shape poker

Thomas Kagan

goal

- Create an AI to self-learn through play

goal

Minecraft?

goal

Minecraft?

goal

Minecraft?

goal

Minecraft?

Minesweeper?

goal

Minecraft?

Minesweeper?

https://www.youtube.com/watch?v=cGUHehFGqBc

goal

Minecraft?

Minesweeper?

Dark Souls II?

goal

- Zero sum
- Imperfect information
- Easy to score

goal

- Zero sum
- Imperfect information
- Easy to score
- Computationally large

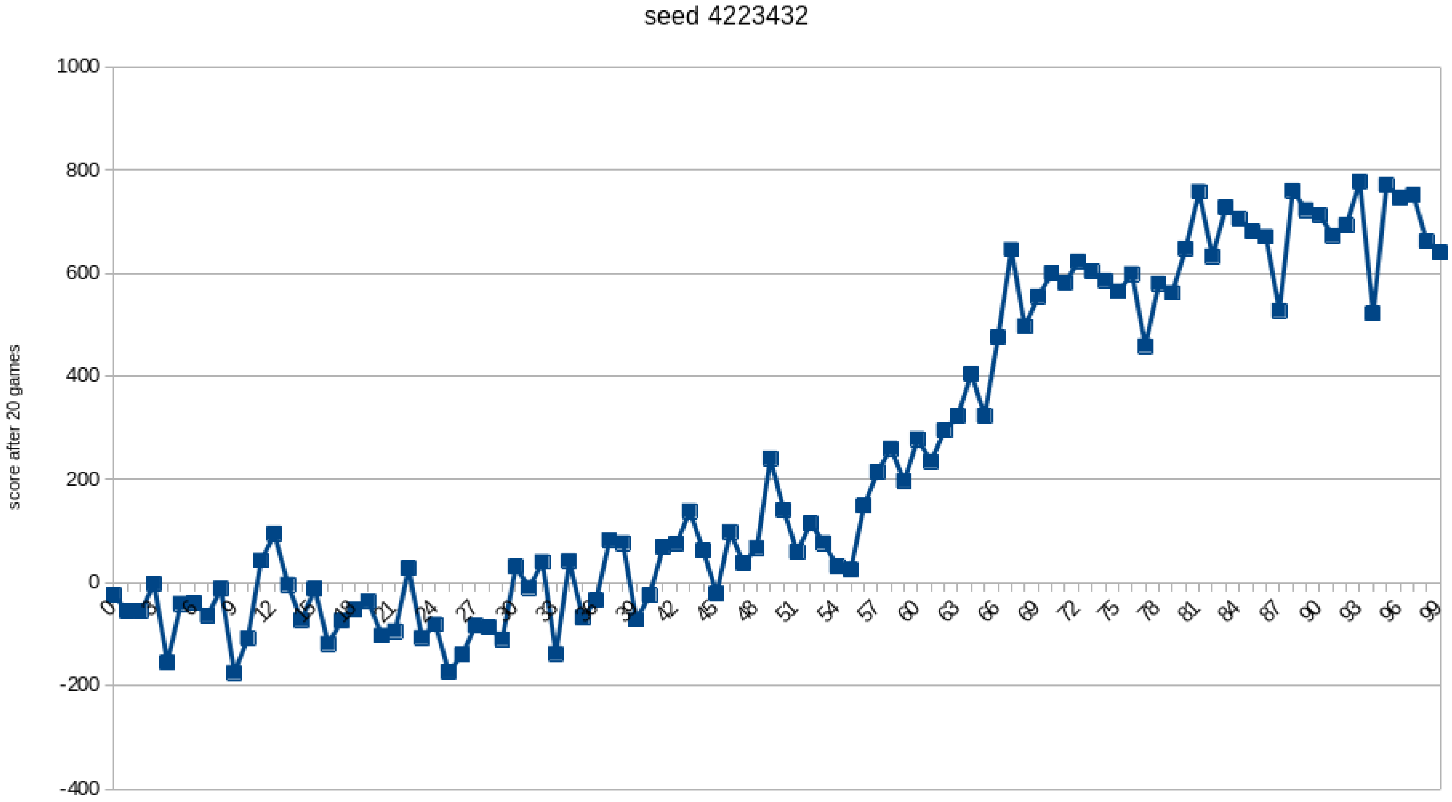
goal

- Create an AI to self-learn through playing *Shape Poker*

goal

- Create an AI to self-learn through playing *Shape Poker*
  - Study local minimums and how to avoid them
  - Study computational feasibility of stochastic methods

success!


