Easy AI with Python

Raymond Hettinger

PyCon 2009

Themes

- Easy!
- Short!
- General purpose tools – easily adaptable
- Great for teaching Python
- Hook young minds with truly interesting problems.

Topics:

1. Exhaustive search using new itertools
2. Database mining using neural nets
3. Automated categorization with a naive Bayesian classifier
4. Solving popular puzzles with depth-first and breath-first searches
5. Solving more complex puzzles with constraint propagation
6. Play a popular game using a probing search strategy

Eight Queens – Six Lines

http://code.activestate.com/recipes/576647/

```
from itertools import permutations

n = 8
cols = range(n)
for vec in permutations(cols):
    if (n == len(set(vec[i]+i for i in cols))
          == len(set(vec[i]-i for i in cols))):
        print vec
```

Alphametics Solver

```
>>> solve('SEND + MORE == MONEY')
9567 + 1085 == 10652
```

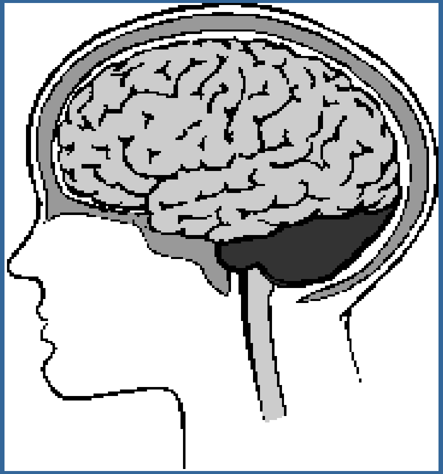
- 'VIOLIN * 2 + VIOLA == TRIO + SONATA',
- 'SEND + A + TAD + MORE == MONEY',
- 'ZEROES + ONES == BINARY',
- 'DCLIZ + DLXVI == MCCXXV',
- 'COUPLE + COUPLE == QUARTET',
- 'FISH + N + CHIPS == SUPPER',
- 'SATURN + URANUS + NEPTUNE + PLUTO == PLANETS',
- 'EARTH + AIR + FIRE + WATER == NATURE',
- ('AN + ACCELERATING + INFERENTIAL + ENGINEERING + TALE + ' +
-     'ELITE + GRANT + FEE + ET + CETERA == ARTIFICIAL + INTELLIGENCE'),
- 'TWO * TWO == SQUARE',
- 'HIP * HIP == HURRAY',
- 'PI * R ** 2 == AREA',
- 'NORTH / SOUTH == EAST / WEST',
- 'NAUGHT ** 2 == ZERO ** 3',
- 'I + THINK + IT + BE + THINE == INDEED',
- 'DO + YOU + FEEL == LUCKY',
- 'NOW + WE + KNOW + THE == TRUTH',
- 'SORRY + TO + BE + A + PARTY == POOPER',
- 'SORRY + TO + BUST + YOUR == BUBBLE',
- 'STEEL + BELTED == RADIALS',
- 'ABRA + CADABRA + ABRA + CADABRA == HOUDINI',
- 'I + GUESS + THE + TRUTH == HURTS',
- 'LETS + CUT + TO + THE == CHASE',
- 'THATS + THE + THEORY == ANYWAY',

Alphametics Solver – Recipe 576647

```
def solve(s):
    words = findall('[A-Za-z]+', s)
    chars = set(''.join(words))
    assert len(chars) <= 10
    firsts = set(w[0] for w in words)
    chars = ''.join(firsts) + ''.join(chars - firsts)
    n = len(firsts)
    for perm in permutations('0123456789', len(chars)):
        if '0' not in perm[:n]:
            trans = maketrans(chars, ''.join(perm))
            equation = s.translate(trans)
            if eval(equation):
                print equation
```

Neural Nets for Data-Mining

- ASPN Cookbook Recipe: 496908
- Parallel Distributed Processing: IAC example



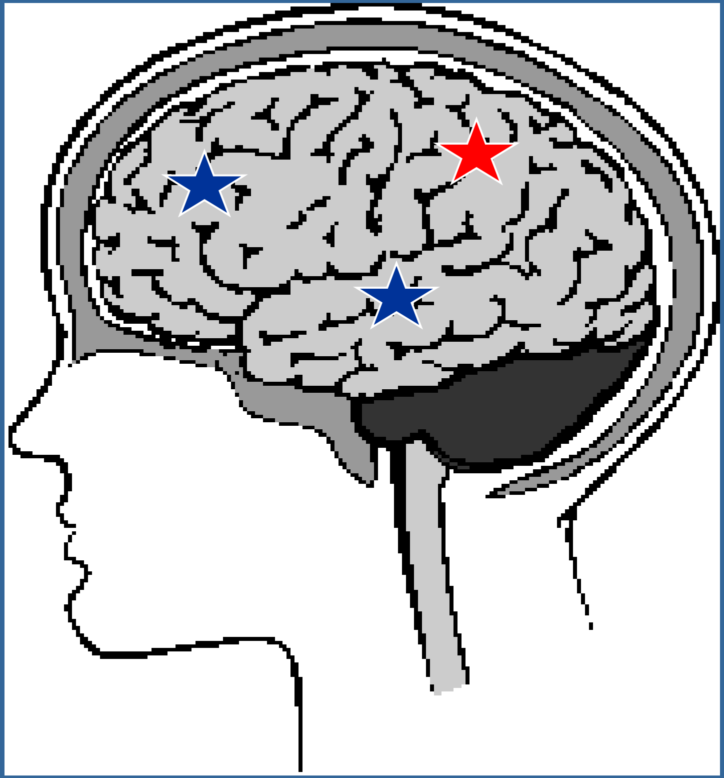
What is the core concept?

