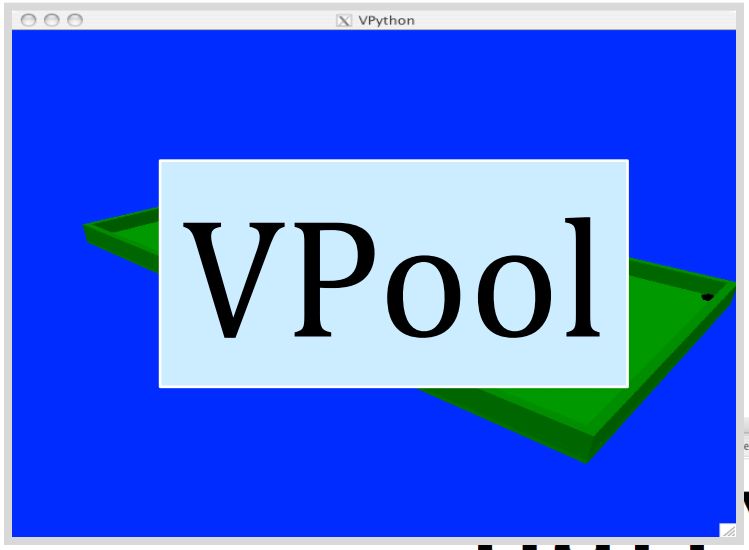
Today: Find a Cool Project

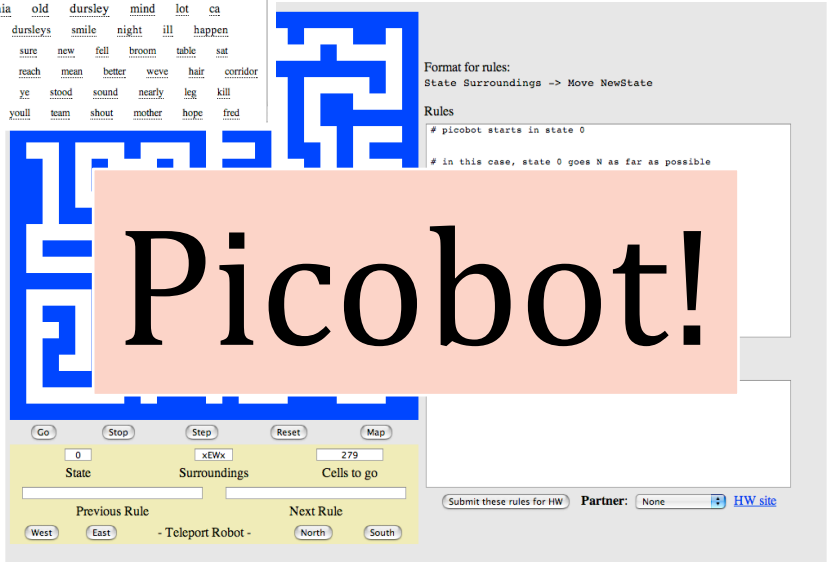


C4 AI





TextID

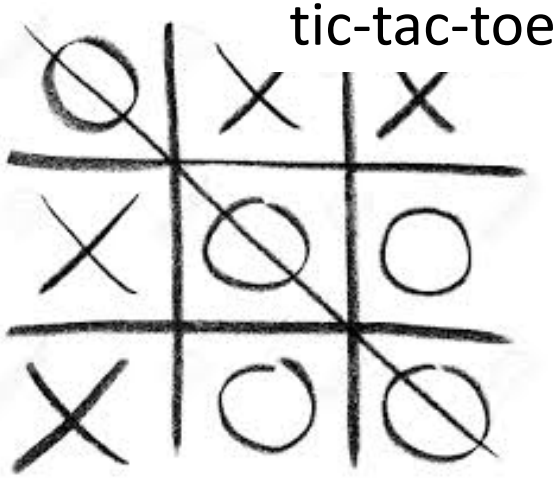


Picobot!

Connecting Connect Four ...



tic-tac-toe







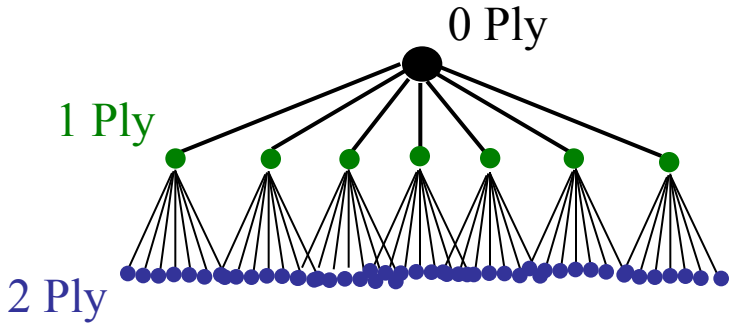
How complex are these games, from least to most?

... to other strategy games.

Games' Branching Factors

On average, Connect 4 players have **seven choices** per move.

Chess players have more, **perhaps around 40**, possible choices in a given move.



Boundaries for ***qualitatively*** different games...

Branching Factors
for different two-player games

Tic-tac-toe	4
Connect Four	7
Checkers	10
Reversi	30
Chess	40
Go	300
Arimaa	17,000

"solved" games

computer-dominated

human-dominated

only 'til 2015-16

A Knowledge-based Approach of Connect-Four

The Game is Solved: White Wins

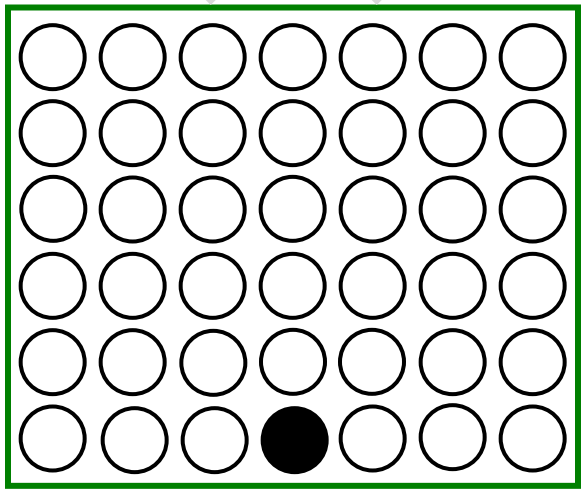
Victor Allis

Department of Mathematics and Computer Science

Vrije Universiteit

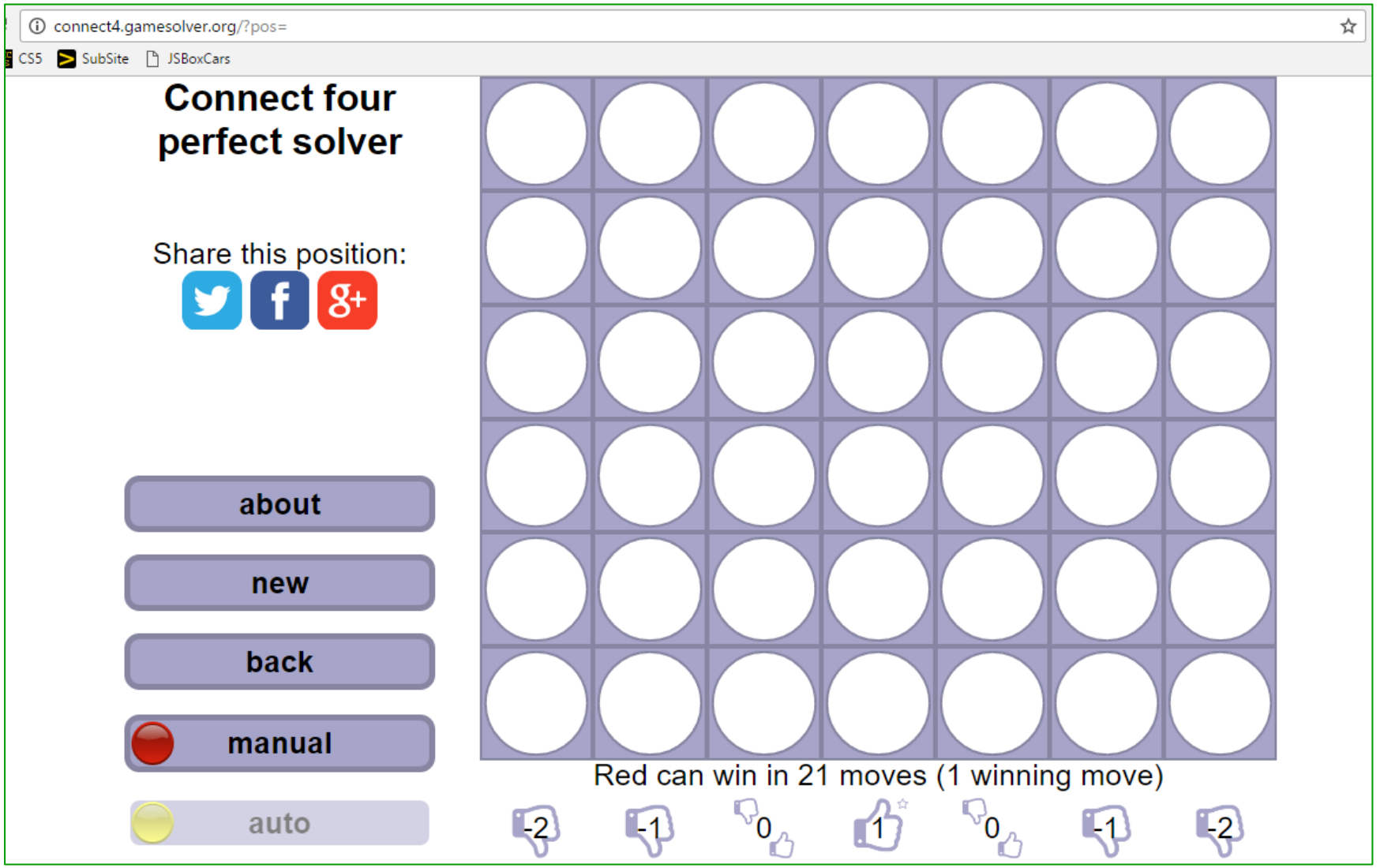
Amsterdam, The Netherlands

Masters Thesis, October 1988 †



Connect 4 was solved in **1988.**

Nice site showing ideal play...



