:

```
(use aima-vacuum
     test)

(test
 "Penalizing vacuum with reflex agent"
 998
 (simulate-penalizing-vacuum (make-world dirty dirty)
                             (make-reflex-agent left)))
```

The reflex agent would require state to determine that e.g. the world was clean and that it didn't need to move anymore.

25.2 DONE b

CLOSED: *2012-06-29 Fri 17:52*

```
(use aima
     aima-vacuum
     test
     vector-lib)

(debug? #f)

(define-record unknown)

(define unknown (make-unknown))

(define (all-clean? world)
```

```
  ;; Vector bleeds a little world.
  (vector-every (lambda (location) (clean? location)) world))

(test
 "Stateful agent in penalizing environment"
 1995
 (simulate-penalizing-vacuum
  (make-world dirty dirty)
  (make-reflex-agent
   left
   ;; We could also make an initial pessimistic hypothesis of all-dirty.
   (let ((world (make-world unknown unknown)))
     (lambda (location clean?)
       (if clean?
           (begin
             ;; Extra work here every time; otherwise, we'd have an
             ;; extra `all-clean?' check after we set the state.
             ;; `vector-set!', I'd wager, is cheaper than
             ;; `all-clean?'.
             (vector-set! world location clean)
             (if (all-clean? world)
                 ;; Symbols appropriate here, or should we have predefined
                 ;; go-left, go-right, clean, do-nothing? We're message
                 ;; passing, after all; I suppose a lambda wouldn't make any
                 ;; sense?
                 ;;
                 ;; Can't be lambdas unless we redefine e.g. `go-right'
                 ;; to penalize in the case of
                 ;; `make-penalizing-environment'; better to keep as
                 ;; symbols and dispatch, right? There should be some
                 ;; sort of data-directed model we could use, though,
                 ;; instead of the case-based dispatch.
                 'noop
                 (if (right? location)
                     'left
                     'right)))
           'suck))))))

(test
 "Stateful agent in penalizing environment (from the egg)"
```

```
1995
(simulate-penalizing-vacuum
 (make-world dirty dirty)
 (make-stateful-reflex-agent left)))
```

25.3 DONE c

CLOSED: *2012-06-29 Fri 17:52*

- CLOSING NOTE *2012-06-29 Fri 17:52*
  Should we actually implement it?

If the simple and stateful reflex agents are omniscient w.r.t. the environment, they are equivalent; the stateful agent will simply update its state according to its omniscient percept and the simple one will simply act accordingly.

26 DONE 2.11

CLOSED: *2012-07-16 Mon 16:59*

26.1 DONE a

CLOSED: *2012-07-16 Mon 14:00*

A simple reflex agent wouldn't be able to explore an environment of unknown extent without exhausting all possible paths of the corresponding $n \times n$ space; given sufficient time, such an exhaustive agent would asymptotically approach rationality toward $t = \infty$.

Given reasonable time constraints, however, or e.g. penalties for moving, such an agent would not be rational; if it maintained all possible paths in a table, it would also contravene the directive on p. 47:

> The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a smallish program rather than from a vast table.

More fundamentally, an agent wouldn't be able to exhaust the space without maintaining some sort of state (e.g. paths traversed).

Even more fundamentally, however, the agent can't discern whether it's hit a wall; this changes in 2.12, however, when the agent gets a bump sensor.

26.2 DONE b

CLOSED: *2012-07-16 Mon 14:00*

The average score for a randomized agent in a 20-node world is roughly 17300.0; see the demonstration-video.

```
(use aima aima-vacuum test)

(parameterize ((current-test-epsilon 0.005))
  (test
   "Test the randomized graph agent on 100 different worlds."
   17300.0
   (let* ((scores
           (list-tabulate
            100
            (lambda (i)
              (let* ((world (make-graph-world))
                     (start (random-start world))
                     (agent (make-randomized-graph-agent start)))
                (parameterize ((random-seed i)
                               (debug? #f))
                  (simulate-graph world agent))
                (agent-score agent))))))
     (/ (apply + scores) (length scores)))))
```

26.3 DONE c

CLOSED: *2012-07-16 Mon 14:49*

The randomized agent will perform poorly in a linear environment, since many of its movement choices will be no-op; in fact, the average score of a randomized agent on a linear world of 20 nodes is roughly 15000.0 ($\approx 13\%$ less than the random 20-node world in 2.11b):

```
(use aima aima-vacuum debug test)
```

```
(parameterize ((current-test-epsilon 0.1))
  (test
   "Test the randomized graph agent on a linear world 100 times."
   15000.0
   (let* ((world (make-linear-world))
          (start (random-start world)))
     (let ((scores (list-tabulate
                    100
                    (lambda (i)
                      (let* ((world (copy-world world))
                             (agent (make-randomized-graph-agent start)))
                        (parameterize ((debug? #f))
                          (simulate-graph world agent))
                        (agent-score agent))))))
       (/ (apply + scores) (length scores))))))
```

See the video.

26.4 DONE d

CLOSED: *2012-07-16 Mon 16:58*

An agent with state can outperform a stateless agent and maximize its performance by systematically exploring the environment la e.g. depth-first search; in a 20 node environment, the stateful agent performs ≈ 20% better than its randomized counterpart (cf. 2.11c).

See the video.

```
(use aima aima-vacuum test)

(parameterize ((current-test-epsilon 0.005))
  (test
   "Test the stateful graph agent on 100 different worlds."
   19176.35
   (let* ((scores
           (list-tabulate
            100
            (lambda (i)
              (let* ((world (make-graph-world))
                     (start (random-start world))
```

```
                    (agent (make-stateful-graph-agent start)))
                (parameterize ((random-seed i)
                               (debug? #f))
                  (simulate-graph world agent))
                (agent-score agent))))))
    (/ (apply + scores) (length scores)))))
```

The basic algorithm is as follows:

1. Is the current location dirty? Clean it.
2. Otherwise, visit and clean (if necessary) all the current-location's unvisited neighbors.
3. If there are no unvisited neighbors for the current location, go back the way we came.
4. If there are no unvisited (and uncleaned) locations, stop.

Traversing the world in this fashion is linearly complex.

27 DONE 2.12

`CLOSED:` *2012-07-16 Mon 17:09*

A simple reflex agent with a bump sensor will perform just as well as a random agent since, upon detecting a bump, it can randomly change directions. The same constraints on random agent apply: e.g. poor performance in linear spaces.

The state agent is fundamentally unchanged: instead of deducing implicit bumps, however, due to non-movement (requiring e.g. a bump-sentinel in the movement stack); it can incorporate the bump-data directly into its state.

If the bump sensor stops working, unfortunately, the agent would have to fall back on random behavior (see 2.11b).

28 DONE 2.13

`CLOSED:` *2012-07-16 Mon 17:20*

28.1 DONE a

CLOSED: *2012-07-16 Mon 17:20*

By expending an extra step at each square to make sure that it is, in fact, clean (and repeatedly cleaning otherwise); one can still clean the entire environment in linear time.

If the dirt sensor is wrong some percentage of the time, one would repeatedly sense the status of the location until some significance criterion is reached. For instance, it appears as though one would have to sense the status 17 times per location to achieve a confidence level of 95%.

28.2 DONE b

CLOSED: *2012-07-16 Mon 17:20*

I don't see how you could avoid repeatedly exploring and cleaning the world according to some acceptable interval; if the dirt is not evenly distributed, store statistics about dirty hotspots (and hit those locations more frequently).

29 DONE 3.1

CLOSED: *2012-08-19 Sun 03:42*

Formally, a well-defined problem (see page 66) contains a goal-test; such that we can't define a problem until we've formulated a goal.

Page 65 states, furthermore, that "problem formulation is the process of deciding what actions and states to consider, given a goal;" searching for solutions can't proceed without goal-formulation, either.

30 DONE 3.2

CLOSED: *2012-08-19 Sun 19:23*

30.1 DONE a

CLOSED: *2012-08-19 Sun 19:23*

States A graph whose nodes point east, west, north, south (pointing to a special sentinel-node in the case of adjoining walls); the orientation of the robot; the square currently occupied by the robot.

Initial state Graph of one central node; northward orientation

Actions Turn east, west, north, south; move forward.

Transition model Moves in the currently oriented direction wherever a wall does not intervene; where a wall intervenes, however, the robot stays put.

Goal test Is the robot out of the maze?

Path cost None

The state space is infinite: you can reach any square, for instance, over an unlimited number of nops (e.g. pressing forward into a wall, turning redundantly, &c.).[5]

Let's say that redundant nops have been pruned, however; and that the graph, furthermore, doesn't have any cycles: the state space is at least n, corresponding to the number of nodes in the graph. It is at least $4n$, though, since the robot can be in any of the four orientations in each square. Let's say, furthermore, for every node n_i there are m_i unique paths (i.e. sequences $n_{0,1,\ldots,i}$) from the initial node n_0 to n_i: the state space for n_i alone becomes all the ways to fulfill each path in m_i including unnecessary turns and false-forwards into walls. Let the set of all fulfillments for a given m_i be M_i.

The state space is the sum of M_i over the number of nodes in the graph; plus the robots current position; plus the robot's orientation.

30.2 DONE b

CLOSED: *2012-08-19 Sun 19:23*

[5] From page 67: "The state space . . . is the set of all states reachable from the initial state by any sequence of actions."

States A graph whose nodes point to one or more of east, west, north, south if a neighbor exists in that direction; the orientation of the robot; the square currently occupied by the robot.

Initial state Graph of one central node; northward orientation

Actions Turn east, west, north, south if the robot is at an intersection; or move forward.

Transition model Moves in the currently oriented direction wherever a wall does not intervene; where a wall intervenes, however, the robot stays put.

Goal test Is the robot out of the maze?

Path cost None

The state-space is still infinite, since the problem admits of redundant forwards at intersections; the pruned state space is smaller than 3.2a, however, since the fulfillment of m_i doesn't involve unnecessary turns in corridors.

30.3 DONE c

`CLOSED:` *2012-08-19 Sun 19:23*

States A graph whose nodes point to one or more of east, west, north, south if a neighbor exists in that direction; the square currently occupied by the robot.

Initial state Graph of one central node

Actions Turn east, west, north, south

Transition model Move east, west, north, south until the robot is at a turning point.

Goal test Is the robot out of the maze?

Path cost None

If the maze has loops, the state space is infinite; otherwise, the state-space is significantly smaller than even 3.2b, since M_i doesn't require any superfluous movement to exhaust it.

The robot's orientation is irrelevant, since we'll travel along the chosen direction to the next turning point.

30.4 DONE d

`CLOSED:` *2012-08-19 Sun 19:23*

At least three simplifications; that:

1. the maze is oriented along strict cardinal directions;
2. the passages are straight;
3. the passages are passable;
4. the robot has unlimited energy.

31 TODO 3.3

31.1 DONE a

`CLOSED:` *2012-08-20 Mon 04:29*

State A state specifies the location of each friend on the map $n(\{i,j\})$, and the respective paths that led them there $P(\{i,j\})$.