- A database can be modeled as a brain (neural net)
- Unique field values in the database are neurons
- Table rows define mutually excitory connections
- Table columns define competing inhibitory connections

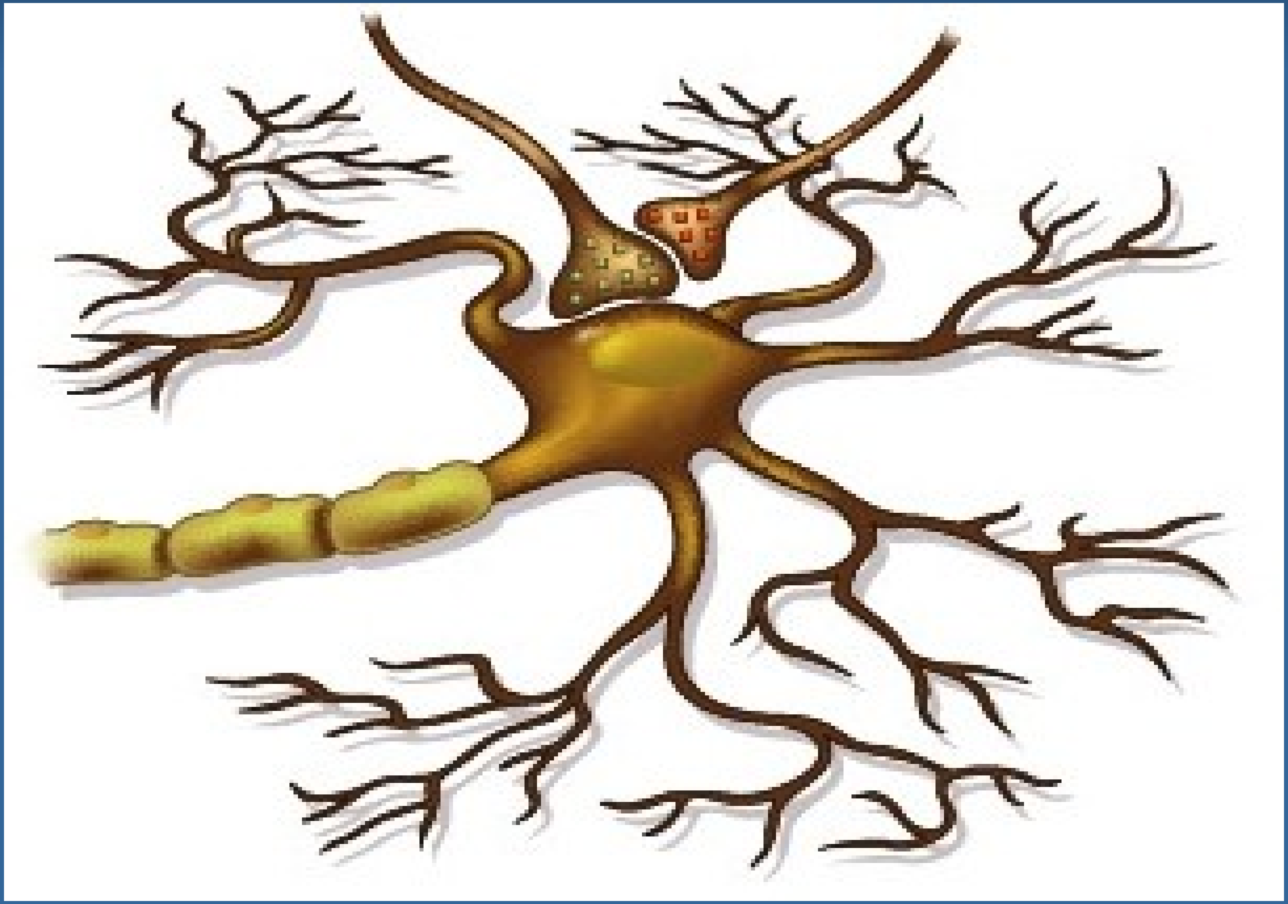
How do you use it?