Connect 4 is solved ~ like tic-tac-toe

Science 14 September 2007:
Vol. 317. no. 5844, pp. 1518 – 1522
DOI: 10.1126/science.1144079

RESEARCH ARTICLES

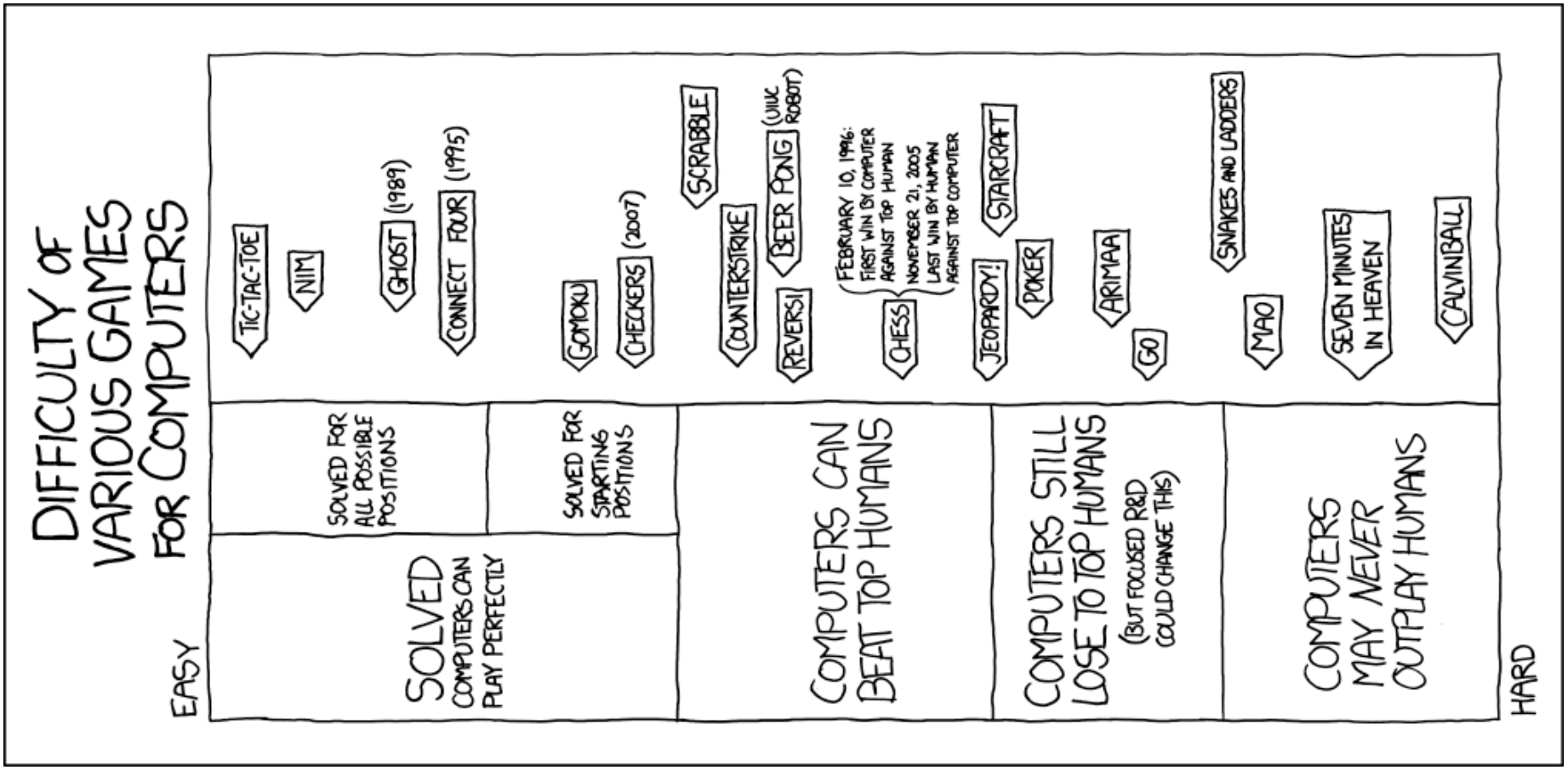
Checkers Is Solved

Jonathan Schaeffer,* Neil Burch, Yngvi Björnsson,† Akihiro Kishimoto,‡ Martin Müller, Robert Lake, Paul Lu, Steve Sutphen

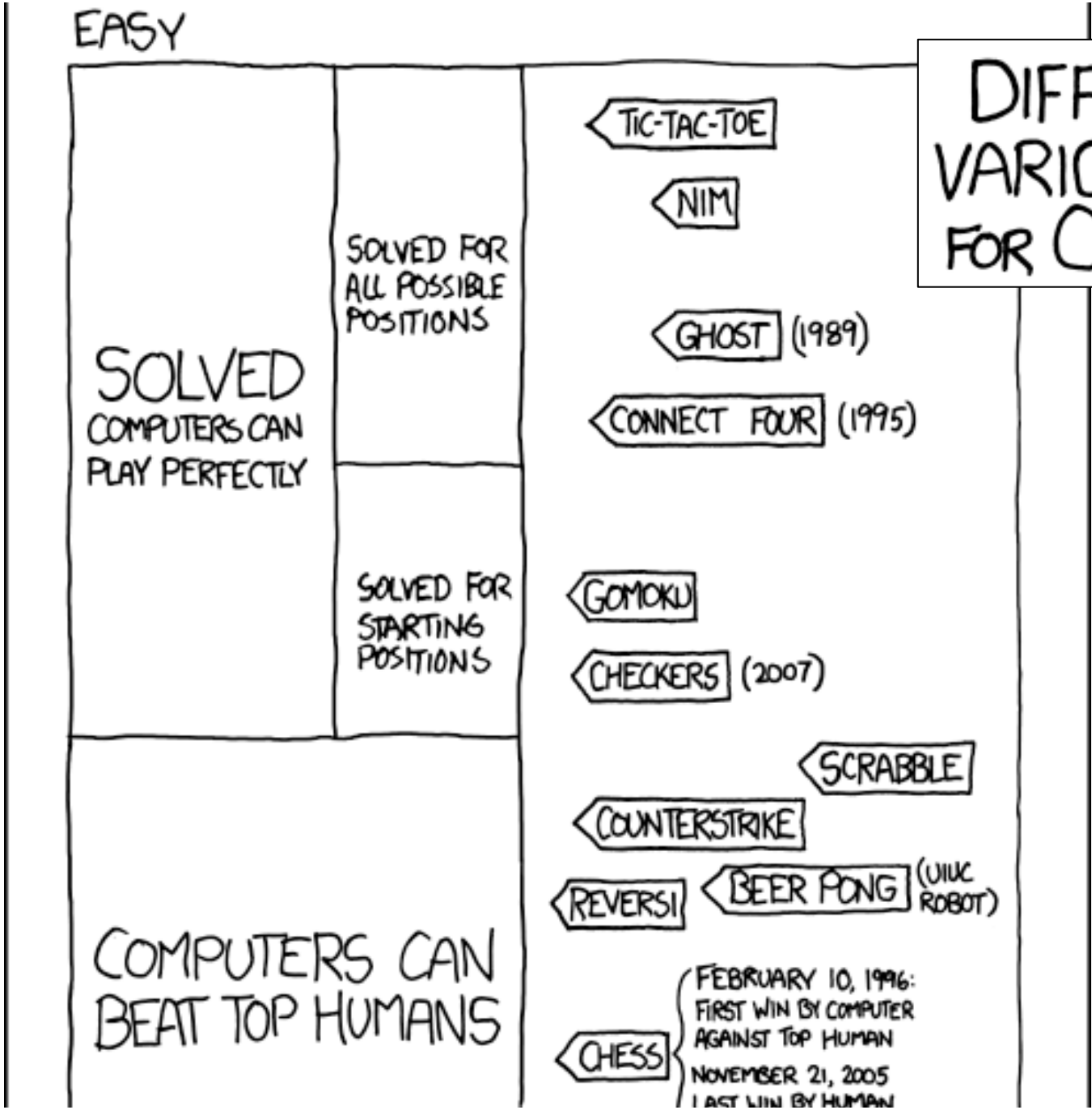
The game of checkers has roughly 500 billion billion possible positions (5×10^{20}). The task of solving the game, determining the final result in a game with no mistakes made by either player, is daunting. Since 1989, almost continuously, dozens of computers have been working on solving checkers, applying state-of-the-art artificial intelligence techniques to the proving process. This paper announces that checkers is now solved: Perfect play by both sides leads to a draw. This is the most challenging popular game to be solved to date, roughly one million times as complex as Connect Four. Artificial intelligence technology has been used to generate strong heuristic-based game-playing programs, such as Deep Blue for chess. Solving a game takes this to the next level by replacing the heuristics with perfection.

Checkers was solved in **2007**.