Initial state A friend a each city $n(\{i,j\})_0$, $P(\{i,j\})$ are empty.

Actions Each friend moves to a neighboring city [6].

Transition model Each friend moves to a neighboring city.

Goal test $n(i) = n(j)$, the two friends are in the same city.

Path cost $max(T(i_n, i_{n+1}), T(j_n, j_{n+1}))$, where T is the time required to move from one city to the next.

This is reminiscent of bidirectional search (§3.4.6) with a heuristic.

31.2 DONE b

`CLOSED:` *2012-08-20 Mon 04:29*

An admissible heuristic is one that never overestimates the cost to reach a goal.

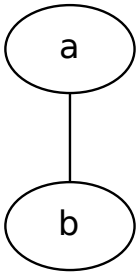
[6] Nop, where one friend stays put; is apparently not an option. If it were, there wouldn't be cases where vacillation is required to convene.

1. $D(i, j)$, the straight-line-distance heuristic, is admissible (by the triangle inequality).
2. $2D(i, j)$, on the other hand, is not admissible; since it might overestimate the cost of reaching the goal.
3. $D(i, j)/2$ is admissible, since $D(i, j)/2 < D(i, j)$ and $D(i, j)$ is admissible.

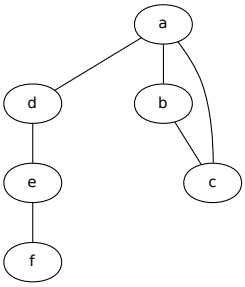
31.3 DONE c

`CLOSED:` *2012-08-20 Mon 04:29*

If nop is allowed, no completely connected map exists for which there is no solution: one friend stays still, the other meets her; if nops are not allowed, on the other hand, even a completely connected graph with two-nodes has no solution (since the problem states that i and j are different cities).


a
b

31.4 TODO d



32 TODO 3.7

See convex hulls.

33 Meetups

33.1 Mon Jun 11 2012

- Had to redefine the rational from "exercizing reason" to "maximizing utility function" because they gave up an AI as thinking machines in

the 60s.

- Mitochondria were once autonomous agents; cells as composite agents
- Thin vs. thick agents and skynet
- In games like poker, the mind of the adversarial agents are part of the environment; requires a theory of mind to discern things like: "is he bluffing?"

33.2 Mon Jun 18 2012

- David has the international version, which would have you write an essay on evolution and autonomy; see e.g. Turing on child AI:

> We have thus divided our problem into two parts. The and child-programme the education process. These two remain very closely connected. We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications
>
> **Structure of the child machine** Hereditary material
>
> **Changes** Mutations
>
> **Natural selection** Judgment of the experimenter
>
> One may hope, however, that this process will be more expeditious than evolution. The survival of the fittest is a slow method for measuring advantages. The experimenter, by the exercise of intelligence, should be able to speed it up. Equally important is the fact that he is not restricted to random mutations. If he can trace a cause for some weakness he can probably think of the kind of mutation which will improve it.

33.2.1 CANCELED Test a simple agent in each (Python, Java, Clojure) implementation.

`CLOSED:` *2012-06-20 Wed 03:19*

- CLOSING NOTE *2012-06-20 Wed 03:19*
  Looks like we're going to standardize on Clojure.

33.2.2 DONE Get some standard cables to connect to the projector.

`CLOSED:` *2012-06-20 Wed 03:19*

33.2.3 DONE See if we can use `xrandr` to get twin-view with an external HDMI.

`CLOSED:` *2012-06-15 Fri 06:04*

```
xrandr --output eDP1 --off
xrandr --output HDMI1 --mode 1280x720
```

33.3 Mon Jun 25 2012

33.3.1 Discussion

- 2.1
  Change performance measure; utility-based agent aware of its own performance measure: can react accordingly? Not a reflex agent, though, that's pre-programmed.

  Example: given a penalty for each move, a reflex agent's expected performance would be just as good as any other's given T = 2; but not when T = 1000 (it would require a state-based agent to realize that the world is clean and stop moving).

- 2.2

  c Memory, motor, bump sensor (or penalty); learn geography, probability of becoming dirty. Clustering algorithm: centers of mass for dirt.

- 2.3

  a Best action given available information.

  c One-square, not dirty.

  d Page 51 (program) vs. page 36 (function).

  f See **c** above.

  g Stochastic vs. deterministic vacuum world: reflex agents are still rational. Performance measure still the same.

  i Even an omniscient poker player is subject to luck.

33.3.2 TODO Get a minimal Clojure example up here.

33.3.3 TODO Set up `csrg.org` with a mailing list.

This is an alternative to e.g. Google groups and whatever mechanism Meetup has.

(Or is it `csrg.com`? It is indeed `csrg.org`.)

33.4 Mon Jul 2 2012

- Each parameter in the agent-program corresponds to a sensor.
- Should we pass a status-variable into the sensor instead of a boolean? Might be more consistent with the singletons. Nah, fuck it.

33.5 Tue Jul 24 2012

- Michael mentioned that the triangle-inequality based consistency (monotonicity) condition states merely that the heuristic function monotonically decrease as you approach the goal.

33.6 Mon Aug 6 2012

33.6.1 3.1

Stock example:

Initial state One P&G share

Actions available Hold and sell

Transition model Hold, retain the share; sell, lose the share

Goal test P&G is up 10%.

Path cost Cost of selling: dividends, &c.

Simple agent, given historical data, might determine that, in n days time, P&G is up; then blindly executes $n-1$ holds followed by one buy.

How to apply to crossfit?

34 Notes

34.1 1

- Two dimensions: thought vs. action, humanity vs. rationality.
- Physical simulation of a person is unnecessary for intelligence.
  - Mind-body dualism of Descartes?
- Cognitive science brings together computer models from AI and experimental techniques from psychology.
- Real cognitive science, however, is necessarily based on experimental investigation of actual humans.
- The standard of rationality is mathematically well defined and completely general.
- We will adopt the working hypothesis that perfect rationality is a good starting point for analysis.
- Limited rationality: acting appropriately when there is not enough time

- Materialism, which holds that the brain's operation according to the laws of physicas constitutes the mind.
- Logical positivism
- Carnap, The Logical Structures of the World, was probably the first theory of mind as a computational process.
- Intelligence requires action as well as reasoning.
- Actions are justified by a logical connection between goals and knowledge of the action's outcome.
- Regression planning system
- The leap to a formal science required a level of mathematical formalization: logic, computation, probability.
- The world is an extremely large problem instance.
- Models based on satisficing—making decisions that are "good enough"—gave a better description of actual human behavior.
- Searle: brains cause minds.
- Behaviorism
- "A cognitive theory should be like a computer program."
- Intelligence and an artifact
- Parallelism—a curious convergence with the properties of the brain.
- The state of a neuron was conceived of as "factually equivalent to a proposition which proposed its adequate stimulus." McCulloch and Pitts (1943)
  - Neural events and the relations among them can be treated by means of propositional logic.
  - For any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes.
  - For every net behaving under one assumption, there exists another net which behaves under the other and gives the same results.
- Perhaps "computational rationality" would have been more precise and less threatening, but "AI" stuck.

- AI from the start embraced the idea of duplicating human faculties such as creativity.
- John McCarthy referred to this period as the "Look, Ma, no hands!" era.
- "A physical symbol system has the necessary and sufficient means for general intelligent action."
- 1958 . . . McCarthy define Lisp, which was to become the dominant AI programming language for the next 30 years.
- It is useful to have a formal, explicit representation of the world and its workings and to be able to maniplutae that representation with deductive processes.
- McCarthy, Programs with Common Sense
  - In this program the procedures will be described as much as possible in the language itself and, in particular, the heuristics are all so described.
  - If one wants a machine to be able to discover an abstraction, it seems most likely that the machine must be able to represent this abstraction in some relatively simple way.
  - The improving mechanism should be improvable.
  - Must have or evolve concepts of partial success.
    - Something about ˜1995 that made for a cute blog.
  - For example, to mest people, the number 3812 is not an object: they have nothing to say about it except what can be deduced from its structure. On the other hand, to most Americans the number 1776 is an object because they have filed somewhere the fact that it represents the year when the American Revolution started.
  - One might conjecture that division in man between conscious and unconscious thought occurs at the boundary between stimulus-response heuristics which do not have to be reasoned about but only obeyed, and the others which have to serve as premises in deductions.
- Machine evolution (genetic algorithms): Friedberg, 1958, 1959.

  - Friedberg. 1958. A learning machine Part 1. IBM Journal of Research and Development, 2, 2–13.
    * From and intent, to be sure, are related quite discontinuously in the compact, economical programs that programmers wrte.
  - Friedberg, Dunham, North. 1959. A learning machine, Part 2. IBM Journal of Research and Development, 3, 282–287.
- Failure to come to grips with the "combinatorial explosion"
- The new back-propagation learning algorithms for multilayer netwrks that were to cause an enormous resurgence in neural-net research in the late 1980s were actually discovered first in 1969.
- Bruce Buchanan: a philosopher turned computer scientist
- DENDRAL was the first successful knowledge-intensive system (expert system).
- AI Winter
- Parallel Distributed Processing (Rumelhart, McClelland. 1986)
- Connectionist models: competitors to symbols models and logicist approach
- Ones that act rationally according to the laws of decision theory and do not try to imitate the thought steps of human experts
- Control theory deals with designing devices that act optimally on the basis of feedback from the environment.

34.2 2

- Rational agents
- Agents behaves as well as possible (utility function?)
- Agent perceives its environment through sensors and acts through actuators.
  - Hands are actuators and sensors.
- Percept :: agent's perceptual inputs at any given instant

- Agent's choice depends on percept sequence to date.
- Agent function :: maps percept sequence to action.
- External characterization of agent (agent function): table mapping percept sequences to actions; internally: agent program.
- In a sense, all areas of engineering can be seen as designing artifacts that intaract with the world.
  - Trivializing agents to view e.g. calculators as such.
- Intelligent agents, on the other hand: non-trivial decision making.
- Rational agents: does the right thing (utility).
- Performance measure
  - (This all sounds reminiscent of Mitchell, by the way.)
- Sequence of actions causes the environment to go through states: environmental states are distinct from agent states.
  - Basing performance merely off of agent-states is a form of coherentism.
- Design performance measures according to what one actually wants in the environment.
- "We leave these question as an exercise for the diligent reader."
  - Classic.
- Rationality: performance measure, agent's prior (i.e. *a priori*) knowledge, agent's actions, agent's percept sequence.[7]
  - "Percept," it turns out, is the converse of "concept": "A Percept or Intuition is a single representation . . . a Concept is a collective (general or universal) representation of a whole class of things." (F. C. Bowen Treat. Logic)
- For each percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given its percept sequence and a priori knowledge.
- Omniscience vs. rationality

[7] Bizarre to me that a programmer is responsible for the *a priori*; playing god, anyone?