- Provide a stimulus to parts of the neural net
- Then see which neurons get activated the most

The Human Brain



Neurons, Axons, Synapses



What can this do that SQL can't?

- Make generalizations
- Survive missing data
- Extrapolate to unknown instances

Jets and Sharks example

Art	Jets	40	jh	sing	pusher
Al	Jets	30	jh	mar	burglar
Sam	Jets	20	col	sing	bookie
Clyde	Jets	40	jh	sing	bookie
Mike	Jets	30	jh	sing	bookie
. . .					
Earl	Sharks	40	hs	mar	burglar
Rick	Sharks	30	hs	div	burglar
Ol	Sharks	30	col	mar	pusher
Neal	Sharks	30	hs	sing	bookie
Dave	Sharks	30	hs	div	pusher

Neuron for Every unique value in the database

```
Art, Al, Sam, Jets, Sharks, 20, 30, 40,
   pusher, bookie ...
```

- All neurons start in a resting state
- Excited by neural connections – defined by the table rows
- Inhibited by other neurons in the same pool – defined by table columns
- Can also be excited or inhibited externally – probes into the database

Generalizing from a specific instance

- touch('Ken', weight=0.8)

```
Ken: 0.82 Nick: 0.23     Neal: 0.23
  Rick: 0.03
Earl: 0.03     Pete: 0.03     Fred: 0.03

Sharks: 0.53   Jets: -0.13
20: 0.41   30: -0.05 40: -0.13
hs: 0.54   jh: -0.14 col: -0.14
sing: 0.53     mar: -0.13     div: -0.13

burglar: 0.41  pusher: -0.11  bookie: -0.11
```

Query the neural net for given facts

- touch('Sharks 20 jh sing burglar')

```
Ken: 0.54 Lance: 0.47    John: 0.47     Jim:
   0.47
George: 0.47
Sharks: 0.78   Jets: 0.48
20: 0.85  30: -0.15 40: -0.15
jh: 0.84  hs: -0.12 col: -0.15
sing: 0.80     mar: 0.02 div: 0.02
burglar: 0.85  bookie: -0.15  pusher: -0.15
```

Compensate for missing data

- touch('Lance')
- depair('Lance','burglar')

```
Lance: 0.82    John: 0.54     Jim: 0.30 George:
  0.30
Al: 0.26
Jets: 0.66     Sharks: -0.14
20: 0.63  30: -0.13 40: -0.14
jh: 0.66  hs: -0.14 col: -0.14
mar: 0.58 div: -0.08     sing: -0.14
burglar: 0.54  pusher: -0.14  bookie: -0.14
```

Neuron class

```
class Unit(object):
    def __init__(self, name, pool):
        self.name = name
        self.pool = pool
        self.reset()
        self.exciters = []
        unitbyname[name] = self
    def computenewact(self):
        ai = self.activation
        plus = sum(exciter.output for exciter in self.exciters)
        minus = self.pool.sum - self.output
        netinput = alpha*plus - gamma*minus + estr*self.extinp
        if netinput > 0:
            ai = (maxact-ai)*netinput - decay*(ai-rest) + ai
        else:
            ai = (ai-minact)*netinput - decay*(ai-rest) + ai
        self.newact = max(min(ai, maxact), minact)
```

Pool class

```
class Pool(object):
    __slots__ = ['sum', 'members']
    def __init__(self):
        self.sum = 0.0
        self.members = set()
    def addmember(self, member):
        self.members.add(member)
    def updatesum(self):
        self.sum = sum(member.output for member in self.members)
```

Engine

```
def run(times=100):
    """Run n-cycles and display result"""
    for i in xrange(times):
        for pool in pools:
            pool.updatesum()
        for unit in units:
            unit.computenewact()
        for unit in units:
            unit.commitnewact()
    print '-' * 20
    for pool in pools:
        pool.display()
```

What have we accomplished:

- 100 lines of Python models any database as neural net
- Probing the brain reveals or confirms hidden relationships

Mastermind

- Mastermind-style Games
- Making smart probes into a search space
- ASPN Cookbook Recipe: 496907

What is Mastermind?

- A codemaker picks a 4-digit code: (4, 3, 3, 7)
- The codebreaker makes a guess: (3, 3, 2, 4)
- The codemaker scores the guess: (1, 2)
- The codebreaker uses the information to make better guesses
- There are many possible codebreaking strategies
- Better strategies mean that fewer guesses are needed

What is the interesting part?