XKCD: Games' Branching Factors

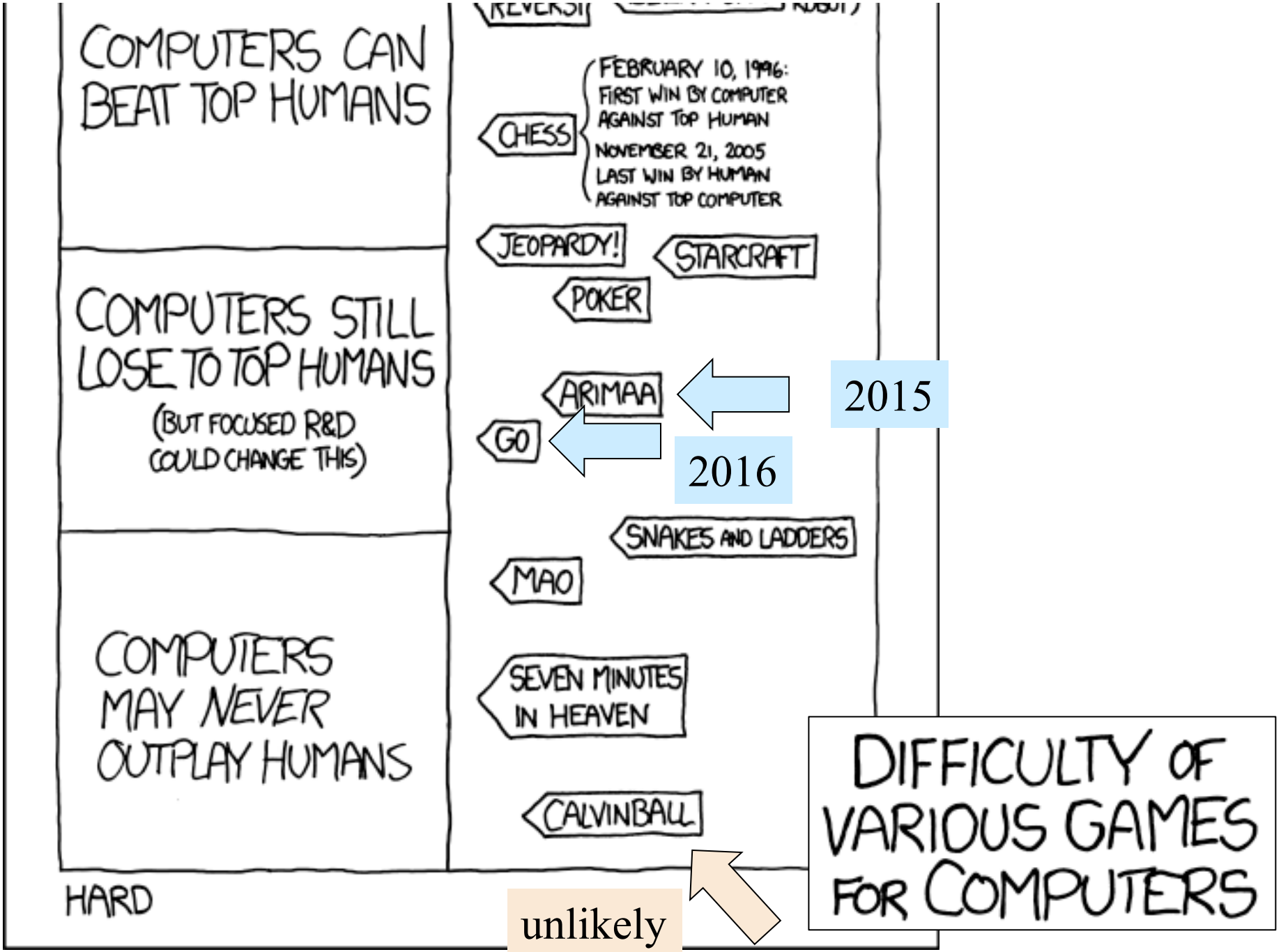


https://xkcd.com/1002/



DIFFICULTY OF VARIOUS GAMES FOR COMPUTERS

(Games' Branching Factors)


COMPUTERS CAN BEAT TOP HUMANS
COMPUTERS STILL LOSE TO TOP HUMANS
(BUT FOCUSED R&D COULD CHANGE THIS)
COMPUTERS MAY NEVER OUTPLAY HUMANS
HARD
CHESS
FEBRUARY 10, 1996: FIRST WIN BY COMPUTER AGAINST TOP HUMAN
NOVEMBER 21, 2005 LAST WIN BY HUMAN AGAINST TOP COMPUTER
JEOPARDY!
STARCRAFT
POKER
ARIMAA
2015
GO
2016
SNAKES AND LADDERS
MAO
SEVEN MINUTES IN HEAVEN
CALVINBALL
unlikely
DIFFICULTY OF VARIOUS GAMES FOR COMPUTERS

COMPUTER
BEAT TOP H

COMPUTERS
LOSE TO TOP
(BUT FOCUSED
COULD CHANGE

COMPUTER
MAY NEVE
OUTPLAY HU

HARD

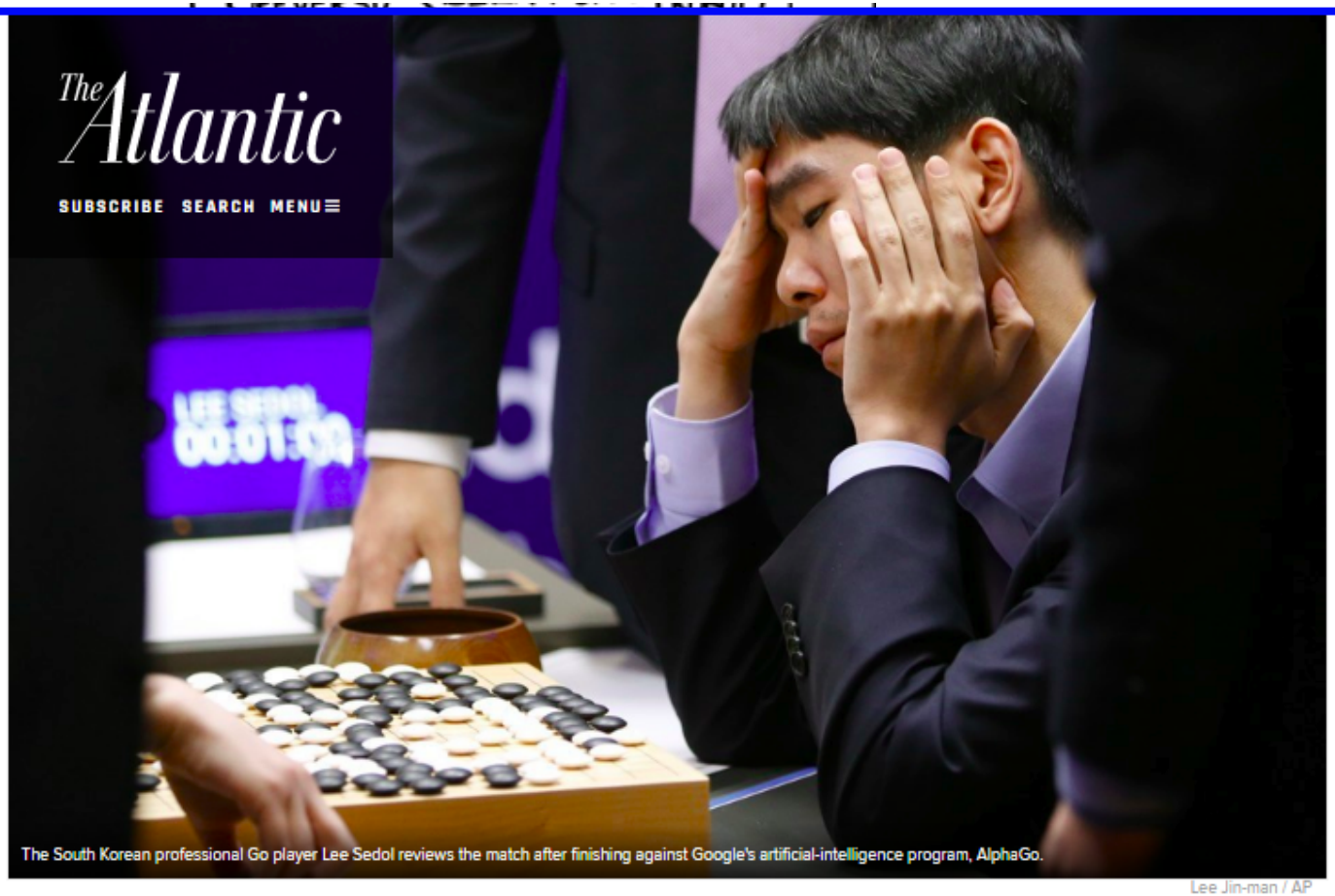
CALVINBALL

VARIOUS GAMES
FOR COMPUTERS

unlikely

The Atlantic

SUBSCRIBE SEARCH MENU



The South Korean professional Go player Lee Sedol reviews the match after finishing against Google's artificial-intelligence program, AlphaGo.

Lee Jin-man / AP

How Google's AlphaGo Beat a Go World Champion





CALVINBALL

DIFFICULTY OF VARIOUS GAMES FOR COMPUTERS

'tis the season for final projects…

Today is about your final projects
+ it's a *sales pitch* for possible options:

vPool

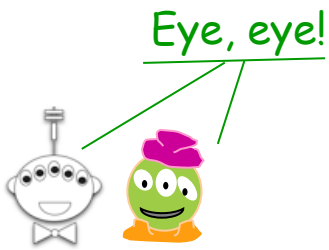
TextID

Picobot

Connect 4 AI



Final projects

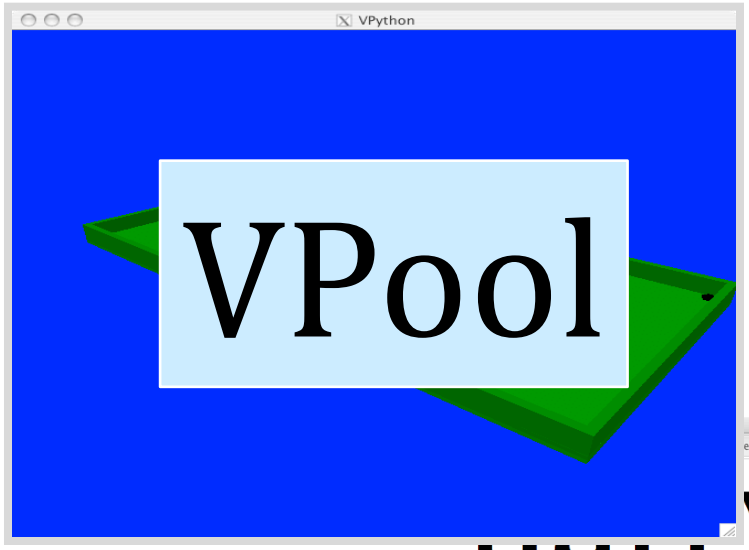


Working in teams of 1-3 is OK

Teams need to work together—*in the same place*—and need to share the work equally

Teaming is extra-encouraged on the final project!

The projects...