- Rationality maximizes *expected* performance; perfection, *actual* performance.
- Our definition of rationality does not require omniscience.
  - It's possible sometimes, by the way, to detect transitions in authorship.
- Information gathering: actions in order to modify future percepts.
- *a priori* rather than percepts: lacks autonomy.
- Ration agent: autonomous; boostrap with *a priori*, though.
- Just as evolution provides animals with built-in reflexes to survive long enough to learn for themselves
- Task environments
- PEAS :: Performance, Environment, Actuators, Sensors
  - Mitchell has: task, performance measure, training experience, target function, target function representation.
- Fully observable vs. partially observable environment.
- Task environment effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action, performance measure.
- Single agent vs. multiagent
- Entity *may* vs. *must* be viewed as an agent.
- Competitive vs. cooperative multiagent environment
- Communication
- In some competitive environments, randomized behavior is rational because it avoids predictability.
- Deterministic vs. stochastic environment
- "Uncertain" environment: not fully observable or not deterministic
- Stochastic: uncertainty about outcomes quantified in terms of probabilities; nondeterministic: actions characterized by possible outcomes, no probabilities attached.

- Episodal vs. sequential: atomic episodes: receives percept and performs single action; sequential: current decision affect all future decisions.
- Static vs. dynamic: environment change while agent is deliberating.
- Discrete vs. continuous: state of the environment, time, percepts, actions.
- Known vs. unknown: "laws of physics" of the environment
- Hardest: partially observable, multiagent, stochastic, sequential, dynamic, continuous, unknown.
- Code repository includes environment simulator that places one or more agents in a simulated environment, observes their behavior over time, evaluates them according to a given performance measure.
  - Shit: this is something we could implement in Scheme (java, python, lisp, data); lot of work, though? Glory?
    * A lot of the utilities are in SRFI-1; e.g. `transpose` is `zip`.
    * Infinity is there.
    * Might have to write `rms-error`; `ms-error`.
    * `sample-with-replacement`; `sample-without-replacement`
      · Combinatorics SRFI, anyone?
    * `fuzz`
    * `print-grid`, &c.
    * `the-biggest`, `the-biggest-random-tie`, `the-biggest-that`, &c.
    * Binary tree stuff
    * Queue
    * Heap
    * They did CLOS-like stuff
  - Damn, they put some work in; could do it incrementally? Will be ad-hoc, I guarantee it.
  - Maybe we can program it, given the agents API.

    - `run-environment` looks like something central.
  - What would happen if we merely translated the code to Chicken? Could do so, perhaps, without fully understanding it; write an idiomatically Scheme-port later.
  - In that case, find some alternative to CLOS; or use tinyCLOS?
  - Also, beginning to think that we misnamed our repo: we're calling it `aima`, but we'd like to write an `aima` egg; with `aima-agents`, `aima-search`, `aima-logic`, `aima-planning`, `aima-uncertainty`, `aima-learning`, `aima-language` modules.
    - Call it `aima-egg`? `aima-chicken`?
  - Translation seems like the way to go: relatively mechanical.
  - Incidentally, "We need a new language for logical expressions, since we don't have all the nice characters (like upside-down A) that we would like to use." We can use in Scheme, can't we? Sure. Tough to type? Maybe. Also, think font-lock.
  - May not be up-to-date for 3e; let's see; also, rife with `defmethod` and other OOisms. Can ignore it, possibly, and its type-checking; `defstructure` is similar, I think, to SRFI-9.
  - Damn, they implemented unification.
  - Not to mention: the learning stuff (e.g. decision trees).
  - Man, should we implement this stuff ad-hoc; or otherwise depend on the existing implementations?
  - Path of least resistance: do it in Allegro? Ouch.
- The job of AI is to design an agent program that implements the agent function, the mapping from percepts to actions.
- Agent = architecture + program
- The agent program takes the current percept as input; the agent function, which takes the entire percept history.
- The agent function that the program embodies
- Write programs that produce rational behavior from a smallish program rather than a vast table.

- Reflex agents; model-based reflex agents; goal-based agents; utility-based agents.
- ```
  - table-driven-agent percept
    - persistent
      - percepts ()
      - table
    - append percept to percepts
    - lookup percepts table
  ```
- ```
  - reflex-vacuum-agent location status
    - if dirty? status
      - suck
      - else if location = A
        - right
      - else
        - left
  ```
- Simple reflex agent: select actions on the basis of the current precept
  - Learned responses, innate reflexes
  - ```
    - simple-reflex-agent percept
      - persistent
        - rules
      - state = interpret-input(percept)
      - rule = rule-match(state rules)
      - rule.action
    ```
  - Works only if the correct decision can be made on the current percept: i.e. if the environment is fully observable.
  - Escape from infinite loops is possible if the agent can randomize its actions.
  - In single-agent environments, randomization is usually not rational.
  - In most cases we can do better with more sophisticated deterministic agents.
- Model-based reflex agents
  - Partial observability: keep track of the part of the world it can't see now.
    ```

  - Internal state
  - Knowledge of how world evolves independently from agent
  - Knowledge of actions affect the world
  - Model of the world
  - ```
    - model-based-reflex-agent percept
      - persistent
        - state
        - model
        - rules
        - action
      - state = update-state(state action percept model)
      - rule = rule-match(state)
      - (action rule)
    ```
  - State of the world can contain goals.
- Goal-based agents
  - In addition to current state, goal information that describes situations that are desirable
  - Search, planning
  - Flexible, reason about world vis vis goals
- Utility-based agents
  - Whereas goals are happy/unhappy, more general performance measure: utility
  - Utility functional: internalization of performance measure
  - This is not the only way to be rational: rational agent for vacuum has no idea what its utility function is (see exercise 1.3).
  - When there are conflicting goals, utility function opts for the appropriate tradeoff.
  - Partial observability, stochasticity: decision making under uncertainty.
  - Expected utility of the action outcomes: derive, given the probabilities and utilities of each outcome.
    ```

  - Any rational agent must behave as if it possesses a utility function.
  - An agent that possesses an explicit utility function can make rational decisions with a general-purpose algorithm that does not depend on the specific utility function being maximized.
  - Global rationality: local constraint on rational-agent designs.
  - Choosing utility-maximizing course of action

- Learning agents
  - Now we're getting into some Mitchell-action: critic, learning element, performance element, problem generator, &c.
    - Need a book on big data?
  - Operate in initially unknown environments and become more competent.
  - Learning element
    - Making improvements
  - Performance element
    - Selecting external actions
  - Critic
    - How performance element should be modified vis vis fixed performance standard.
    - Performance standard must be fixed (i.e. checkmate).
    - Performance standard outside agent; conforms thereto.
  - Problem generator
    - Suggesting actions that will lead to new and informative experiences.
    - Suboptimal actions short run, better actions long run.
  - Utility-based agents learning utility information
  - Performance standard distinguishes percept as reward or penalty.
  - Process of modification of each component

- Atomic, factored, structured environments
  - Atomic
    - Each state of the world indivisible
  - Factored
    - Variables, attributes
  - Structured
    - Objects and relationships can be described explicitly
  - Increasing expressiveness
  - Intelligent systems: operate at all points along the expressiveness-axis simultaneously.
- Agents perceives and acts; agent function maps percept seq -¿ action.
- Performance measure evaluates behavior.
- Maximize expected performance measure.
- Task environment: performance measure, external environment, actuators, sensors.
- Nicomachean Ethics
- McCarthy, Programs with Common Sense
- Newell and Simon, Human Problem Solving
- Horvitz suggests the use of rationality conceived as the maximization of expected utility as the basis for AI. Pearl, 1988.[8]
  - Horvitz, E., 1988. Reasoning Under Varying and Uncertain Resource Constraints, Proc. of the 7th National Conference on AI, Minneapolis, MN, Morgan Kauffman, pp:111-116.
  - Horvitz, E. J., Breese, J.S., Henrion, M. 1988. Decision The- ory in Expert Systems and Artificial Intelligence, Journal of Approximate Reasoning, 2, pp247-302.

[8] Which of these following papers do you think he's talking about? Probably the latter: it carries an *et al.*

34.3 3

- If environment is unknown, agent has no choice but to try actionsat random (see exercise 2.11).
- Atomic representation: states of world, wholes: no internalstructure visible. (What the fuck does this mean?)
- Informed search algorithms
- Goal vs. problem formulation
- Search, solution, execution
- Open-loop: executed while ignoring its precepts: breaks loop between agent and environment.
  - http://en.m.wikipedia.org/wiki/Open-loop_controller:

    > Computes its input using only its model and the current state.
- (Wow: formulates plan in one pass and executes.)
- Initial state :: `in(arad)`
- Actions applicable in state s :: `in(arad) -> { go(Sibiu), &c. }`
- Transition model :: `result(s, a) -> successor; result(in(arad), go(zerind)) -> in(zerind)`
- State space :: initial state, actions, transitions; graph; path
- Goal test :: given state = goal state; enumerable or abstract
- Path cost :: step cost: `c(s, a, s')`
- Solution: leads from initial to goal state. Optimal solution minimizes cost.
- Abstraction: removing detail from representation. (Strange that it's formulated negatively.)
- Valid: expanded to more detailed environment; useful: easier than original problem.
- Toy problems: sliding block, n-queens
- N-queens: no path cost; incremental vs. complete-state

- Knuth conjecture: factorial, square root and floor reach any desired positive integer.
- Protein design
- Search tree: branches are actions, nodes: states
- Leaf nodes, frontier
- Expanding nodes on frontier, until solution (or not)
- State space: finite; search tree: infinite (loopy)
- Frontier separates the state-space graph into the explored region and the unexplored region.
- Node: state, parent, action (whence), path-cost (cumulative)
  - (They're getting implementationy.)
  - (Seem to be talking about breadth-first search here: there's a queue.)
- `solution` function: following parents up to root

```
- child-node(problem, parent, action) -> node
  - let
    - state (problem-result (node-state parent) action)
    - parent parent
    - action action
    - path-cose (+ (path-cost parent) (step-cost problem (node-state
      parent) action)
```

- (Oh, never mind: queues are supersets of stacks, apparently; LIFOs queues being the latter. Also, priority queues.)
- Canonical form: bit-vector, sorted list (keys for hash-tables)
- Completeness, optimality, time complexity, space complexity
- In AI, graph represented implicitly by initial state, actions, transition model; frequently infinite.
- Complexity:

  b branching factor (maximum successors)

  d depth

m maximum path in state space

- (State space: finally.)
- Search cost
- Total cost: search cost + path cost
- Uninformed search (blind search)
- Order in which nodes are expanded: informed search, heuristic search
- Breadth-first: root and successors expanded first
  - FIFO queue
  - Goal-test applied to each node when it is generated
  - Breadth first expands the shallowest nodes
- Uniform-cost search: expands n with lowest path-cost
  - (Greedy algorithm? Apparently not, since the goal-test is applied at expansion.)
  - Frontier: priority queue ordered by g
  - Goal-test applied when node selected for expansion; goal node that is generated may be on a suboptimal path
  - Second test added, in case a better path is found to a node currently on the frontier. (Where is it in the algorithm?)
  - Properties of priority queue and hash-table
  - (There are these negative proofs.)
  - Completeness is guaranteed, provided the cost of every step exceeds some small positive constant ϵ.
    - (ϵ and floats)
  - C^*: optimal value for C.
  - Uniform-cost does more work than breadth-first, expanding nodes even after finding a solution.
- Depth-first search
  - Expands deepest node in current frontier.