- Experimenting with different strategies to find the best
- Finding ways to make it fast

The basic code takes only 30 lines:

```
import random
from itertools import izip, imap
digits = 4
fmt = '%0' + str(digits) + 'd'
searchspace = tuple([tuple(map(int,fmt % i)) for i in
   range(0,10**digits)])
def compare(a, b, map=imap, sum=sum, zip=izip, min=min):
    count1 = [0] * 10
    count2 = [0] * 10
    strikes = 0
    for dig1, dig2 in zip(a,b):
        if dig1 == dig2:
            strikes += 1
        count1[dig1] += 1
        count2[dig2] += 1
    balls = sum(map(min, count1, count2)) - strikes
    return (strikes, balls)
```

(Continued)

```
def rungame(target, strategy, maxtries=15):
    possibles = list(searchspace)
    for i in xrange(maxtries):
        g = strategy(i, possibles)
        print "Out of %7d possibilities.  I'll guess %r" % (len(possibles), g),
        score = compare(g, target)
        print ' ---> ', score
        if score[0] == digits:
            print "That's it.  After %d tries, I won." % (i+1,)
            break
        possibles = [n for n in possibles if compare(g, n) == score]
    return i+1
```

(Continued)

```
def s_allrand(i, possibles):
    'Simple strategy that randomly chooses one remaining possibility'
    return random.choice(possibles)



hiddencode = (4, 3, 3, 7)
rungame(hiddencode, s_allrand)
```

Step 1: List out the search space

```
digits = 4

fmt = '%0' + str(digits) + 'd‘

searchspace = tuple([tuple(map(int,fmt % i)) for i in range(0,10**digits)])
```

This creates a sequence of all possible plays:

(0, 0, 0, 0),
(0, 0, 0, 1),
(0, 0, 0, 2),
. . .
(9, 9, 9, 9),

Step 2: Build the scoring function

```
def compare(a, b, map=imap, sum=sum, zip=izip,
   min=min):
     count1 = [0] * 10
     count2 = [0] * 10
     strikes = 0
     for dig1, dig2 in zip(a,b):
          if dig1 == dig2:
               strikes += 1
          count1[dig1] += 1
          count2[dig2] += 1
     balls = sum(map(min, count1, count2)) - strikes
     return (strikes, balls)
```

Step 3: Devise a strategy for choosing the next move

```
def s_allrand(i, possibles):
    'Simple strategy that randomly chooses one possibility'
    return random.choice(possibles)
```

Step 4: Make an engine to run the game

- Start with a full search space
- Let the strategy pick a probe
- Score the result
- Pare down the search space using the score

Step 5:   Run it!

```
hiddencode = (4, 3, 3, 7)
rungame(hiddencode, s_allrand)
```

- Out of 10000 possibilities. I'll guess (0, 0, 9, 3) ---> (0, 1)
- Out of 3052 possibilities. I'll guess (9, 4, 4, 8) ---> (0, 1)
- Out of 822 possibilities. I'll guess (8, 3, 8, 5) ---> (1, 0)
- Out of 123 possibilities. I'll guess (3, 3, 2, 4) ---> (1, 2)
- Out of 6 possibilities. I'll guess (4, 3, 3, 6) ---> (3, 0)
- Out of 2 possibilities. I'll guess (4, 3, 3, 1) ---> (3, 0)
- Out of 1 possibilities. I'll guess (4, 3, 3, 7) ---> (4, 0)
- That's it. After 7 tries, I won.

Step 6: Make it fast

- psyco gives a 10:1 speedup
- code the compare() function in C

Step 7: Experiment with a new strategy

- If possible, make a guess with no duplicates

```
def s_trynodup(i, possibles):
    for j in xrange(20):
        g = random.choice(possibles)
        if len(set(g)) == digits:
            break
    return g
```

Step 8: Try a smarter strategy:

- The utility of a guess is how well it divides-up the remaining possibilities

```
def utility(play, possibles):
    b = {}
    for poss in possibles:
        score = compare(play, poss)
        b[score] = b.get(score, 0) + 1
    return info(b.values())
```

(Continued)

- The information content of that division is given by Shannon’s formula

```
def info(seqn):
    bits = 0
    s = float(sum(seqn))
    for i in seqn:
        p = i / s
        bits -= p * log(p, 2)
    return bits
```

(Continued)

- Select the probe with the greatest information content

```
def s_bestinfo(i, possibles):
    if i == 0:
        return s_trynodup(i, possibles)
    plays = random.sample(possibles, min(20, len(possibles)))
    _, play = max([(utility(play, possibles), play) for play in plays])
    return play
```

Step 9: Make it fast

- Instead of evaluating the utility of every move, pick the best of a small sample:

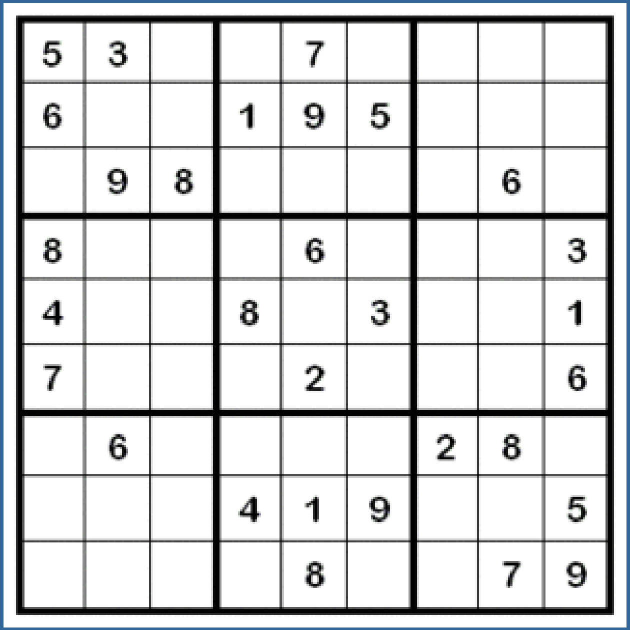
```
def s_samplebest(i, possibles):
    if i == 0:
        return s_trynodup(i, possibles)
    if len(possibles) > 150:
        possibles = random.sample(possibles, 150)
        plays = possibles[:20]
    elif len(possibles) > 20:
        plays = random.sample(possibles, 20)
    else:
        plays = possibles
    _, play = max([(utility(play, possibles), play) for play in plays])
    return play
```

Step 10: Bask in the glow of your model:

- The basic framework took only 30 lines of code
- Four different strategies took another 30
- It’s easy to try out even more strategies
- The end result is not far from the theoretical optimum

Sudoku-style Puzzles

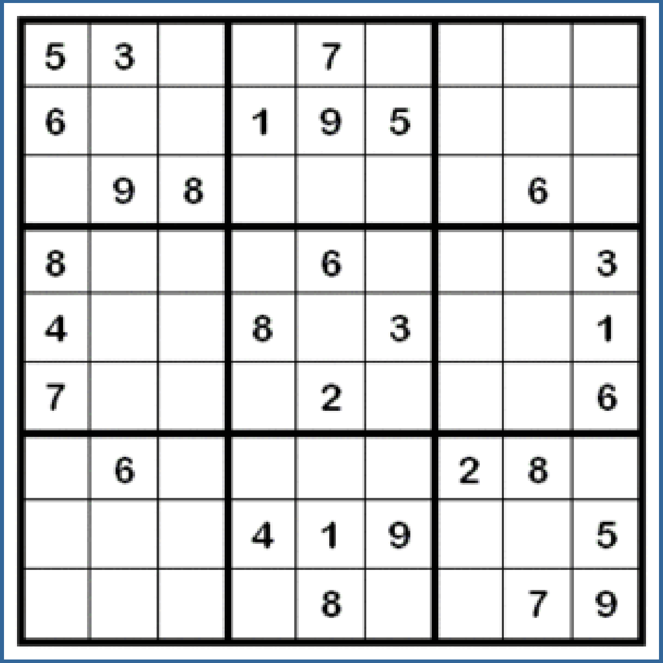
- An exercise in constraint propagation
- ASPN Cookbook Recipe
- Google for: sudoku norvig
- Wikipedia entry: sudoku



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

What does a Sudoku puzzle look like?

```
27 | 15|  8
   |3  |7 4
   | 7 |
---+---+---
 5 |1  | 7
  9|   |2
 6 |  2| 5
---+---+---
   | 8 |
6 5|  4|
8  |59 | 41
```



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

What does a Sudoku puzzle look like when it is solved

```
276|415|938
581|329|764
934|876|512
---+---+---
352|168|479
149|753|286
768|942|153
---+---+---
497|681|325
615|234|897
823|597|641
```

What is the interesting part?

- Use constraints to pare-down an enormous search-space
- Finding various ways to propagate constraints
- Enjoy the intellectual challenge of the puzzles without wasting time
- Complete code takes only 56 lines (including extensive comments)

Step 1: Choose a representation and a way to display it

```
'53  7    6  195    98    6 8   6   34  8 3  17   2   6 6
  28    419  5    8  79‘

def show(flatline):
    'Display grid from a string (values in row major order
  with blanks for unknowns)'
    fmt = '|'.join(['%s' * n] * n)
    sep = '+'.join(['-'  * n] * n)
    for i in range(n):
        for j in range(n):
            offset = (i*n+j)*n2
            print fmt % tuple(flatline[offset:offset+n2])
        if i != n-1:
            print sep
```

Step 2: Determine which cells are in contact with each other

```
def _find_friends(cell):
 'Return tuple of cells in same row, column, or subgroup'
    friends = set()
    row, col = cell // n2, cell % n2
    friends.update(row * n2 + i for i in range(n2))
    friends.update(i * n2 + col for i in range(n2))
    nw_corner = row // n * n3 + col // n * n
    friends.update(nw_corner + i + j for i in range(n) for j in
   range(0,n3,n2))
    friends.remove(cell)
    return tuple(friends)
friend_cells = map(_find_friends, range(n4))
```