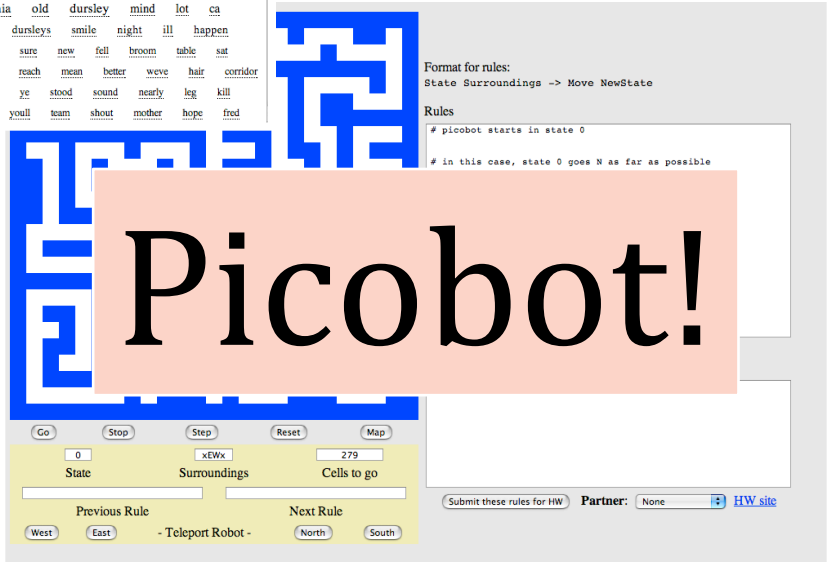




TextID

C4 AI





Picobot!

Connect Four ...

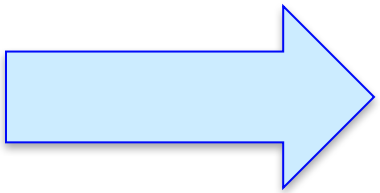
- Create an N-ply lookahead
- Allow computer vs. computer and computer vs. human games of any difficulty...
- Extend the game in some way:

(1) Change the rules!

- Gravity
- Randomness
- WildCards
- Tetris

(2) Graphics

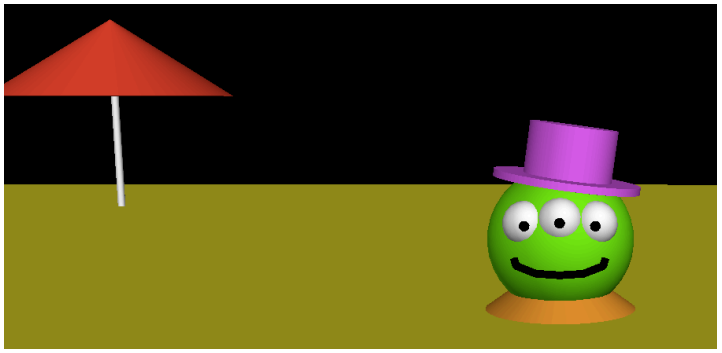
- 2d or 3d



vPool

3d graphics-based game using VPython





... it's not really very constrained at all!

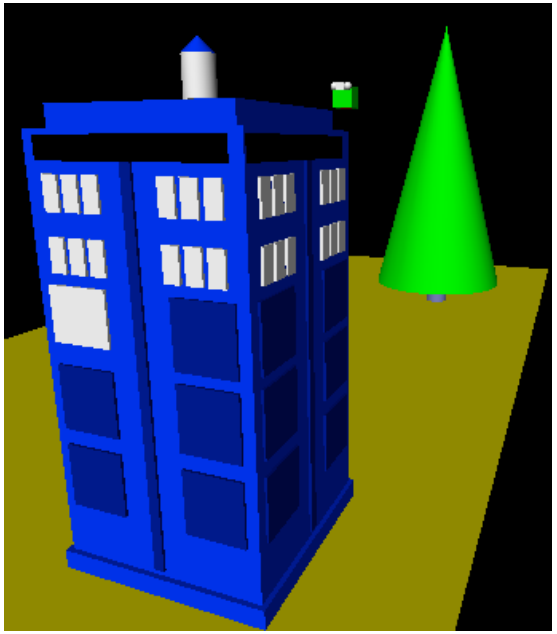
A few constraints...

need ≥4 physically interacting objects

allow the user to direct 1+ objects, either by keyboard or mouse or both

needs a game goal + be winnable!

must detect **some** "linear" and some "spherical" collisions and implement their results on the motion of the objects



The vPool project

VPython was designed to make 3d physics simulations simpler to program – as a result, the library itself is physics-free!

⇒ Phunky Physics is welcome!

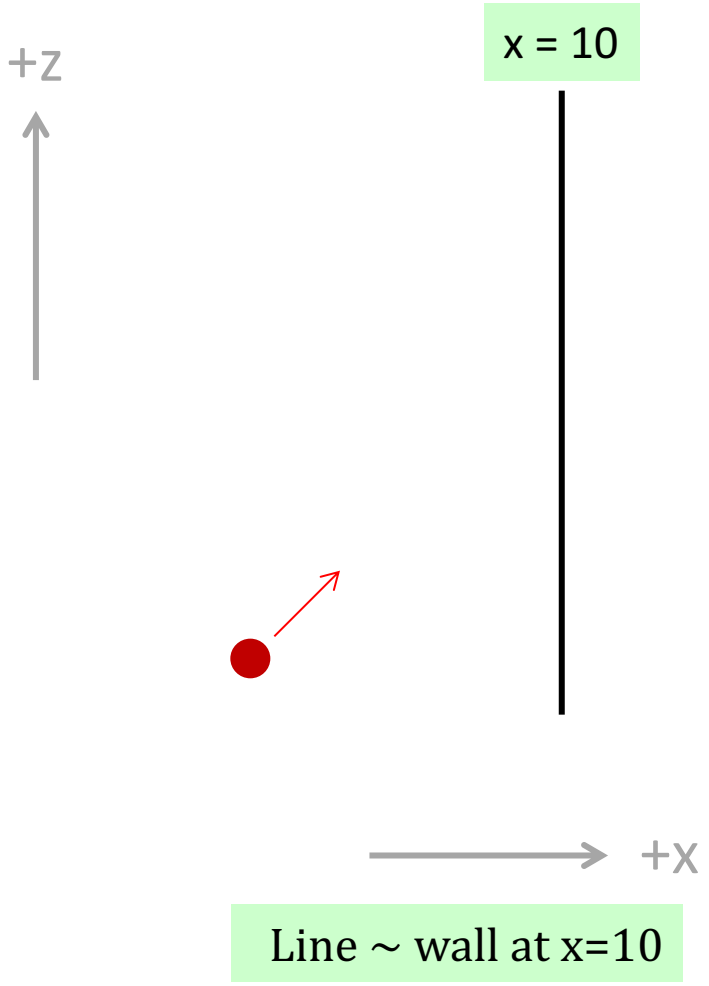
sounds like *all* physics to me!



- ***Linear collisions*** should be somewhere ("walls")
- ***Spherical collisions*** should be somewhere ("points")
- You need "pockets"—*or some other game objective*
- You need **user control** of at least one object (mouse/kbd)

vPython: Linear collisions

At least some of the game needs to be about *detecting collisions* and *changing velocities*

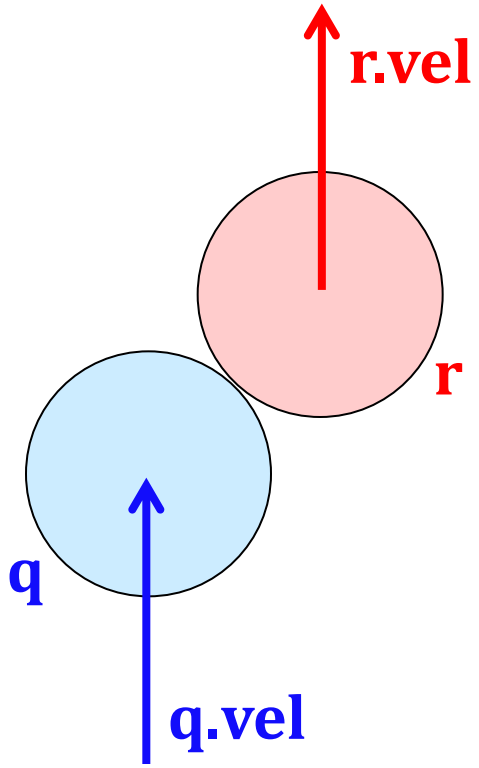



```
ball.vel = vector(1,0,1)
while True:
    ball.pos += dt*ball.vel
    if ball.pos.x > 10:
        ball.vel.x
        ball.pos.x
```

How to **bounce**?

What else to do?

Spherical collisions

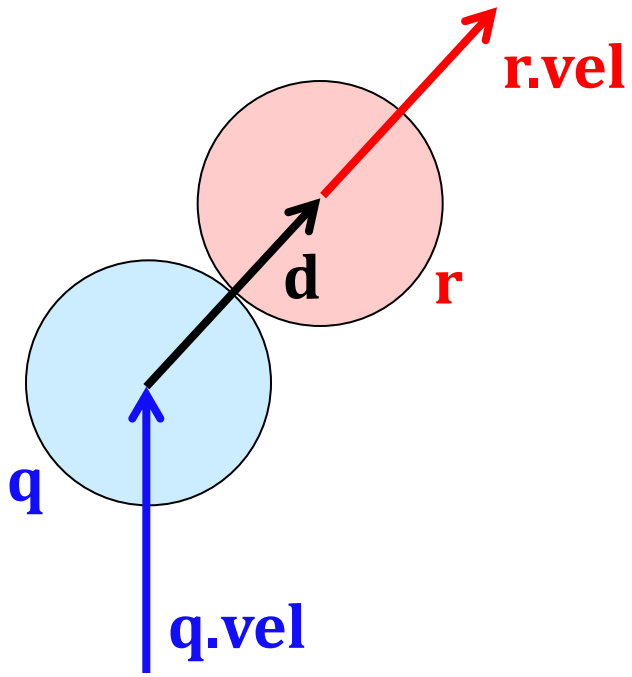


1 First approximation:

Stop **q**. *Undo any overlap.*

Make **r**.vel = **q**.vel.

Spherical collisions



1 First approximation:

Stop **q**. *Undo any overlap.*

Make **r**.vel = **q**.vel.