  - LIFO
  - In infinite state spaces, fails if infinite non-goal path encountered.
  - Depth-first search not optimal.
    * (Really? Interesting; depends, of course, on the order of nodes selected.)
  - m itself can be much larger than d; infinite, if tree unbounded.
  - Space complexity: for a graph search, no advantage; for a tree search: need store only a single path from root to leaf (plus unexplored siblings).
  - Depth first: constraint satisfaction, propositional satisfiability, logic programming
    * (Remember, though, that kanren had some breadth-first facilities.)
  - Backtracking search: modifying current state description
- Depth-limited search
  - Time complexity $O(b^l)$; space complexity $O(bl)$
  - Two kinds of failure: search failure, cutoff failure.
  - Diameter of state space: any node can reach any other node in at most D steps
- Iterative deepening (depth-first) search
  - Gradually increases the depth limit
  - Like depth-first: $O(bd)$ space complexity
  - Like breadth-first: complete when branching factor is finite and optimal when the path cost is non-decreasing

```
- iterative-deepening-search(problem) -> solution or failure
  - for depth = 0 to infinity do
    - result = depth-limited-search(problem, depth)
    - if result != cutoff
      - return result
```

  - Iterative deepening, preferred uninformed search method when search space is large and depth unknown

  - Do breadth-first until almost all memory consumed; iterative deepening on all nodes in the frontier.
  - Iterative analog to uniform-cost search? (Overhead, apparently.)

- Bidirectional search
  - Forward from the initial state and backwards from the goal: $b^{d/2} + b^{d/2} << b^d$
  - Replace goal-test to see whether frontiers intersect
  - Check when node selected for expansion; with hash-table: constant time
    - (Hash-tables for membership testing: I like it. Hence: canonical form, &c.)
  - Time, space complexity: $O(b^{d/2})$
  - Reduce by half if one of the two searches done by iterative deepening; at least one of the frontiers must be kept in memory
  - Requires mechanism for computing predecessors; in reversible case: reverse successor. Non-reversible: ingenuity
  - Explicitly listed goal-states: construct dummy goal-states whose predecessors are actual goal states.
    - (Why?)
  - Informed search
    - Problem-specific knowledge
    - Best-first search
      - Evaluation function, $f(n)$; cost estimate
      - Uniform-cost search, except that f instead of g orders the priority queue
      - A component of $f(n)$, the heuristic function $h(n)$: estimated cost of cheapest path from the state at node n to a goal state.
      - $h(n)$, unlike $g(n)$, depends only on the *state* at that node.
      - If n is a goal node, $h(n) = 0$.

* Greedy best-first search
    - $f(n) = h(n)$
    - E.g. straight-line distance heuristic
    - The amount of reduction depends on the particualr problem and on the quality of the heuristic.
* A^* search (minimizing total estimated cost)
    - $f(n) = h(n) + g(n)$ = estimated cost of the cheapest solution through n.
    - Provided that the heuristic function ($h(n)$) satisfies certain conditions, A^* is complete and optimal.
    - Identical to uniform-cost search, except that A^* uses $g+h$ instead of g.
    - Admissible heuristic :: Never overestimates the cost to reach the goal.
    - Consistent heuristic :: $h(n) \leq c(n, a, n') + h(n')$; triangle inequality: if there were a route from n to G_n via n' that was cheaper than $h(n)$, that would violate the property that $h(n)$ is a lower bound on the cost to reach G_n.
    - Every consistent heuristic, admissible.
    - h_{SLD}: consistent heuristic; straight line between n and n' no greater than $c(n, a, n')$.
    - The tree-search version of A^* is optimal if $h(n)$ is admissible, while the graph-search version is optimal if $h(n)$ is consistent.
    - If $h(n)$ is consistent, values of $f(n)$ along any path are non-decreasing.
    - $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$
    - There would have to be another frontier node n′ on the optimal path, by the graph selection property of 3.9.
    - See page 77: The frontier separates the state-space graph into the explored region and the unexplored region, so

that every path from the inital state to an unexplored state has to pass through a state in the frontier. (This is the graph separation property.)

· With an admissible but inconsistent heuristic, A^* requires some extra book-keeping to ensure optimality.

· It follows that the sequence of nodes expanded by A^* is in non-decreasing order of $f(n)$. Hence, the first goal node selected for expansion must be an optimal solutian because f is the true cost for goal nodes (which have $h = 0$) and all later goal nodes will be at least as expensive.

· The fact that f-costs are non-decreasing means we can draw contours in the state space.

· A^* fans out from the start node, adding nodes in concertric bands of increasing f-cost.

· With uniform-cost search (A^* using $h(n) = 0$), the bands will be circular; more accurate heuristics: bands will stretch toward the goal state and become more narrowlsy focused around the optimal path.

· A^* expands all nodes with $f(n) < C^*$.

· A^* might expand some nodes on the goal-contour before selecting the goal node.

· Completeness requires that there be only finitely many nodes with cost less than or equal to C^*, a condition that is true if all step costs exceed some finite ϵ and if b is finite.

· (Actually (i.e. accidentally) finite, or finite in principle? The latter doesn't matter; but we can put upper bounds on b in the case of e.g. tries.

· Subtree below Timisoara is pruned: because h_{SLD} is admissible.

· A^* is optimally efficient for any given consistent heuristic: no other optimal algorithm is guaranteed to expand fewer nodes than A^* (except possibly tie-breaking). Any

algorithm that does not expand all nodes with $f(n) < C$ runs the risk of missing the optimal solution.

- · The catch is that the number of states within the goal contour search space is still exponential.
- · Absolute error, relative error
- · Simplest model: state space, single goal, tree with reversible actions
- · A^* usually runs out of space before it runs out of time

* Memory-bounded heuristic search
  - · Iterative-deepening A^* (IDA*): cutoff is the f-cost $(g + h)$ rather than the depth. Practical for unit step costs; avoid the overhead of sorted queue of nodes.
  - · Recursive best-first search: using linear space
  - · Modifies currnt f; unwinds along the recursion
  - · Somewhat more efficient than IDA*, suffers from excessive node regeneration
  - · Space complexity: linear; time-complexity: depends on heuristic and how often best path changes.
  - · Memory-bounded A*
  - · Drop oldest worst leaf when memory full
  - · SMA* is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph.
* Metalevel state-space
  - · Goal of learning is to minimize total cost of problem solving.

- 8-puzzle: exhaustive tree search: 3^{22}; graph search: 181,440 distinct states.
- Manhattan distance
- Quality of heuristic: effective branching factor b^*. (If the total nuber of nodes generated by A^* for a particualr problem is N and the solution

depth is d, then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain $N + 1$ nodes.)

- Experimental measurements of b^* on a small set of problems can provide a good uide to the heuristic's overall usefulness
- A well-designed heuristic would have a value of b^* close to 1.
- "h_2 dominates h_1."
- Problem with fewer restrictions on the actions is called a relaxed problem: whence heuristic. The state-space of the relaxed problem is a supergraph of the original state space.
- The relaxed problem may have better solutions of the added edges provide short cuts.
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.
- It must obey the triangle inequality and is therefore consistent.
- Relaxed problems solved without search, otherwise: too expensive.
- `Absolver`: Prieditis
- $h(n) = max h_1(n), ..., h_m(n)$
- Composite heuristic: h is admissible; consistent, dominates component heuristics.
- Pattern databases
- Solutions share moves, cannot be additively combined
- Disjoint pattern databases
- Nonadditive heuristic
- Inductive methods supplied with features of a state that are relevant to predicting the state's value.
- Goal identified, well-defined problem formulated.
- Initial state, actions, transition model, goal test, path cost. This is the state space. Path through state space to goal state: solution.
- States and actions: atomic; don't consider internal structure. (What the fuck does this mean: features?)

- Tree-search: all paths; graph-search: avoids redundant paths.
- Completeness, optimality, time complexity, space complexity.
- Uninformed search
  - Breadth-first
  - Uniform-cost
  - Depth-first
  - Depth-limited
  - Iterative deepening
  - Bidirectional
- Informed search
  - Heuristic function
  - Best-first search
  - Greedy best-first search
  - A^* search
  - RBFS
  - SMA*
- Dynamic programming can be seen as a form of depth-first search on graphs.

34.4 Lectures

34.4.1 1

- AI: mapping from sensors to actuators
  - Voice, child-like engagement
- Fully vs. partially observable
- Deterministic vs. stochastic
- Discrete vs. continuous

- Benign vs. adversarial
- Uncertainty management

34.4.2 2

- Initial state
- `actions`$(state) \to a_1, a_2, a_3, \ldots$
- `result`$(state, action) \to state'$
- `goal-test`$(state) \to T|F$
- `path-cost`$(state \xrightarrow{action} state \xrightarrow{action} state) \to n$
- `step-cost`$(state, action, state') \to n$
- Navigate the state space by applying actions
- Separate state into three parts: ends of paths (frontier); explored and unexplored regions.
- Step-cost
- Tree-search
  - Family-resemlance; difference: which path to look at first.
- Depth-first search: shortest-first search

34.5 Turing, Computing Machinery and Intelligence

- Can machines think?
- It is A's object in the game to try and cause C to make the wrong identification.
  - Didn't realize there was an adversarial element to the Turing test.
- What will happen when a machine takes the part of A in this game?
- . . . drawing a fairly sharp line between the physical and the intellectual capacities of man.
  - A reasonable dualism

- May not machines carry out something which ought to be described as thinking but which is very different from what a man does?
  - The humanity/rationality plane of AI?
- Imitation game
  - Simulacrum sufficeth
- It is probably possible to rear a complete individual from a single cell of the skin (say) of a man . . . but we would not be inclined to regard it as a case of "constructing a thinking machine".
- Digital computer:
  1. Store
  2. Executive unit
  3. Control
- It is not normally possible to determine from observing a machine whether it has a random element, for a similar effect can be produced by such devices as making the choices depend on the digits of the decimal for π.
- Discrete state machines: strictly speaking there are no such machines. Everything really moves continuously.
- This is reminiscent of Laplace's view that from the complete state of the universe at one moment of time, as described by the positions and velocities of all particles, it should be possible to predict all future states.
- This special property of digital computers, that they can mimic any discrete state machine, is described by saying that they are universal machines.
- "Are there imaginable digital computers which would do well in the imitation game?" $\rightarrow$ "Are there discrete state machines which would do well?"
- I believe that in about fifty years' time it will be possible to programme computers, with a storage capacity of about 10^9, to make them play the imitation game so well that an average interrogator will not have more than 70% chance of making the right identification after five minutes of questioning.