Step 3: Write a solver

```python
def solve(possibles, pending_marks):
    # Apply pending_marks (list of cell,value pairs) to possibles (list of str).
    # Mutates both inputs.  Return solution as a flat string (values in row-major order)
    # or return None for dead-ends where all possibilites have been eliminated.
    for cell, v in pending_marks:
        possibles[cell] = v
        for f in friend_cells[cell]:
            p = possibles[f]
            if v in p:
                p = possibles[f] = p.replace(v, '')     # exclude value v from friend f
                if not p:
                    return None                  # Dead-end:  all possibilities eliminated
                if len(p) == 1:
                    pending_marks.append((f, p[0]))
```

(Continued)

```
# Check to see if the puzzle is fully solved (each cell has only one
   possible value)
if max(map(len, possibles)) == 1:
        return ''.join(possibles)

# If it gets here, there are still unsolved cells
cell = select_an_unsolved_cell(possibles)
for v in possibles[cell]:   # try all possible values for that cell
    ans = solve(possibles[:], [(cell, v)])
    if ans is not None:
        return ans
```

Hey, what did that solver do again?

- Makes an assumption about a cell
- Goes to each of the friend_cells and eliminates that possibility
- Check to see if the puzzle is solved
- If some cell goes down to one possibility, repeat the process
- If some cells are unsolved, make another assumption

Step 4: Pick a search strategy:

```
def select_an_unsolved_cell(possibles, heuristic=min):
    # Default heuristic:  select cell with fewest possibilities
    # Other possible heuristics include:  random.choice() and max()
    return heuristic((len(p), cell) for cell, p in
  enumerate(possibles) if len(p)>1)[1]
```

Here's where it gets fun.
Step 5: Bask in the glow of your model:

- The basic framework took only 56 lines of code
- It's easy to try-out alternative search heuristics
- The end result solves "hard" puzzles with very little guesswork
- It's easy to generate all possible solutions
- The framework can be extended for other ways to propagate constraints
- See the wikipedia entry for other solution techniques

Bayesian Classifier

- Google for: reverend thomas python
- Or link to: http://www.divmod.org /projects/reverend

Principle

- Use training data (a corpus) to compute conditional probabilities
- Given some attribute, what is the conditional probability of a given classification
- Now, given many attributes, combine those probabilities
- Done right, you have to know joint probabilities
- But if you ignore those, it tends to work out just fine

Example

- Guess which decade a person was born
- Attribute 1 – the person’s name is Gertrude
- Attribute 2 – their favorite dance is the Foxtrot
- Attribute 3 – the person lives in the Suburbs
- Attribute 4 – the person drives a Buick Regal

- We don’t know the joint probabilities, but can gather statistics on each on by itself.

Tricks of the trade

- Since we’ve thrown exact computation away, what is the best approximation
- The simplest way is to multiply the conditional probabilities
- The conditionals can be biased by a count of one to avoid multiplying by zero
- The information content can be estimated using the Claude Shannon formula
- The probabilities can be weighted by significance
- and so on . . .

Coding it in Python

- Conceptually simple
- Just count attributes in the corpus to compute the conditional probabilities
- Just multiply (or whatever) the results for each attribute
- Pick the highest probability result
- Complexity comes from effort to identify attributes (parsing, tokenizing, etc)

Language classification example

```
>>> from reverend.thomas import Bayes
>>> guesser = Bayes()
>>> guesser.train('french', 'le la les du un une je il
   elle de en')
>>> guesser.train('german', 'der die das ein eine')
>>> guesser.train('spanish', 'el uno una las de la en')
>>> guesser.train('english', 'the it she he they them
   are were to')
>>> guesser.guess('they went to el cantina')spanish
>>> guesser.guess('they were flying planes')english
```

The hot topic

- Classifying email as spam or ham
- Simple example using reverend.thomas
- Real-world example with spambayes

Generic Puzzle Solving Framework Depth-first and breadth-first tree searches

- Link to: http://users.rcn.com/python/download/puzzle.py
- Google for: puzzle hettinger
- Wikipedia: depth-first search

What does a generic puzzle look like?

- There an initial position
- There is a rule for generating legal moves
- There is a test for whether a state is a goal
- Optionally, there is a way to pretty print the current state

Initial position for the Golf-Tee Puzzle

```
        0
      1   1
    1   1   1
  1   1   1   1
1   1   1   1   1
```

Legal move

Jump over an adjacent tee

and removing the jumped tee

```
        1
      0   1
    0   1   1
  1   1   1   1
1   1   1   1   1
```

Goal state

```
        1
      0   0
    0   0   0
  0   0   0   0
0   0   0   0   0
```