2 Second approximation:

Stop **q**. *Undo any overlap.*

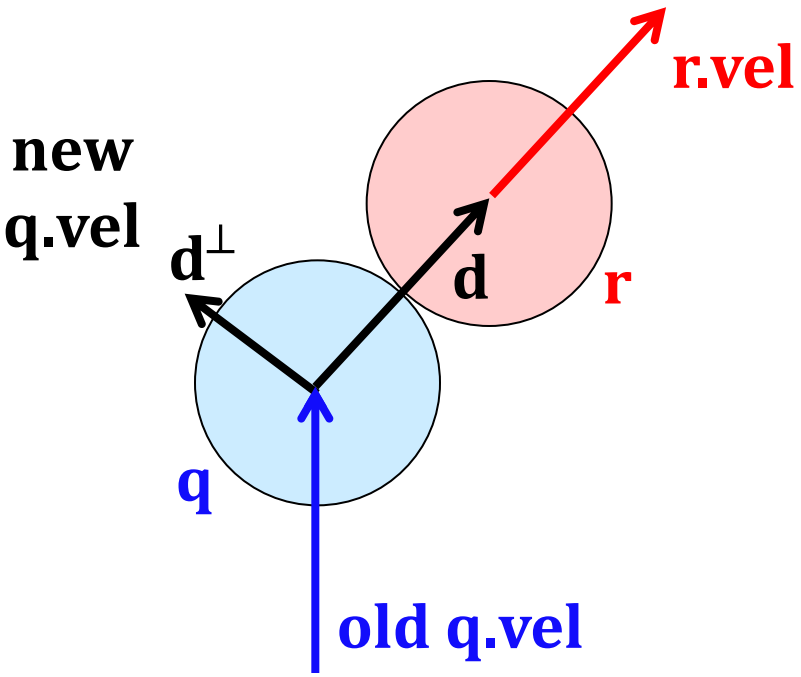
Compute **d** = **r**.pos – **q**.pos

Make **r**.vel = **d**



Reality is just two eyes away!

Spherical collisions



1 First approximation:

Stop **q**. *Undo any overlap.*

Make **r**.vel = **q**.vel.

2 Second approximation:

Stop **q**. *Undo any overlap.*

Compute **d** = **r**.pos – **q**.pos

Make **r**.vel = **d**

3 Third approximation:

Same as **second**, but

Make **q**.vel = $\mathbf{d}^{\perp}$, at 90° from **d**



Reality is just one eye away!

the TextID *IDer*...

Though Robin Ellacott's twenty-five years of life had se... moments o...





April 4, 2013

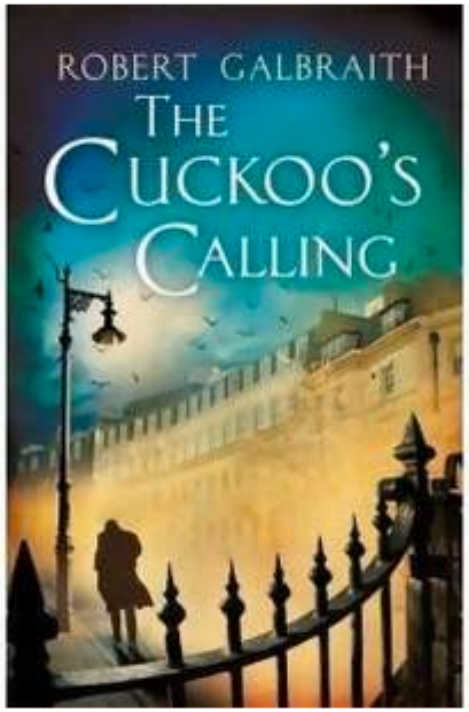
paragraph of The Cuckoo's Calling by R. Galbraith

How Robert Galbraith was found to be JK Rowling

SARAH PAVIS · AUG 08 2013

I was given e-text copies of *Cuckoo* to compare against Rowling's own *The Casual Vacancy*, Ruth Rendell's *The St. Zita Society*, P.D. James' *The Private Patient* and Val McDermid's *The Wire in the Blood*. [...]

I actually ran four separate types of analyses focusing on four different linguistic variables. While anything can in theory be an informative variable, my work focuses on variables that are easy to compute and that generate a lot of data from a given passage of language. One variable that I used, for example, is the distribution of word lengths. Each novel has a lot of words, each word has a length, and so one can get a robust vector of % of the words in this document have exactly letters. Using a distance formula (for the mathematically minded, I used the normalized cosine distance formula instead of the more traditional Euclidean distance you remember from high school), I was able to get a measurement of similarity, with 0.0 being identity and progressively higher numbers being greater dissimilarity.



kottke.org/13/08/how-jk-rowling-was-found-to-be-robert-galbraith

the TextID project

Big ideas:

(1) Build ***lexical*** models of bodies of text…

(2) *Use a similarity score to measure*

Rowlingness
Shakepearity
vs.

NYTimes-iness
WSJournalicity
vs.

Big Bang Theory
Modern Family
vs.

even better: *your own choice of two or more comparisons…*

File

```
I like poptarts and 42 and spam. Spamful poptarts are
like poptartful spams -- and are liked by all!
Will _Thanksgiving_ bring spam poptarts?
```

`class TextModel`

... contains at least five Python dictionaries, e.g.,

count of each word

{'and': 3, 'poptartful': 1, 'liked': 1, 'spamful': 1, 'like': 2, '': 1, 'spam': 2, 'i': 1, '42': 1, 'all': 1, 'thanksgiving': 1, 'will': 1, 'bring': 1, 'poptarts': 3, 'spams': 1, 'by': 1, 'are': 2}

{1: 1, 2: 2, 3: 6, 4: 5, 5: 3, 7: 1, 8: 3, 10: 1, 12: 1}

???

{'and': 3, '': 1, 'all': 1, 'like': 3, 'thanksgiv': 1, 'spam': 4, 'i': 1, '42': 1, 'by': 1, 'will': 1, 'bring': 1, 'ar': 2, 'poptart': 4}

???

{12: 1, 5: 1, 7: 1}

???

???

{'!': 1, '-': 2, '?': 1, '_': 2, '.': 1}

What are these four other dictionaries counting?!

File

```
I like poptarts and 42 and spam. Spamful poptarts are
like poptartful spams -- and are liked by all!
Will _Thanksgiving_ bring spam poptarts?
```

`class TextModel`

... contains at least five Python dictionaries, e.g.,

count of each word

{'and': 3, 'poptartful': 1, 'liked': 1, 'spamful': 1, 'like': 2, '': 1, 'spam': 2, 'i': 1, '42': 1, 'all': 1, 'thanksgiving': 1, 'will': 1, 'bring': 1, 'poptarts': 3, 'spams': 1, 'by': 1, 'are': 2}

{1: 1, 2: 2, 3: 6, 4: 5, 5: 3, 7: 1, 8: 3, 10: 1, 12: 1}

word lengths

{'and': 3, '': 1, 'all': 1, 'like': 3, 'thanksgiv': 1, 'spam': 4, 'i': 1, '42': 1, 'by': 1, 'will': 1, 'bring': 1, 'ar': 2, 'poptart': 4}

stems

{12: 1, 5: 1, 7: 1}

sentence lengths

punctuation {'!': 1, '-': 2, '?': 1, '_': 2, '.': 1}

What are these four other dictionaries counting?!

TextID's *building blocks...*

This looks familiar...



(1) Get text from file...

(2) Split up the text into words (first pass)

(3) Model punctuation marks (optional)

(4) Model sentence lengths (using `'.!?'`)

text with punctuation preserved

(5) Take a breather...

(6) Clean up the words (second pass)

(7) Model words and word lengths

(8) Stem words and model those stems

text with punctuation removed

(9) You're ready to score against your model!

TextID's *library resources*

I bet they told you the age of the library was over... but It's just begun!



7.1. `string` — Common string operations

5.6.1. String Methods

`string.` **`punctuation`**
String of ASCII characters which are considered punctuation

`str.` **`replace`**(*old*, *new*[, *count*])
Return a copy of the string with all occurrences of substring *old* replaced by *new*.

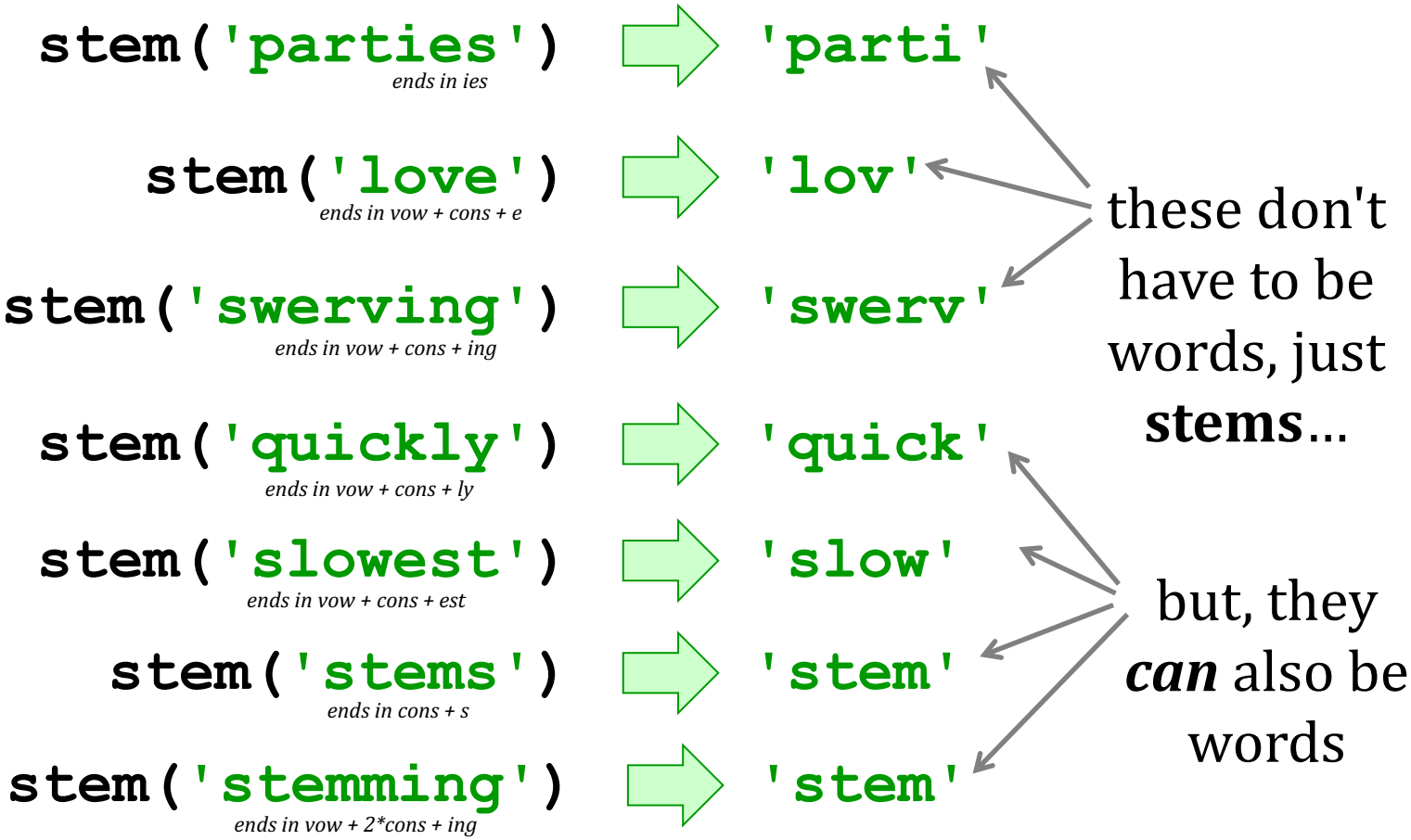
`str.` **`lower`**()
Return a copy of the string with all the cased characters [4] converted to lowercase.