  - Russell/Norvig, 12: storage units: 10^15
  - Loebner prize:

    > Elbot of Artificial Solutions won the 2008 Loebner Prize bronze award, for most human-like artificial conversational entity, through fooling three of the twelve judges who interrogated it (in the human-parallel comparisons) into believing it was human. This is coming very close to the 30% traditionally required to consider that a program has actually passed the Turing test.

    * From a judge:

      > He predicted that by the end of the century, computers would have a 30 per cent chance of being mistaken for a human being in five minutes of text-based conversation.

      I thought this was mistaken (should be 70), but it is indeed correct.
  - In other words, a damn-good guess.

- Conjectures are of great importance since they suggest useful lines of research.
- We might expect that He would only exercise this power in conjunction with a mutation which provided the elephant with an appropriately improved brain to minister to the needs of this soul.
- We like to believe that Man is in some subtle way superior to the rest of creation.
- "The consequences of machines thinking would be too dreadful." I do not think that this argument sufficiently substantial to require refutation. Consolation would would be more appropriate: perhaps this should be sought the transmigration of souls.
- There are limitations to the powers of discrete-state machines. The best known of these results is known as Gdel's theorem, and shows that in any sufficiently powerful logical system statements can be formulated which can neither be proved nor disproved within the system, unless possibly the system itself is inconsistent.

- "Will this machine every answer 'Yes' to any question?" It can be shown that the answer is either wrong or not forthcoming.
- The only way to know that a man thinks is to be that particular man. It is in fact the solipsist point of view.
- I do not wish to give the impression that I think there is no mystery about consciousness. There is, for instance, something of a paradox connected with any attempt to localise it.
- When a burnt child fears the fire and shows that he fears it by avoiding it, I should say that he was applying scientific induction.
- It would deliberately introduce mistakes in a manner calculated to confuse the interrogator.
- By observing the results of its own behaviour it can modify its own programmes so as to achieve some purpose more effectively.
- This is the assumption that as soon as a fact is presented to a mind all consequences of that fact spring into the mind simultaneously with it.
- The undistributed middle is glaring.
- I would defy anyone to learn from these replies sufficint about the programme to be able to predict any replies to untried values.
- A smallish proportion are super-critical. An idea presented to such a mind may give rise to a whole "theory" consisting of secondary, tertiary and more remote ideas.
    - Sponteneity
- These last two paragraphs should be described as "recitations tending to produce belief."
- The only satisfactory support that can be given will be that provided by waiting for the end of the century and then doing the experiment described.
- Estimates for the storage capacity of the brain vary from 10^10 to 10^15 binary digits.
    - Russell/Norvig (12): 10^13 synapses

- At my present rate of working I produce about a thousand digits of programme a day, so that about sixty workers, working steadily through the fifty years might accomplish the job, if nothing went into the waste-paper basket.
  - Mythical man-month?
- The child-programme and the education process
- One might have a complete system of logical inference "built in". The store would be largely occupied with definitions and propositions. Certain propositions may be described as "imperatives". As soon as an imperative is classed as "well-established" the appropriate action takes place.
  - Compare McCarthy, Programs with Common Sense, regarding imperatives.
- These choices make the difference between a brilliant and a footling reasoner.
- We can only see a short distance ahead, but we can see plenty there that needs to be done.

35 TODOs

35.1 TODO Focus on one or two problems for the coming week.

Assign people to be responsible for them.

35.2 TODO Modify 2.11 to list available actions?

No: that changes the problem.

35.3 TODO A listing environment

35.4 TODO Blog about barabsi-albert vs. depth-first graph-creation.

The former produces nice hubs and cycles, but the problem of aligning complementary directions (like graph-coloring?) is NP-complete. Resorting to depth-first with no cycles; would have to have an exponential adjunct for cycles.

Way to quantify the difference between the two with respect to e.g. average degree of nodes?

35.5 DONE 2.11

`CLOSED:` *2012-07-16 Mon 12:57*

35.5.1 DONE Graph world

`CLOSED:` *2012-07-16 Mon 12:44*

```
(use aima-vacuum
     debug
     srfi-25)
```

If we get too fancy, it's going to take us a while; why not just draw a couple random barriers in a box? Not enough to probably block access, &c.

The world should be a graph; we'll search on that graph and thereby anticipate chapter 3. Debug output can produce an animated graph over graphviz (see e.g. the Iraq project). Show how the agent moves and the squares clean?

If we want to create a random graph, but not necessarily a maze that fills up a given extent, we can specify e.g. the number of nodes. We have to have some condition for backtracking, though: random? The fill-up-an-extent maze-algorithm backtracks when it has no unseen neighbors; I suppose we can weight backtrack vs. branch differently and see what happens. I wonder

whether e.g. 50/50 will produce a graph with a lot of shallow paths and high branching. Make backtracking and branching relatively expensive, then.

The random reflex agent might do well in a completely connected graph, incidentally, since it doesn't have to make any decisions about where to go; it might fair poorly in a linear environment, since at least half of the time it will be unable to move.

It turns out that random graphs are an open topic of research; with the ErdsRnyi, Watts-Strogatz, BarabsiAlbert models. Let's go with BarabsiAlbert: it looks relatively easy to implement and has interesting clustering properties (power-law degree-distributions).

Confer Gabor Csardi's implementation of Barabsi-Albert, which has zero-appeal and power-of-preferential-attachment parameters; it also begins with one node. It can dispense with the two nodes from WP because of the zero-appeal parameter. It also does directed and undirected; and distinguishes, somehow, between in- and both in-and-out-degrees. It also does not divide by the total number of nodes (or is it the total number of edges?).

(Holy shit: this chick implements all three in Perl; she claims that you divide by all the degrees in the graph, which is consistent with WP's notation. With Gabor Csardi, she also has a parameter m of edges to add at each time; her graph also starts with no edges, but has m_0 nodes at t_0; they both discuss the problem of multiple edges: it may not suffice to merely not generate them, you should detect and ignore them. No, wait; she's talking about something different: "To keep the code simple, I don't test whether the m nodes that are connecting to the new node are all different." Gabor, on other hand, talks about an algorithm which "never generates multiple edges." Wait again: she's talking about the same thing. Should we ignore duplicate links and add another; or just refrain from duplicating? Damn.)

It sounds like we can create a connected-graph if we start with the seed-graph-of-degree-one hack; using zero-appeal, we might have disconnected but dirty squares. Disconnected but dirty squares are romantic, however; and their effect should fall prey to the law of large numbers if we repeat it enough times.

We are, of course, constrained by the relative directions up, down, left and right; the degree of any node is therefore ≤ 4. In that sense, maybe the matrix is a reasonable representation? Barabsi-Albert isn't going to be able to create hubs with that sort of upper bound. Push forward anyway, or capitulate to grid?

I like the idea of a graph, each of whose nodes contains $0 \leq neighbors \leq 4$; which are assigned the arbitrary designations `up`, `down`, `left` and `right`. The only problem is that, for experimentation, we're going to have to generate a complete grid-like world. That is a pain in the ass.

I take that back: can we create something like a torus or a sphere where the grid wraps around; that is, where every square is totally connected (edges being connected to the opposite edge)? A random reflex-vacuum might be able to do well there.

If with modified Barabsi-Albert we're going to have disconnected nodes, by the way, if e.g. p_i is false for every node; or are we guaranteed to connect m nodes every time? See e.g.:

> The generalized creation process of a scale-free-network is defined by Barabasi and Albert [BaBo03]: Start with a small number of nodes $m0$. During each step, insert a new node with connections to $m \leq m0$ existing nodes. The probability P_v of node v being connected to a new node depends on its degree j_v. The higher its degree, the greater the probability that it will receive new connections ("the rich get richer").

Since there's an upper bound on the dividend, we're going to degenerate into a network whose edges are picked at random; and, worse yet, might end up with many unconnected nodes.

Can't we just create the world initially, by the way, instead of creating the graph and then the world? Interestingly, the relative directions won't meet up: `up` is paired with `left`, &c. Is such a world physically possible? Yes, if the relative directions for each location are different; they're relative, after all. It shouldn't matter to the agents.

It's a weird world, though.