Code for the puzzle

```
class GolfTee( Puzzle ):
        pos = '011111111111111'
        goal = '100000000000000'
        triples = [[0,1,3], [1,3,6], [3,6,10], [2,4,7], [4,7,11], [5,8,12],
                   [10,11,12], [11,12,13], [12,13,14], [6,7,8], [7,8,9], [3,4,5],
                   [0,2,5], [2,5,9], [5,9,14], [1,4,8], [4,8,13], [3,7,12]]
        def __iter__( self ):
            for t in self.triples:
                if self.pos[t[0]]=='1' and self.pos[t[1]]=='1' and self.pos[t[2]]=='0':
                    yield TriPuzzle(self.produce(t,'001'))
                if self.pos[t[0]]=='0' and self.pos[t[1]]=='1' and self.pos[t[2]]=='1':
                    yield TriPuzzle(self.produce(t,'100'))
        def produce( self, t, sub ):
            return self.pos[:t[0]] + sub[0] + self.pos[t[0]+1:t[1]] + \
                   sub[1] + self.pos[t[1]+1:t[2]] + sub[2] + self.pos[t[2]+1:]
        def canonical( self ):
            return self.pos
        def __repr__( self ):
            return '\n        %s\n      %s   %s\n    %s   %s   %s\n  %s   %s   %s
    %s\n%s   %s   %s   %s   %s\n' % tuple(self.pos)
```

How does the solver work?

- Start at the initial position
- Generate legal moves
- Check each to see if it is a goal state
- If not, then generate the next legal moves and repeat

Code for the solver

```
def solve( pos, depthFirst=0 ):
        queue, trail = deque([pos]), {pos.canonical():None}
        solution = deque()
        load = depthFirst and queue.append or queue.appendleft
        while not pos.isgoal():
            for m in pos:
                c = m.canonical()
                if c in trail: continue
                trail[c] = pos
                load(m)
            pos = queue.pop()
        while pos:
            solution.appendleft( pos)
            pos = trail[pos.canonical()]
        return solution
        0
      1   1
    1   1   1
  1   1   1   1
1   1   1   1   1
```

1

0 1

0 1 1

1 1 1 1

1 1 1 1 1

```
        1
      0   1
    1   0   0
  1   1   1   1
1   1   1   1   1
```

```
        1
      1   1
    0   0   0
  0   1   1   1
1   1   1   1   1
```

```
        0
      0   1
    1   0   0
  0   1   1   1
1   1   1   1   1
```

```
        0
      0   1
    1   1   0
  0   0   1   1
1   0   1   1   1
```

```
        0
      0   1
    1   1   0
  0   0   1   1
1   1   0   0   1
```

```
        0
      0   1
    1   1   0
  0   0   1   1
0   0   1   0   1
```

```
        0
      0   1
    1   1   1
  0   0   1   0
0   0   1   0   0
```

```
        0
      0   0
    1   1   0
  0   0   1   1
0   0   1   0   0
```

```
        0
      0   0
    1   1   1
  0   0   0   1
0   0   0   0   0
```

0

0 1

1 1 0

0 0 0 0

0 0 0 0 0

0

0 1

0 0 1

0 0 0 0

0 0 0 0 0

```
        1
      0   0
    0   0   0
  0   0   0   0
0   0   0   0   0
```

How about Jug Filling Puzzle

Given a two empty jugs with 3 and 5 liter capacities and a full jug with 8 liters, find a sequence of pours leaving four liters in the two largest jugs

What does the code look like?

```
class JugFill( Puzzle ):
    pos = (0,0,8)
    capacity = (3,5,8)
    goal = (0,4,4)
    def __iter__(self):
       for i in range(len(self.pos)):
           for j in range(len(self.pos)):
               if i==j: continue
               qty = min(self.pos[i], self.capacity[j] - self.pos[j])
               if not qty: continue
               dup = list( self.pos )
               dup[i] -= qty
               dup[j] += qty
               yield JugFill(tuple(dup))
```

What does the solution look like:

(0, 0, 8)	
(0, 5, 3)	Pour the 8 into the 5 until it is full leaving 3
(3, 2, 3)	Pour the 5 into the 2 until it is full leaving 2
(0, 2, 6)	Pour the 3 into the other the totaling 6
(2, 0, 6)	Pour the 2 into the empty jug
(2, 5, 1)	Pour the 6 into the 5 leaving 1
(3, 4, 1)	Pour the 5 into the 3 until full leaving 4
(0, 4, 4)	Pour the 1 into the 3 leaving 4

How about a black/white marble jumping puzzle?

Given eleven slots in a line with four white marbles in the leftmost slots and four black marbles in the rightmost, make moves to put the white ones on the right and the black on the left. A valid move for a while marble isto shift right into an empty space or hop over a single adjacent black marble into an adjacent empty space -- don't hop over your own color, don't hop into an occupied space, don't hop over more than one marble. The valid black moves are in the opposite direction. Alternate moves between black and white marbles.

(Continued)

In the tuple representation below, zeros are open holes, ones are whites, negative ones are blacks, and the outer tuple track whether it is whites move or blacks.