`str.` **`split`**([*sep*[, *maxsplit*]])
Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

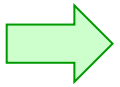
If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

Stemming

An algorithm that outputs the **root** of the input word.



either way, they ***all*** have exceptions!



Model *matching*

Suppose we have two ***trained models***:

WS: { "love": 50,
"spell": 8,
"thou": 42 }

JKR: { "love": 25,
"spell": 275,
"potter": 700 }



how do we handle different-sized texts?



Unknown text: { "love": 3,
"thou": 1,
"potter": 2,
"spam": 4 }

These must have been some *really* avant-garde texts!

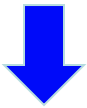


Model *matching*

Suppose we have two ***trained models***:

WS: { "love": 50,
"spell": 8,
"thou": 42 }

JKR: { "love": 25,
"spell": 275,
"potter": 700 }



normalize for size



WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

Unknown text: { "love": 3,
"thou": 1,
"potter": 2,
"spam": 4 }

Model *matching*

Suppose we have two ***normalized models***:

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

How do we compare the models with an unknown text?

What's the **likelihood** of the new model arising from each?

Unknown text: { "love": 3,
"thou": 1,
"potter": 2,
"spam": 4 }

There's probably a way to do this!



Model *matching*

Suppose we have two ***normalized models***:

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

The ***WS***-based probability of each word in ***Unknown text***

.50 • .50 • .50 • _?_ • ? • ? • ? • ? • ? • ? = ???

love love love thou potter potter spam spam spam spam

0? 1?

Unknown text: { "love": 3, "potter": 2,
"thou": 1, "spam": 4 }

I've got near-zero ideas on this one!



Model *matching*

Suppose we have two ***normalized models***:

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

The ***WS***-based probability of each word in ***Unknown text***

.50 • .50 • .50 • .42 • 1 • 1 • 1 • 1 • 1 • 1 = 0.0525

love love love thou potter potter spam spam spam spam

Why is this *especially* incorrect? ???

Unknown text: { "love": 3, "potter": 2,
"thou": 1, "spam": 4 }

I've got near-zero ideas on this one!



Model *matching*

Suppose we have two ***normalized models***:

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

The ***WS***-based probability of each word in ***Unknown text***

Correct, but ***not helpful***!

.50 • .50 • .50 • .42 • 0 • 0 • 0 • 0 • 0 • 0 = 0.0

love love love thou potter potter spam spam spam spam

???

Unknown text: { "love": 3, "potter": 2,
"thou": 1, "spam": 4 }

I've got near-zero ideas on this one!



Model *matching*

Suppose we have two ***normalized models***:

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

The ***WS***-based probability of each word in ***Unknown text***

For missing words, use **half** the smallest value – *across both normalized models!*

.50 • .50 • .50 • .42 • .012 • .012 • .012 • .012 • .012 • .012 = 1.57e-13

love love love thou potter potter spam spam spam spam

3 1 2 4

What?

Unknown text: { "love": 3, "potter": 2,
"thou": 1, "spam": 4 }

half of ϵ !



Model *matching*

Suppose we have two ***normalized models***:

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

the ***WS***-based probability of each word in ***Unknown text***

for missing words, use **half** the smallest value – *across both normalized models!*

.50 • **.50** • **.50** • **.42** • .012 • .012 • .012 • .012 • .012 • .012 = 1.57e-13

love love love thou potter potter spam spam spam spam

3 1 2 4

too small!

take logs of everything!

Unknown text: { "love": 3, "potter": 2,
"thou": 1, "spam": 4 }

half of ϵ !



Model *matching*

Suppose we have two ***normalized models***:

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

the ***WS***-based probability of each word in ***Unknown text***

for missing words, use **half** the smallest value – *across both normalized models!*

3*log(.50) + log(.42) + 2*log(.012) + 4*log(.012) = -29.48

OK!

love love love thou potter potter spam spam spam spam

3 1 2 4

take logs of everything!

Unknown text: { "love": 3, "potter": 2,
"thou": 1, "spam": 4 }

half of ϵ !



Model *matching*

Suppose we have two ***normalized models***:

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

the ***WS***-based probability of each word in ***Unknown text***

3*log(.025) + log(.012) + 2*log(.7) + 4*log(.012) = -33.89

love love love thou potter potter spam spam spam spam

3 1 2 4

Unknown text: { "love": 3, "potter": 2,
"thou": 1, "spam": 4 }

this ***looks*** close…



Model *matching*

Suppose we have two ***normalized models***:

WS: { "love": 0.50,
"spell": 0.08,
"thou": 0.42 }

JKR: { "love": 0.025,
"spell": 0.275,
"potter": 0.700 }

Unknown text:
{ "love": 3, "potter": 2,
"thou": 1, "spam": 4 }

-29.48

the (much) better match…

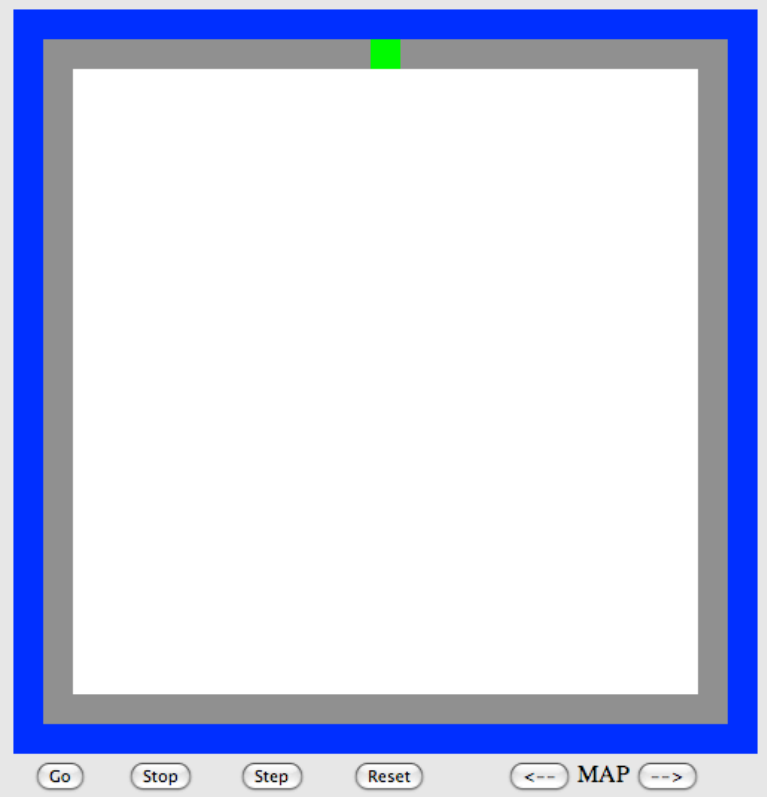
-33.89

The Picobot project

Big ideas

(1) Implement Picobot in Python

(2) *Train Python to write successful Picobot programs!*



Talk about going *full circle...*



Picobot's classes

```
class Program:
```

What in Python could most usefully hold all of these ***rules***?

What type should **`self.rules`** be?

```
0 xxxx -> N 0
0 Nxxx -> W 0
0 NxWx -> S 0
0 xxWx -> S 0
0 xxWS -> E 0
0 xxxS -> E 0
0 xExS -> N 0
0 xExx -> N 0
0 NExx -> S 1
1 xxxx -> S 1
1 Nxxx -> E 1
1 NxWx -> E 1
1 xxWx -> N 1
1 xxWS -> N 1
1 xxxS -> W 1
1 xExS -> W 1
1 xExx -> S 1
1 NExx -> W 0
```

Python dictionary

both *tuples*

key

value

```
self.rules[ (1,"NExx") ] = ("W",0)
```

Picobot's classes

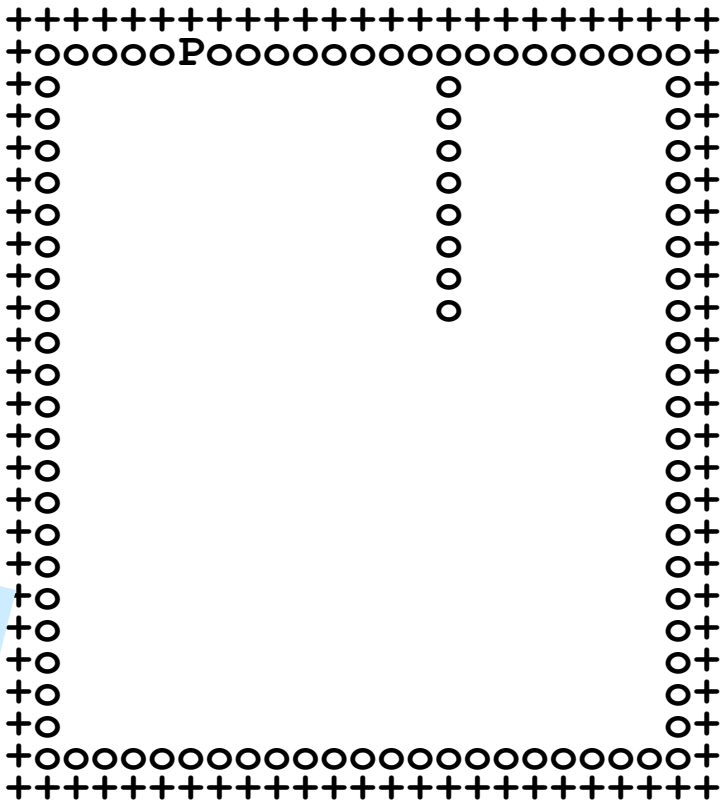
What type in Python could most usefully hold the ***environment***?

```
class World:
```

```
class Board
```

What class that you've already written will be most similar to Picobot's **`World`**?