I want to have an environment that spits out an animated graph of where the agent is (green), which cells are dirty (grey), which are clean (white).

```
(use aima
     aima-vacuum
     debug
     extras
     files
     lolevel
     posix
```

```
     random-bsd
     srfi-1
     srfi-69)

(define (connect! graph connectend connector)
  (hash-table-update!/default
   graph
   connectend
   (lambda (neighbors)
     (lset-adjoin eq? neighbors connector))
   '()))

(define (biconnect! graph connectend connector)
  (connect! graph connectend connector)
  (connect! graph connector connectend))

(define make-node gensym)

(define (sum-degrees graph)
  (hash-table-fold graph
                   (lambda (from to sum) (+ (length to) sum))
                   0))

(define (write-graph-as-dot graph)
  (let ((graph (hash-table-copy graph)))
    (display "digraph G {")
    (for-each (lambda (from) (format #t "~a;" from))
      (hash-table-keys graph))
    (hash-table-walk
     graph
     (lambda (from to)
       (for-each
           (lambda (to) (format #t "~a->~a;" from to)) to)))
    (display "}\n")))

(define (write-graph-as-pdf graph)
  (receive (input output id)
    (process "neato -Tpdf")
    (with-output-to-port output
      (lambda () (write-graph-as-dot graph)))
```

```
    (close-output-port output)
    (with-input-from-port input
      (lambda ()
        (do ((line (read-line) (read-line)))
            ((eof-object? line))
          (write-line line))))
    (close-input-port input)))

(define (display-pdf pdf)
  (system* "evince -s ~a" pdf))

;;; Number of nodes to try and join to the seed.
(define default-n-nodes (make-parameter 1000))

;;; Number of edges to attempt to create per step.
(define default-m (make-parameter 4))

;;; We're going to have to maintain the e.g. clean-dirty data in
;;; separate structures if the graph merely expresses node
;;; relationships.
;;;
;;; That's fine: let's maintain a second hash-table for `node ->
;;; clean?'; so-called obstacles are merely lack of passage.
;;;
;;; We need a mapping from nodes to up, down, left, right; however.
;;; Can we assign this randomly, somehow? Otherwise, we'll get an
;;; up-biased graph; or, rather: an up-, down-, left-, right-biased
;;; graph (in that order).
;;;
;;; Yet another hash-table that assigns direction?
;;;
;;; No: one hash table that expresses the graph structure, another
;;; that maps symbols to objects; which objects contain: status,
;;; relative direction. Problem is: the same node can be linked to
;;; more than once from different relative directions (the graph has
;;; loops; doesn't it? Yes it does: but they're relatively rare. In
;;; this sense: Barabsi-Albert graphs are probably different from
;;; grids with sparse obstactles).
;;;
;;; Fuck it: let's have an up-biased graph; maybe the reflex agent can
```

```
;;; use that knowledge to its advantage.
;;;
;;; Should we allow an optional seed here, so we can reproduce the
;;; input?
(define barabsi-albert-graph
  (case-lambda
   (() (barabsi-albert-graph (default-n-nodes) (default-m)))
   ((n-nodes m)
    (let ((graph (make-hash-table)))
      (biconnect! graph (make-node) (make-node))
      (do ((n-nodes n-nodes (- n-nodes 1)))
          ((zero? n-nodes))
        (let ((new-node (make-node)))
          (do ((from (hash-table-keys graph) (cdr from))
               (m m (- m 1)))
              ((or (null? from) (zero? m)))
            (let ((sum-degrees (sum-degrees graph)))
              (let* ((from (car from))
                     (degrees-from (length (hash-table-ref graph from))))
                (if (and (< degrees-from 4)
                         (< (random-real) (/ degrees-from sum-degrees)))
                    (biconnect! graph from new-node)))))))
      graph))))

;; (with-output-to-file "graph.pdf"
;;   (lambda () (write-graph-as-pdf (barabsi-albert-graph))))

;; (display-pdf "graph.pdf")

(define-record no-passage)
(define no-passage (make-no-passage))
(define passage? (complement no-passage?))

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

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

(define-record location
```

```
  status
  left
  right
  up
  down)

(define-record-printer location
  (lambda (location output)
    (display (record->vector location) output)))

(define default-width (make-parameter 1600))

(define default-height (make-parameter 900))

;;; Height and width are in pixels.
(define write-dot-preamble
  (case-lambda
   ((agent step)
    (write-dot-preamble agent step (default-height) (default-width)))
   ((agent step height width)
    (display "digraph G {")
    (display "node [style=filled, fontname=monospace];")
    (display "edge [fontname=monospace];")
    (if (and height width)
        (begin
          (display "graph [fontsize=48.0, ratio=fill];")
          ;; Phew: viewports are specified in points at 72 per inch;
          ;; size is specified in pixels at 96 per inch.
          (let ((width-in-inches (/ width 96))
                (height-in-inches (/ height 96)))
            (format #t "graph [viewport=\"~a,~a\", size=\"~a,~a!\"];"
                    (* width-in-inches 72)
                    (* height-in-inches 72)
                    width-in-inches
                    height-in-inches))))
    (if step
        (format #t "graph [label=\"Score: ~a; step: ~a\"]"
                (agent-score agent)
                step)))))
```

```
(define (write-dot-nodes world agent)
  (hash-table-walk
   world
   (lambda (name location)
     (let ((color
            (cond ((eq? (agent-location agent) name) "green")
                  ((clean? (location-status location)) "white")
                  (else "gray"))))
       (format #t "~a [fillcolor=~a];" name color)))))

(define (write-dot-edges world)
  (hash-table-walk
   world
   (lambda (name location)
     (let ((left (location-left location))
           (right (location-right location))
           (up (location-up location))
           (down (location-down location)))
       (if (passage? left)
           (format #t "~a->~a [label=l];" name left))
       (if (passage? right)
           (format #t "~a->~a [label=r];" name right))
       (if (passage? up)
           (format #t "~a->~a [label=u];" name up))
       (if (passage? down)
           (format #t "~a->~a [label=d];" name down))))))

(define (write-dot-postscript) (display "}\n"))

(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-height) (default-width)))
   ((world agent step height width)
    (write-dot-preamble agent step height width)
    (write-dot-nodes world agent)
    (write-dot-edges world)
    (write-dot-postscript))))
```

```
(define (write-world-as-pdf world agent pdf)
  (receive (input output id)
    (process "neato" `("-Tpdf" "-o" ,pdf))
    (with-output-to-port output
      ;; Do we really need a blank label, for some reason?
      (lambda () (write-world-as-dot world agent #f #f #f)))
    (flush-output output)
    (close-output-port output)
    (close-input-port input)))

(define (write-world-as-gif world agent frame gif)
  (receive (input output id)
    (process "neato" `("-Tgif" "-o" ,gif))
    (with-output-to-port output
      (lambda () (write-world-as-dot world agent frame)))
    (flush-output output)
    (close-output-port output)
    (close-input-port input)))

(define (graph->world graph)
  (let ((world (make-hash-table)))
    (hash-table-walk
     graph
     (lambda (from to)
       (let* ((rest (make-list (- 4 (length to)) no-passage))
              (to (append to rest))
              (to (shuffle to (lambda (x) (random (length to))))))
         (let ((location
                (make-location
                 dirty
                 (list-ref to left)
                 (list-ref to right)
                 (list-ref to up)
                 (list-ref to down))))
           (hash-table-set! world from location)))))
    world))

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

```
(define (make-randomized-reflex-graph-agent start)
  (make-reflex-agent
   start
   (lambda (location clean?)
     (if clean?
         (list-ref '(left right up down) (random 4))
         'suck))))

(define (make-graph-environment world agent)
  (lambda ()
    (let* ((location (agent-location agent))
           (action ((agent-program agent)
                    location
                    (clean? (location-status
                             (hash-table-ref world location))))))
      (debug-print "agent-action" action)
      (case action
        ((left)
         (let ((left (location-left (hash-table-ref world location))))
           (if (passage? left)
               (agent-location-set! agent left))))
        ((right)
         (let ((right (location-right (hash-table-ref world location))))
           (if (passage? right)
               (agent-location-set! agent right))))
        ((up)
         (let ((up (location-up (hash-table-ref world location))))
           (if (passage? up)
               (agent-location-set! agent up))))
        ((down)
         (let ((down (location-down (hash-table-ref world location))))
           (if (passage? down)
               (agent-location-set! agent down))))
        ((noop))
        ((suck)
         (location-status-set! (hash-table-ref world location) clean))
        (else (error "graph-environment -- Unknown action"))))))

(define (make-graph-world) (graph->world (barabsi-albert-graph)))
```

```
(define (make-graph-performance-measure world agent)
  (lambda ()
    (let ((clean-locations
           ;; Quicker with map and apply?
           (hash-table-fold
            world
            (lambda (name location clean-locations)
              (if (clean? (location-status location))
                  (+ clean-locations 1)
                  clean-locations))
            0)))
      (agent-score-set! agent (+ (agent-score agent) clean-locations)))))

(define (make-graph-animating-environment world agent directory)
  (let ((frame 0))
    (lambda ()
      (let ((gif (make-pathname directory (number->string frame) "gif")))
        (write-world-as-gif world agent frame gif))
      (set! frame (+ frame 1)))))

;;; I think make-step-limited-environment is a special case of
;;; make-finalizing-environment with noop.
;;;
;;; This probably belongs in aima; should we preserve the
;;; step-limited-environment as a specialization of this?
(define (make-finalizing-environment finalizer final-step)
  (let ((step 0))
    (lambda ()
      (set! step (+ step 1))
      (let ((continue? (< step final-step)))
        (if (not continue?) (finalizer))
        continue?))))

(define (make-animation-finalizer directory)
  (lambda ()
    (system "rm -fv graph.gif")
    (system* "convert $(find ~a -type f | sort -k 1.~a -n) -loop 0 graph.gif"
             directory
             (+ (string-length directory) 2))
```

```
    (system* "identify ~a/0.gif" directory)
    (system "mencoder graph.gif -ovc lavc -o graph.avi")))

(define (make-stateful-graph-agent start)
  (make-reflex-agent
   start
   (let ((world (make-hash-table)))
     (lambda (location clean?)
       'right))))

(let* ((world (make-graph-world))
       (start (random-start world))
       (agent (make-stateful-graph-agent start)))
  (let ((directory (create-temporary-directory))
        (steps 20))
    (write-world-as-pdf world agent "world.pdf")
    (display-pdf "world.pdf")
    #;
    (simulate
     (compose-environments
      (make-step-limited-environment 10)
      (make-graph-environment world agent)
      (make-debug-environment agent)
      (make-graph-performance-measure world agent)
      ;; Can't this contain its finalizer? Maybe even give it the
      ;; terminal frame?
      ;; (make-graph-animating-environment world agent directory)
      ;; (make-finalizing-environment
      ;;  (make-animation-finalizer directory)
      ;;  steps)
      ))))
```

This is a cool graph with a large ring, incidentally; I don't know what the random seed was, unfortunately:

```
digraph G {
  node [style=filled, fontname=monospace]
  edge[fontname=monospace]
  g17263 [fillcolor=white]
  g17270 [fillcolor=white]
  g17273 [fillcolor=white]
```

```
g17278 [fillcolor=white]
g17298 [fillcolor=white]
g17310 [fillcolor=gray]
g17316 [fillcolor=white]
g17329 [fillcolor=gray]
g17336 [fillcolor=gray]
g17357 [fillcolor=white]
g17226 [fillcolor=white]
g17227 [fillcolor=gray]
g17229 [fillcolor=white]
g17231 [fillcolor=gray]
g17232 [fillcolor=green]
g17234 [fillcolor=gray]
g17235 [fillcolor=gray]
g17237 [fillcolor=gray]
g17238 [fillcolor=white]
g17239 [fillcolor=gray]
g17245 [fillcolor=white]
g17258 [fillcolor=white]
g17259 [fillcolor=gray]
g17262 [fillcolor=gray]
g17263->g17258 [label=d]
g17270->g17262 [label=l]
g17273->g17245 [label=u]
g17278->g17245 [label=l]
g17298->g17262 [label=u]
g17310->g17258 [label=l]
g17310->g17262 [label=u]
g17316->g17259 [label=l]
g17329->g17258 [label=u]
g17336->g17259 [label=u]
g17357->g17259 [label=u]
g17226->g17229 [label=l]
g17226->g17227 [label=r]
g17227->g17226 [label=d]
g17229->g17226 [label=l]
g17229->g17234 [label=r]
g17229->g17232 [label=u]
g17229->g17231 [label=d]
g17231->g17229 [label=l]
```

```
  g17232->g17229 [label=l]
  g17232->g17238 [label=u]
  g17234->g17235 [label=l]
  g17234->g17229 [label=r]
  g17234->g17239 [label=u]
  g17234->g17237 [label=d]
  g17235->g17234 [label=u]
  g17237->g17245 [label=l]
  g17237->g17234 [label=r]
  g17237->g17239 [label=u]
  g17238->g17259 [label=l]
  g17238->g17258 [label=u]
  g17238->g17232 [label=d]
  g17239->g17234 [label=l]
  g17239->g17237 [label=r]
  g17245->g17278 [label=l]
  g17245->g17273 [label=r]
  g17245->g17262 [label=u]
  g17245->g17237 [label=d]
  g17258->g17310 [label=l]
  g17258->g17263 [label=r]
  g17258->g17329 [label=u]
  g17258->g17238 [label=d]
  g17259->g17357 [label=l]
  g17259->g17336 [label=r]
  g17259->g17316 [label=u]
  g17259->g17238 [label=d]
  g17262->g17245 [label=l]
  g17262->g17310 [label=r]
  g17262->g17298 [label=u]
  g17262->g17270 [label=d]
}
```

Converting the `gif` output to an `avi` la `mencoder`:

```
mencoder graph.gif -ovc lavc -o graph.avi
```

35.5.2 DONE Depth-first graph constructor

`CLOSED:` *2012-07-16 Mon 12:44*

Like Barabsi-Albert, the probability of branching is relative to the degree of the current node (plus some probability for degreeless nodes); otherwise, follow a random branch; otherwise, backtrack.