The code is straight-forward:

```
pos = (1,(1,1,1,1,0,0,0,-1,-1,-1,-1))

goal =  (-1,-1,-1,-1,0,0,0,1,1,1,1)

def isgoal( self ):
    return self.pos[1] == self.goal

def __iter__( self ):
    (m,b) = self.pos
    for i in range(len(b)):
        if b[i] != m: continue
        if 0<=i+m+m<len(b) and b[i+m] == 0:
            newmove = list(b)
            newmove[i] = 0
            newmove[i+m] = m
            yield MarblePuzzle((-m,tuple(newmove)))
        elif 0<=i+m+m<len(b) and b[i+m]==-m and b[i+m+m]==0:
            newmove = list(b)
            newmove[i] = 0
            newmove[i+m+m] = m
            yield MarblePuzzle((-m,tuple(newmove)))
```

What does the solution look like?

[wwww...bbbb, wwww..b.bbb, www.w.b.bbb, www.wb..bbb, ww.wwb..bbb, ww.wwb.b.bb, ww.w.bwb.bb, ww.wb.wb.bb, ww..bwwb.bb, ww.b.wwb.bb, ww.b.w.bwbb, wwb..w.bwbb, w.bw.w.bwbb, w.bw.wb.wbb, .wbw.wb.wbb, bw.w.wb.wbb, b.ww.wb.wbb, b.wwbw..wbb, b.wwb.w.wbb, b.wwb.wbw.b, b.w.bwwbw.b, b.w.bwwbwb., b.w.bwwb.bw, b.wb.wwb.bw, b.wb.w.bwbw, bbw..w.bwbw, bbw...wbwbw, bbw..bw.wbw, bb.w.bw.wbw, bb.w.bwbw.w, bb..wbwbw.w, bb.bw.wbw.w, bb.bw.wb.ww, bbb.w.wb.ww, bbb.w..bwww, bbb.w.b.www, bbb..wb.www, bbb.bw..www, bbb.b.w.www, bbbb..w.www,

How about a sliding block puzzle?

```
1122
1133
45..
6788
6799
```

(Continued)

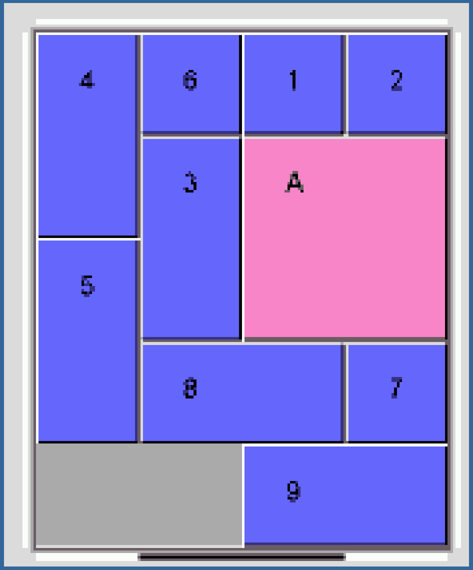
7633

7622

..54

1199

1188



Sliding Block Code

```
class Sliding( Puzzle ):
       pos = '11221133450067886799'
       goal = re.compile( r'................1...' )
       def isgoal(self):
           return self.goal.search(self.pos) != None
       def __repr__( self ):
           ans = '\n'
           pos = self.pos.replace( '0', '.' )
           for i in [0,4,8,12,16]:
               ans = ans + pos[i:i+4] + '\n'
           return ans
       xlat = string.maketrans('38975','22264')
       def canonical( self ):
           return self.pos.translate( self.xlat )
       block = { (0,-4):None, (1,-4):None, (2,-4):None, (3,-4):None,
                 (16,4):None, (17,4):None, (18,4):None, (19,4):None,
                 (0,-1):None, (4,-1):None, (8,-1):None, (12,-1):None, (16,-1):None,
                 (3,1):None, (7,1):None, (11,1):None, (15,1):None, (19,1):None, }
```

(Continued)

```
def __iter__( self ):
            dsone = self.pos.find('0')
            dstwo = self.pos.find('0',dsone+1)
            for dest in [dsone, dstwo]:
                for adj in [-4,-1,1,4]:
                    if (dest,adj) in self.block: continue
                    piece = self.pos[dest+adj]
                    if piece == '0': continue
                    newmove = self.pos.replace(piece, '0')
                    for i in range(20):
                        if 0 <= i+adj < 20 and self.pos[i+adj]==piece:
                            newmove = newmove[:i] + piece + newmove[i+1:]
                    if newmove.count('0') != 2: continue
                    yield PaPuzzle(newmove)
```

What have we accomplished?

- A 36 line generic puzzle solving framework
- Easy adapted to a huge variety of puzzles
- The fun part is writing the move generator
- Everything else is trivial (initial position, goal state, repr function)

Optimizing

- Fold symmetries into a single canonical states (don’t explore mirror image solutions)
- Use collections.deque() instead of a list()
- Decide between depth-first and breadth-first solutions. (first encountered vs shortest solution)

Q & A