What will **`self.room`** be?



Wall: `+`

Visited: `o`

Picobot: `P`

The same as the Connect-Four board's **`self.data`**!

a list-of-lists-of-one-character-strings....

The Picobot project

```
Current State: 1
Current Rule:  1 N*W* -> X 2

++++++++++
+o++o+o+++
+ooooooo ++
++++o++   +
+oooo+++++
++++o     +
+oooo+++ +
++++o+++++
+Pooo     +
++++++++++
```

Picobot started here…

and is now here…

First, build an ***ASCII simulation***

then, ***evolve*** it…

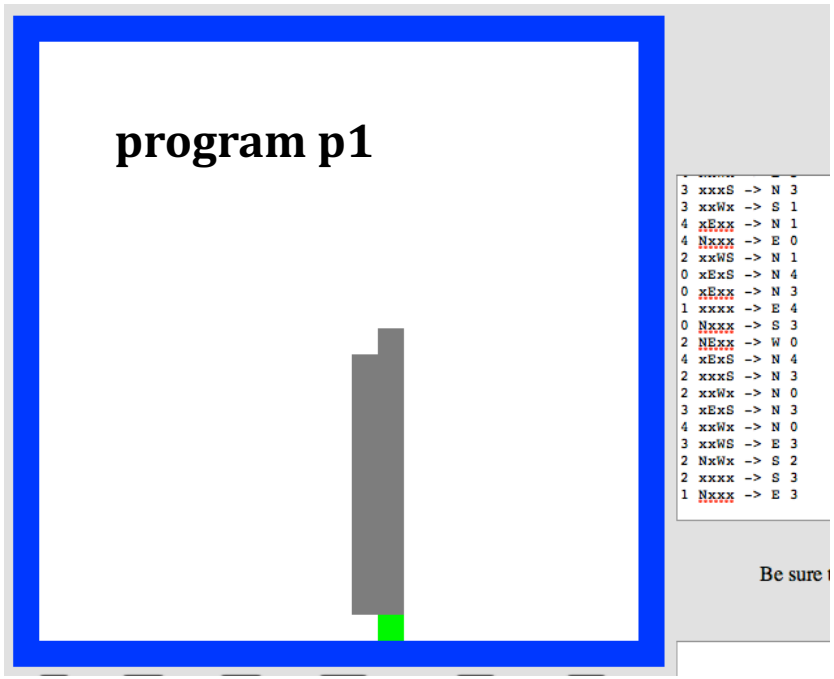
with your own genetic algorithm

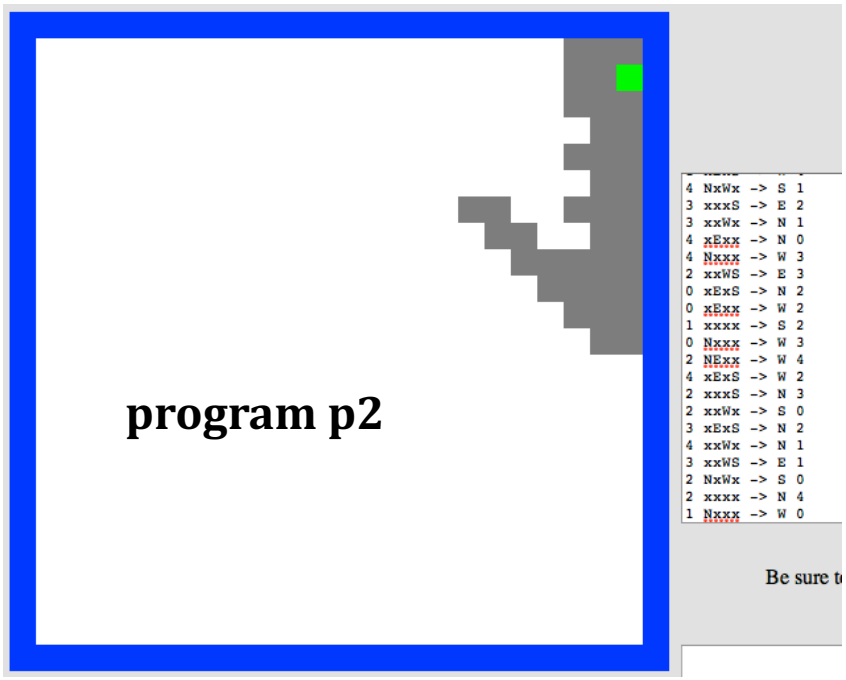
Your actual ASCII is likely to be more monochromatic!

http://rednuht.org/genetic_cars_2/ or http://boxcar2d.com/
Box2d: https://www.youtube.com/watch?v=uxourrlPlf8

Program ***evolution***

An example of ***genetic algorithms***, which are used for optimizing *hard-to-describe* functions with *easily-splittable* solutions.

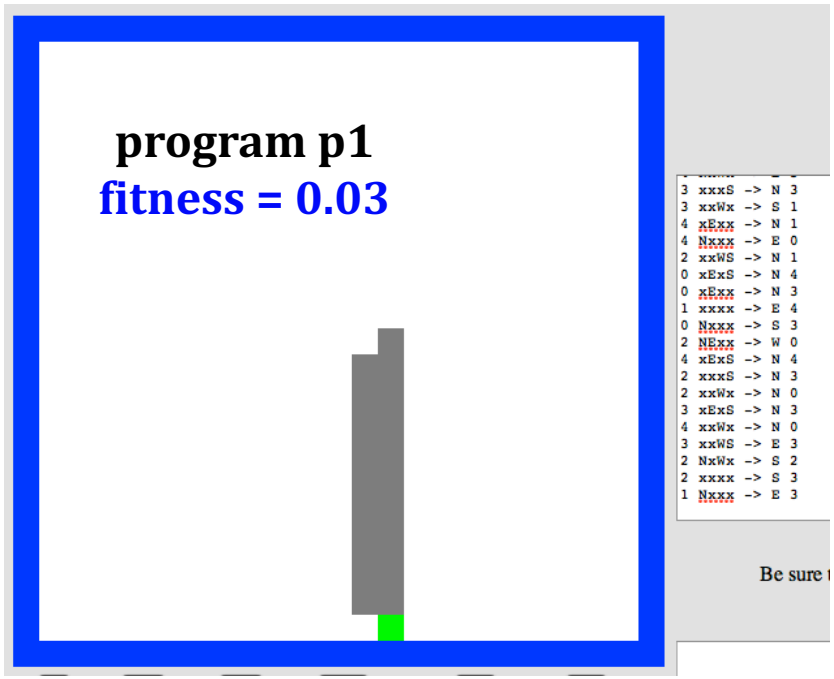


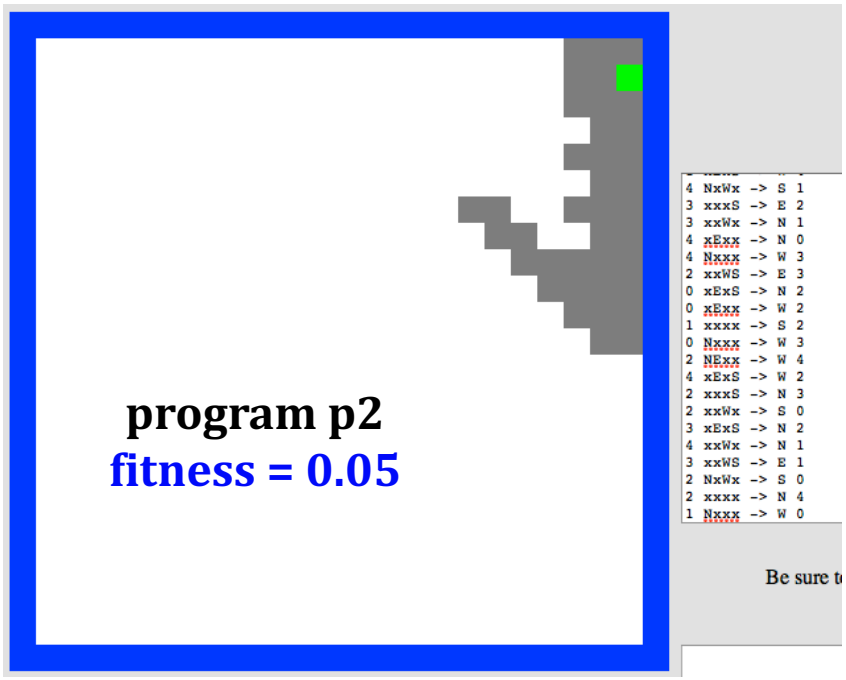


Start with a population of, say, ~200 ***random*** Picobot programs…

Program ***evolution***

An example of ***genetic algorithms***, which are used for optimizing hard-to-describe functions with easily-splittable solutions.