```
- make-seed-world
  - let
    - world make-hash-table
    - root make-node
    - child make-node
      - biconnect! world root child (random 4)
      - values world root
- # Let's recreate location-left{,-set!}, &c.
- make-preferential-depth-first-world n-nodes
  - receive world root
    - make-seed-world
    - let iter
      - node root
      - n-nodes n-nodes
      - n-degrees 0
        - if zero? n-nodes
          - world
          - hash-table-update
            - world
            - node
            - lambda location
              - # Wouldn't we make it easier on ourselves by having
                the children of location be a tetradic vector with
                left, right, up and down? Yes, a thousand times yes!
              - # If barabsi dictates add a child, add it; otherwise,
                take a random path. This random path could backtrack,
                or it could continue on.
              - # Keep adding until we have the requisite number of
                nodes.
              - let
                - degrees-from
                  - vector-fold
                  - lambda (i degrees passage) if (no-passage? passage)
                    degrees (+ degrees 1)
                  - 0
                  - location-passages location
```

```
                - if
                  - and
                    - < degrees-from 4
                    - < (random-real) (/ degrees-from n-degress)
                  - let
                    - directions-from
                      - vector-fold
                        - lambda (direction directions-from passage) if
                          (no-passage? passage) (cons direction
                          directions-from) directions-from
                        - '()
                        - location-passages location
                    - direction (list-ref directions-from (random
                      (length directions-from)))
                      - let new-node (make-node)
                        - # Maybe biconnect! can take care of adding the
                          node to the world?
                        - biconnect! world node new-node direction
                        - iter new-node (- n-nodes 1) (+ n-degrees 2)
                  - let
                    - passages-from
                      - vector-fold
                        - lambda (direction passages-from passage) if
                          (passage? passage) (cons passage
                          passages-from) passages-from
                        - '()
                        - location-passages location
                    - passage (list-ref
            - make-location
              - dirty
              - #(no-passage no-passage no-passage no-passage)
```

```
(use aima
     aima-vacuum
     debug
     files
     posix
     (prefix random-bsd bsd-)
     srfi-69
     stack
```

```
     vector-lib)

(define-record no-passage)
(define no-passage (make-no-passage))
(define passage? (complement no-passage?))

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

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

(define-record location
  status
  neighbors)

(define-record-printer location
  (lambda (location output)
    (display (record->vector location) output)))

(define (copy-location location)
  (make-location (location-status location)
                 (vector-copy (location-neighbors location))))

(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))

(define make-node gensym)

(define (random-direction) (bsd-random 4))

(define (reverse-direction direction)
```

```
  (cond ((left? direction) right)
        ((right? direction) left)
        ((up? direction) down)
        ((down? direction) up)))

(define (make-dirty-location)
  (make-location dirty
                 (vector no-passage
                         no-passage
                         no-passage
                         no-passage)))

(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)))

(define (make-seed-world)
  (let ((world (make-hash-table))
        (start (make-node))
        (neighbor (make-node)))
    (connect! world start neighbor (random-direction))
    world))

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

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

(define (count-nodes world)
  (length (hash-table-keys world)))

(define (count-degrees world)
  (hash-table-fold
   world
   (lambda (node location n-degrees)
     (+ n-degrees (vector-count
                   (lambda (direction neighbor)
                     (passage? neighbor))
                   (location-neighbors location))))
   0))

(define (n-neighbors location)
  (vector-fold
   (lambda (direction n-neighbors neighbor)
     (if (no-passage? neighbor)
         n-neighbors
         (+ n-neighbors 1)))
   0
   (location-neighbors location)))

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

;;; This, of course, won't produce any cycles.
(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)))))
                ;; To make this Barabsi-like, we could try to
                ;; pick a prexisting node; and, failing that,
                ;; produce one.
                ;;
                ;; Why not just produce a direction-sensitive
                ;; Barabsi? Now that we have neighbors as a
                ;; vector, it should be less unpleasant.
                ;;
                ;; To connect this node to a prexisting one,
                ;; however; we'd have to find nodes with
                ;; compatible, available directions.
                ;;
                ;; We could produce a tree, of course, and
                ;; randomly create appropriate cycles.
```

```
                        (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))))))))))

(define default-width (make-parameter 1600))

(define default-height (make-parameter 900))

(define default-font-size (make-parameter 48.0))

(define default-title (make-parameter #f))

;;; Height and width are in pixels.
(define write-dot-preamble
  (case-lambda
   ((agent step)
    (write-dot-preamble agent
                        step
                        (default-width)
                        (default-height)
                        (default-font-size)
                        (default-title)))
   ((agent step width height font-size title)
    (display "digraph G {")
    (display "node [style=filled, fontname=monospace];")
    (display "edge [fontname=monospace];")
    (if (and width height)
        (begin
```

```
          (format #t "graph [fontsize=~a, ratio=fill];" font-size)
          ;; Phew: viewports are specified in points at 72 per inch;
          ;; size is specified in pixels at 96 per inch.
          (let ((width-in-inches (/ width 96))
                (height-in-inches (/ height 96)))
            (format #t "graph [viewport=\"~a,~a\", size=\"~a,~a!\"];"
                    (* width-in-inches 72)
                    (* height-in-inches 72)
                    width-in-inches
                    height-in-inches))))
    (if step
        (format #t "graph [label=\"~aScore: ~a; step: ~a\"]"
                (if title (format "~a\\n" title) "")
                (agent-score agent)
                step)))))

(define (write-dot-nodes world agent)
  (hash-table-walk
   world
   (lambda (name location)
     (let ((color
            (cond ((eq? (agent-location agent) name) "green")
                  ((clean? (location-status location)) "white")
                  (else "gray"))))
       (format #t "~a [fillcolor=~a];" name color)))))

(define (write-dot-edges world)
  (hash-table-walk
   world
   (lambda (name location)
     (let ((left (vector-ref (location-neighbors location) left))
           (right (vector-ref (location-neighbors location) right))
           (up (vector-ref (location-neighbors location) up))
           (down (vector-ref (location-neighbors location) down)))
       (if (passage? left)
           (format #t "~a->~a [label=l];" name left))
       (if (passage? right)
           (format #t "~a->~a [label=r];" name right))
       (if (passage? up)
           (format #t "~a->~a [label=u];" name up))
```

```
        (if (passage? down)
            (format #t "~a->~a [label=d];" name down))))))

(define (write-dot-postscript) (display "}\n"))

(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))))

(define (write-world-as-pdf world agent pdf)
  (receive (input output id)
    (process "neato" `("-Tpdf" "-o" ,pdf))
    (with-output-to-port output
      ;; Do we really need a blank label, for some reason?
      (lambda () (write-world-as-dot world agent #f #f #f)))
    (flush-output output)
    (close-output-port output)
    (close-input-port input)))

(define (display-pdf pdf)
  (system* "evince -s ~a" pdf))

(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)))))

(define (make-graph-environment world agent)
  (lambda ()
    (let* ((node (agent-location agent))
           (location (hash-table-ref world node))
           (action ((agent-program agent)
                    node
                    (clean? (location-status location)))))
      (debug-print "agent-action" action)
      (case action
        ((left)
         (let ((left (vector-ref (location-neighbors location) left)))
           (if (passage? left)
               (agent-location-set! agent left))))
        ((right)
         (let ((right (vector-ref (location-neighbors location) right)))
           (if (passage? right)
               (agent-location-set! agent right))))
        ((up)
         (let ((up (vector-ref (location-neighbors location) up)))
```

```
            (if (passage? up)
                (agent-location-set! agent up))))
          ((down)
           (let ((down (vector-ref (location-neighbors location) down)))
             (if (passage? down)
                 (agent-location-set! agent down))))
          ((noop))
          ((suck)
           (location-status-set! (hash-table-ref world node) clean))
          (else (error "graph-environment -- Unknown action"))))))

(define (make-graph-performance-measure world agent)
  (lambda ()
    (let ((clean-locations
           ;; Quicker with map and apply?
           (hash-table-fold
            world
            (lambda (name location clean-locations)
              (if (clean? (location-status location))
                  (+ clean-locations 1)
                  clean-locations))
            0)))
      (agent-score-set! agent (+ (agent-score agent) clean-locations)))))

(define make-graph-animating-environment
  (case-lambda
   ((world agent directory)
    (make-graph-animating-environment world
                                      agent
                                      directory
                                      (default-width)
                                      (default-height)
                                      (default-font-size)
                                      (default-title)))
   ((world agent directory width height font-size title)
    (let ((frame 0))
      (lambda ()
        (let ((gif (make-pathname directory (number->string frame) "gif")))
          (write-world-as-gif world
                              agent
```

```
                                frame
                                gif
                                width
                                height
                                font-size
                                title))
          (set! frame (+ frame 1)))))))

;;; I think make-step-limited-environment is a special case of
;;; make-finalizing-environment with noop.
;;;
;;; This probably belongs in aima; should we preserve the
;;; step-limited-environment as a specialization of this?
(define (make-finalizing-environment finalizer final-step)
  (let ((step 0))
    (lambda ()
      (set! step (+ step 1))
      (let ((continue? (< step final-step)))
        (if (not continue?) (finalizer))
        continue?))))