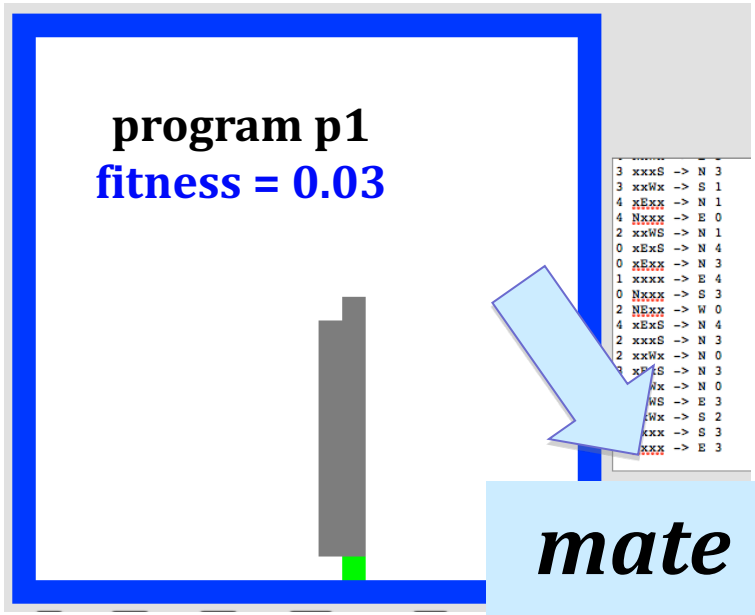


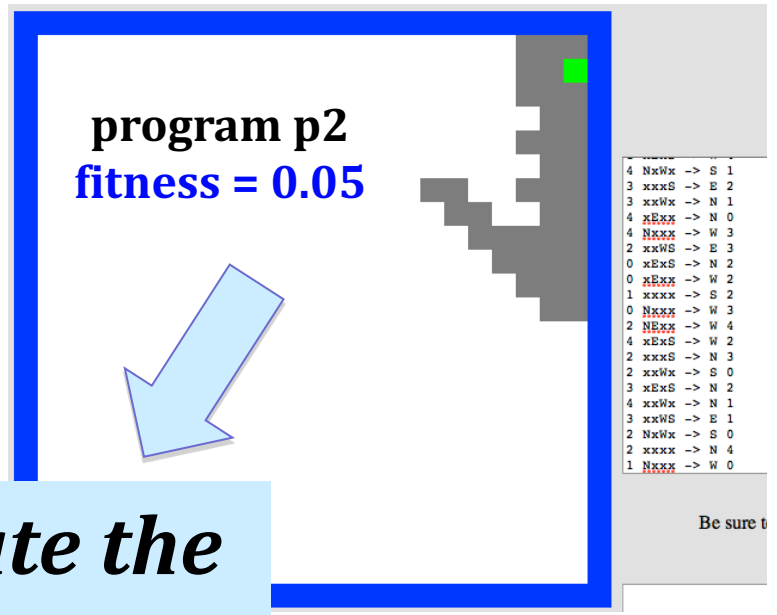


Then, ***measure*** *the fitness* of all of those programs

Measure?
How??

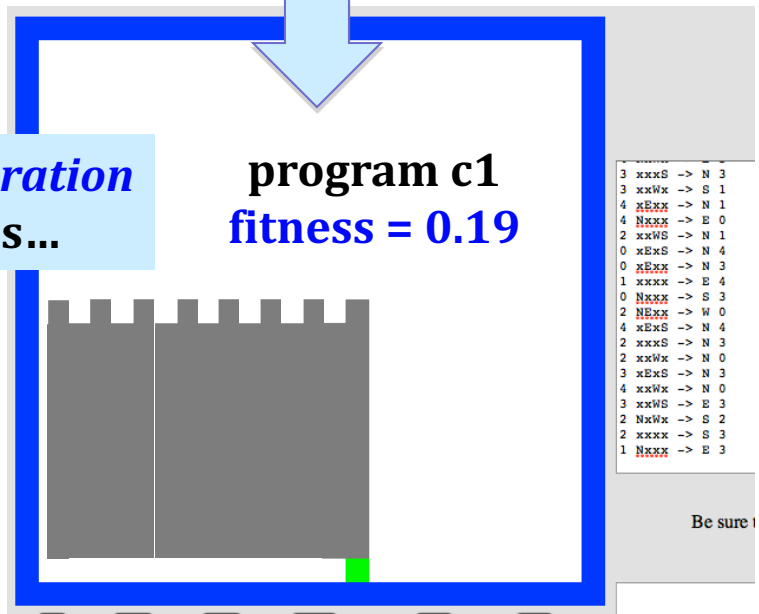






mate + mutate the fittest rulesets

to create a *new generation* of ~200 programs...

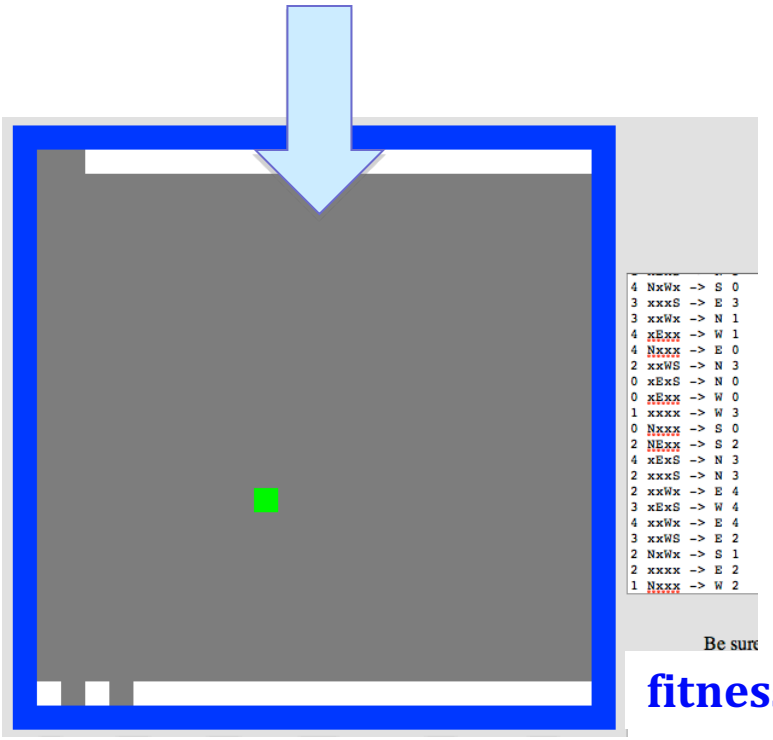


What the goal?



Repeat this "***survival of the fittest***" process for many generations…

… and by the end, your Python code should/will have evolved a much more capable Picobot program!



fitness = 0.90

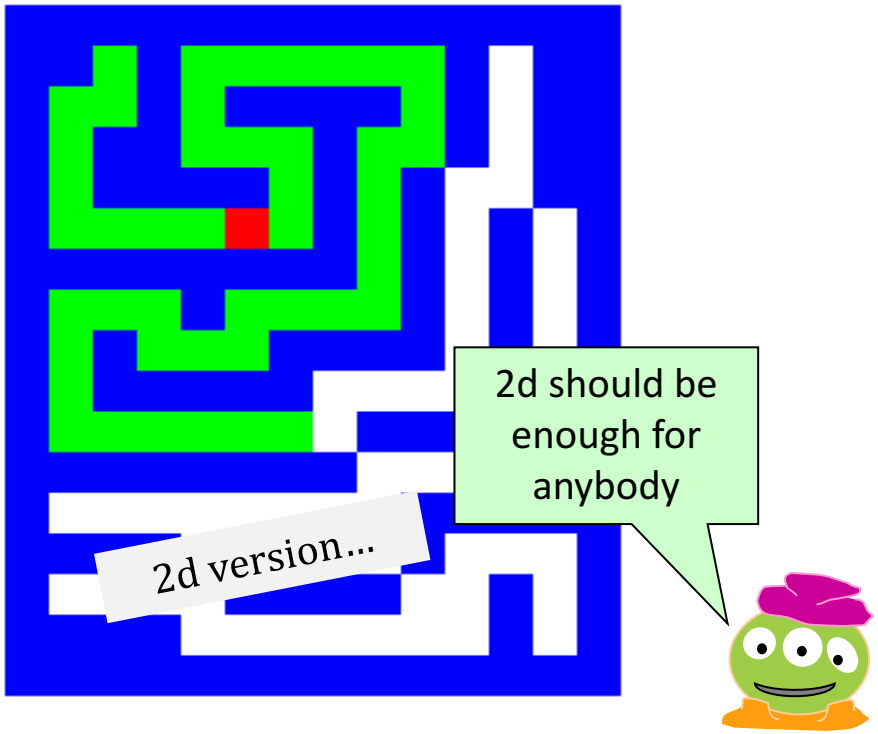
Solving hw0!

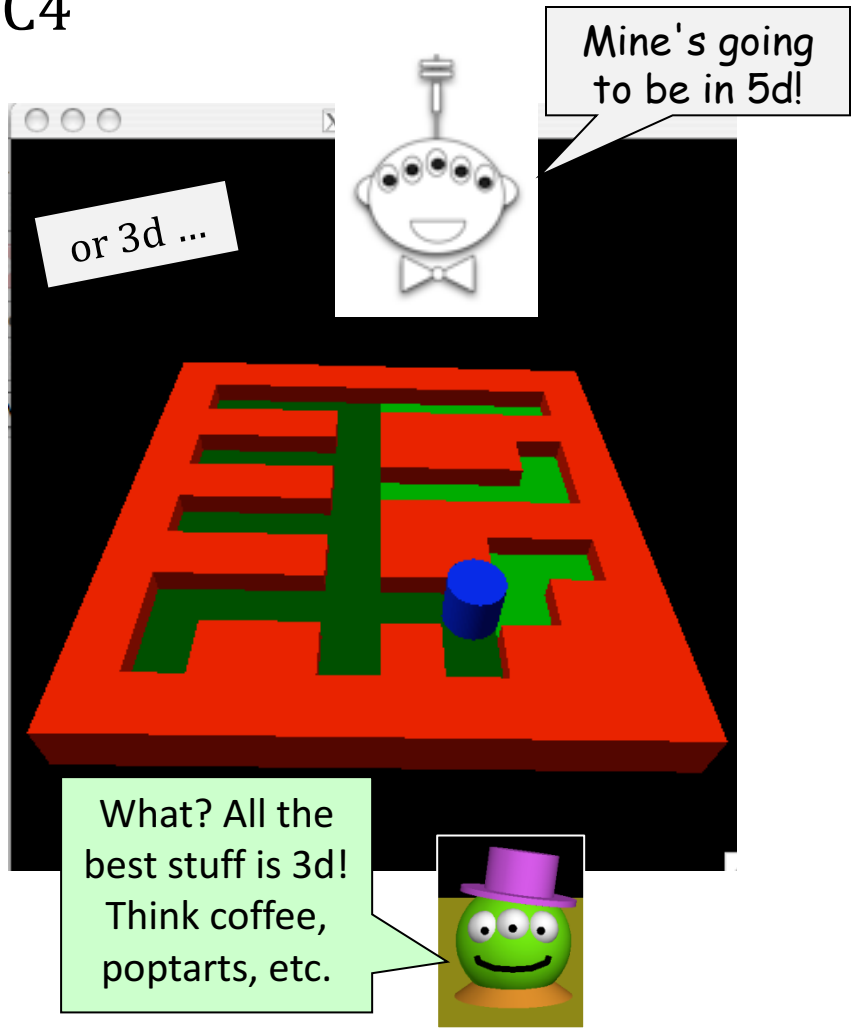


Extra: Picobot graphics…

Choice of graphical packages

- 2d: graphics.py from C4
- 3d: VPython





Project ***space***...