(define (make-animation-finalizer directory file)
  (lambda ()
    (system* "rm -fv ~a.gif" file)
    (system* "convert $(find ~a -type f | sort -k 1.~a -n) -loop 0 ~a.gif"
             directory
             (+ (string-length directory) 2)
             file)
    (system* "identify ~a/0.gif" directory)
    (system* "mencoder ~a.gif -ovc lavc -o ~a.avi" file file)))

(define (make-composite-animation-finalizer combinandum combinator file)
  (let ((composite-directory (create-temporary-directory)))
    (system* "cd ~a && for i in *; do echo $i; convert +append $i ~a/$i ~a/$i; done"
             combinandum
             combinator
             composite-directory)
    (make-animation-finalizer composite-directory file)))

(define (make-unknown-location clean?)
```

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

(define (undiscovered-directions location)
  (vector-fold
   (lambda (direction undiscovered-directions neighbor)
     (if (unknown? neighbor)
         (cons direction undiscovered-directions)
         undiscovered-directions))
   '()
   (location-neighbors location)))

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

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

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

(define-record cycle)
(define cycle (make-cycle))

;;; Dealing with all this move-location punning makes things complex;
;;; we can clean this up a little bit by writing some germane
;;; abstractions on the world.
;;;
;;; We're not dealing with cycles yet, by the way; does this entail
;;; determining whether or not a new node is accounted for in the
;;; world? I believe so.
(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?))))
               ;; The following is general house-keeping on the state.
               (if (stack-empty? moves)
                   ;; We're dealing with an uninitialized agent: set
                   ;; the world. This could also be a terminal
                   ;; agent, couldn't it? Is there a better place to
                   ;; initialize?
                   (hash-table-set! world node location)
                   ;; We need to distinguish the case, apparently,
                   ;; where we've just backtracked; this isn't quite
                   ;; the same as a fail-to-move.
                   ;;
                   ;; In 2.12, when we're dealing with a bump
                   ;; sensor, when don't have to play these games
                   ;; with an implicit bump.
                   (let ((last-move (stack-peek moves)))
                     (if (eq? last-move 'backtrack)
                         ;; Our position is the result of
                         ;; backtracking; remove the special
                         ;; backtracking move.
                         (stack-pop! moves)
                         (if (eq? (stack-peek nodes) node)
                             ;; We tried to move but could not; mark the
                             ;; last direction as no-passage.
                             (let ((last-move (stack-pop! moves)))
                               (vector-set! (location-neighbors location)
```

```
                                (move->direction last-move)
                                no-passage))
                    (let* ((last-node (stack-peek nodes))
                           ;; Need to replace hash-table-ref, &c.
                           ;; with something more germane.
                           (last-location
                            (hash-table-ref world last-node)))
                      (if (hash-table-exists? world node)
                          ;; Cycle detected! Push the
                          ;; cycle-sentinel.
                          (stack-push! nodes cycle)
                          (begin
                            ;; This is a new node: add it
                            ;; to the world.
                            (hash-table-set! world node location)
                            ;; Also, add it to the list of
                            ;; interesting nodes.
                            (stack-push! nodes node)))
                      ;; This location's reverse-move points to
                      ;; the last node.
                      (vector-set! (location-neighbors location)
                                   (move->direction
                                    (reverse-move last-move))
                                   last-node)
                      ;; The last location's move points to
                      ;; this node.
                      (vector-set! (location-neighbors
                                    last-location)
                                   (move->direction last-move)
                                   node))))))
    ;; Are there any other undiscovered passages?
    (let ((new-moves (map direction->move
                          (undiscovered-directions location))))
      (if (or (cycle? (stack-peek nodes))
              (null? new-moves))
          (begin
            ;; Remove this node from the interesting
            ;; nodes: it's been thoroughly explored.
            (stack-pop! nodes)
            (if (stack-empty? moves)
```

```
                              ;; No moves lest; let's rest. This may change
                              'noop
                              (let ((move (stack-pop! moves)))
                                ;; Push the special backtrack move onto the
                                ;; stack; this helps us distinguish the
                                ;; backtracking case from the case where
                                ;; we've hit a wall.
                                ;;
                                ;; The bump-sensor should obviate the
                                ;; need for this, I think; or not.
                                (stack-push! moves 'backtrack)
                                ;; Go back the way we came.
                                (reverse-move move))))
                            (let ((move (list-ref new-moves
                                                  (bsd-random (length new-moves)))))
                              (stack-push! moves move)
                              move))))))))))

(define (make-known-world)
  (let ((world (make-hash-table))
        (a (make-node))
        (b (make-node))
        (c (make-node))
        (d (make-node))
        (e (make-node))
        (f (make-node)))
    (connect! world 'a 'b right)
    (connect! world 'b 'c down)
    (connect! world 'b 'd right)
    (connect! world 'd 'e down)
    (connect! world 'e 'f down)
    (connect! world 'f 'e right)
    (connect! world 'f 'a down)
    world))

(define default-file (make-parameter "graph"))

(define simulate-graph
  (case-lambda
   ((world agent)
```

```
    (simulate-graph world agent (default-steps)))
   ((world agent steps)
    (simulate-graph world
                    agent
                    steps
                    (default-width)
                    (default-height)
                    (default-font-size)
                    (default-title)
                    (default-file)))
   ((world agent steps width height font-size title file)
    (let ((directory (create-temporary-directory)))
      (simulate
       ;; Order of composition matters, apparently; be thoughtful.
       (compose-environments
        (make-step-limited-environment steps)
        (make-debug-environment agent)
        (make-graph-environment world agent)
        (make-graph-performance-measure world agent)))
      directory))))

(define simulate-graph/animation
  (case-lambda
   ((world agent)
    (simulate-graph/animation world agent (default-steps)))
   ((world agent steps)
    (simulate-graph/animation world
                    agent
                    steps
                    (default-width)
                    (default-height)
                    (default-font-size)
                    (default-title)
                    (default-file)))
   ((world agent steps width height font-size title file)
    (let ((directory (create-temporary-directory)))
      (simulate
       ;; Order of composition matters, apparently; be thoughtful.
       (compose-environments
        (make-step-limited-environment steps)
```

```
        ;; Can't this contain its finalizer? Maybe even give it the
        ;; terminal frame?
        (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))))

(define (simulate-comparatively world agent steps width height font-size title)
  (let ((directory (create-temporary-directory)))
    (simulate
     ;; Order of composition matters, apparently; be thoughtful.
     (compose-environments
      (make-step-limited-environment steps)
      ;; Can't this contain its finalizer? Maybe even give it the
      ;; terminal frame?
      (make-graph-animating-environment world
                                        agent
                                        directory
                                        width
                                        height
                                        font-size
                                        title)
      (make-debug-environment agent)
      (make-graph-environment world agent)
      (make-graph-performance-measure world agent)))
    directory))

;;; We should generalize this.
(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; do
              directory-one
```

```
                directory-two
                composite-directory)
        ((make-animation-finalizer composite-directory composite-file)))))))

(let* ((world (make-preferential-depth-first-world 20))
       (start (random-start world)))
  (let ((stateful-agent (make-stateful-graph-agent start))
        (random-agent (make-randomized-reflex-graph-agent start)))
    (parameterize ((default-steps 10)
                   (randomize! bsd-randomize)
                   (random-seed 0))
      (compare-graphs world
                      stateful-agent
                      "Stateful agent"
                      random-agent
                      "Random agent"
                      "composite-agent-harro"))))
```

35.5.3 DONE Allow specification of random-seeds

CLOSED: *2012-07-16 Mon 12:44*

35.5.4 DONE Stateful graph agent

CLOSED: *2012-07-16 Mon 12:44*

The stateful graph agent has to divine indirectly that it encountered an obstacle based on whether or not the last action was movement and whether or not it's in the same position (this will change slightly with the bump sensor in 2.12).

```
- make-stateful-graph-agent start
  - let
    - world make-hash-table
      - # This is going to have to be a stack: we're iterating here,
        after all. Ah, the problem is that we can't recurse back up the
        tree because the relative directions are asymmetrical.
      - # I knew this would bite me in the ass; or is there a more
        expensive way to backtrack? Can't e.g. try random directions
```

```
    because there's no way back.
  - # Is it possible, somehow, to progress through the world via
    random walk; turning away when we encounter a taken path? No, we
    need to be able to backtrack.
  - # Fuck it: just wrote a preferential depth-first world-generator
    that's not quite as good as Barabsi-Albert in the sense that it
    doesn't generate cycles.
  - # On the other hand, wouldn't it have been equivalent to store
    the desirable node and the direction required to get there? I
    still need the inverse of the current direction to backtrack, I
    believe.
  - # It would appear as though we're going to have to push a list
    of interesting vertices onto the stack as well as the actions
    required to get here; this notion, therefore, of visit node
    entails: back-tracking until we find it.
- path ()
- visitanda (start)
- last-action #f
- lambda node clean?
  - if null? visitanda
    - noop
    - if dirty?
      - # This is where pre-/post-order becomes interesting;
        pre-order it, so we can maximize our score.
      - clean
      - begin
        - hash-table-update!/default
          - world
          - node
          - lambda location
            - location-set-status (if clean? clean dirty)
            - let*
              - neighbors (location-neighbors location)
                - if eq? last-node node
                  - case last-action
                    - left vector-set! neighbors left no-passage
                    - right vector-set! neighbors right no-passage
                    - up vector-set! neighbors up no-passage
                    - down vector-set! neighbors down no-passage
                  - let last-node (car visitandum)
```

```
                - case last-action
                  - left vector-set! neighbors right last-node
                  - right vector-set! neighbors left last-node
                  - up vector-set! neighbors down last-node
                  - down vector-set! neighbors up last-node
                - set! visitanda (cons node visitanda)
            - location
          - # Let's nominate this make-unknown-location (as opposed to:
            make-dirty-location).
          - make-location clean? #(unknown unknown unknown unknown)
```

35.5.5 CANCELED `make-equilibrium-limited-environment`

`CLOSED:` *2012-07-16 Mon 12:44*

Compare the delta on scores for some sort of equilibrium; or lack of movement?

35.5.6 DONE Compare stateful and randomized reflex agents.

`CLOSED:` *2012-07-16 Mon 12:43*

- CLOSING NOTE *2012-07-16 Mon 12:43*
  See http://youtu.be/B28ay_zSnoY.

35.5.7 DONE Figure out the correct aspect ratio for youtube.

`CLOSED:` *2012-07-08 Sun 05:42*

- CLOSING NOTE *2012-07-08 Sun 05:42*
  Involves the viewport, &c.

See this; ideally, this would start at graphviz.

35.6 TODO Should we structure this somehow as a blog instead of an org-doc?

Can we create a jekyll-based blog from this org-doc and push it to the github pages?

That would be beautiful: we could get rid of the generated pdf and might not be confined to the relative obscurity of org (relative to, say, html).

35.7 TODO Some sort of blog post or other publicity?

35.8 TODO Find a reasonable pseudocode package in LaTeX.

See this survey. Algorithm2e isn't bad; doesn't seem to have a function, though. Pseudocode seems to be relatively natural; even if the output is a little ugly.

The alternative, I suppose, is straight up lists.

Algorithm2e has to be wrapped in dollars, which sucks; also: bizarre camel-case macros. Looks good, otherwise. Has no functions, apparently, either.

35.9 TODO Should we tangle to a bunch of text files?

Looking for an alternative to the big-ass pdf.

35.10 DONE Reimplement the Lisp environment in Scheme.

`CLOSED:` *2012-06-01 Fri 03:09*

Should we try to map CLOS to coops? Or maybe TinyCLOS would suffice. This takes balls. See aima-chicken.

35.11 DONE Personal notes as footnotes.

`CLOSED:` *2012-06-01 Fri 03:09*

35.12 CANCELED Should we try to release an e.g. Wumpus World egg?

`CLOSED:` *2012-06-01 Fri 03:08*

- CLOSING NOTE *2012-06-01 Fri 03:08*
  This is superseded by the Chicken port of the Lisp implementation (aima-chicken).

I wonder if it would be worthwhile to study the canonical Lisp examples.