seed 4223432
score after 20 games
1000
800
600
400
200
0
-200
-400
0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 75 78 81 84 87 90 93 96 99

project

- The game
- The player
- The learning algorithm

project

- The game – components, rules, simulation
- The player
- The learning algorithm

project

- The game – components, rules, simulation
- The player – perception, decision making
- The learning algorithm

project

- The game – components, rules, simulation
- The player – perception, decision making
- The learning algorithm – feedback, learning

project

- The game – components, rules, simulation
- The player – perception, decision making
- The learning algorithm – feedback, learning

*My contributions

project

- The game – components, rules, simulation
- The player – perception, decision making
- The learning algorithm – feedback, learning

game

seed 1512398
round 1 (P2 has earned 0)
deal, pot is 2
R: [triangle(red), circle(green)]
P1: [triangle(green), diamond(lavender)] (0)
P2: [circle(green), square(blue)] (0)
P1 raises 2
P2 raises 1
P1 calls
flop, pot is 10
R: [triangle(red), circle(green), square(blue)]
P1: [triangle(green), diamond(lavender)] (4)
P2: [circle(green), square(blue)] (1)
P1 folds
P2 wins!

round 2 (P2 has earned +6)
deal, pot is 2
R: [triangle(lavender), circle(blue)]
. . .

game

seed
 6734345
playing 10000 rounds
0.49328194979689616

seed
 9756767
playing 10000 rounds
0.6509583818701684

P1 Win %

seed
 4223432
playing 10000 rounds
0.45613470613470614

seed
 2498375
playing 10000 rounds
0.4293637505232315
. . .

- Either player is equally likely to win

game

seed
 6734345
playing 10000 rounds
0.49328194979689616

seed
 9756767
playing 10000 rounds
0.6509583818701684

seed
 4223432
playing 10000 rounds
0.45613470613470614

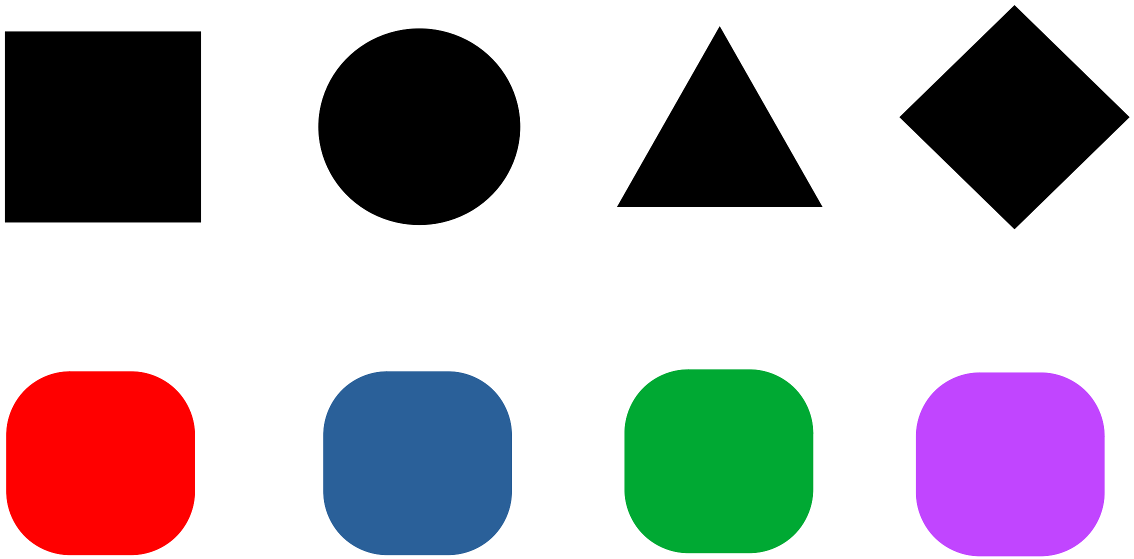
seed
 2498375
playing 10000 rounds
0.4293637505232315
. . .

- Either player equally likely to win, for any two random players
- Some players are objectively better than others

game

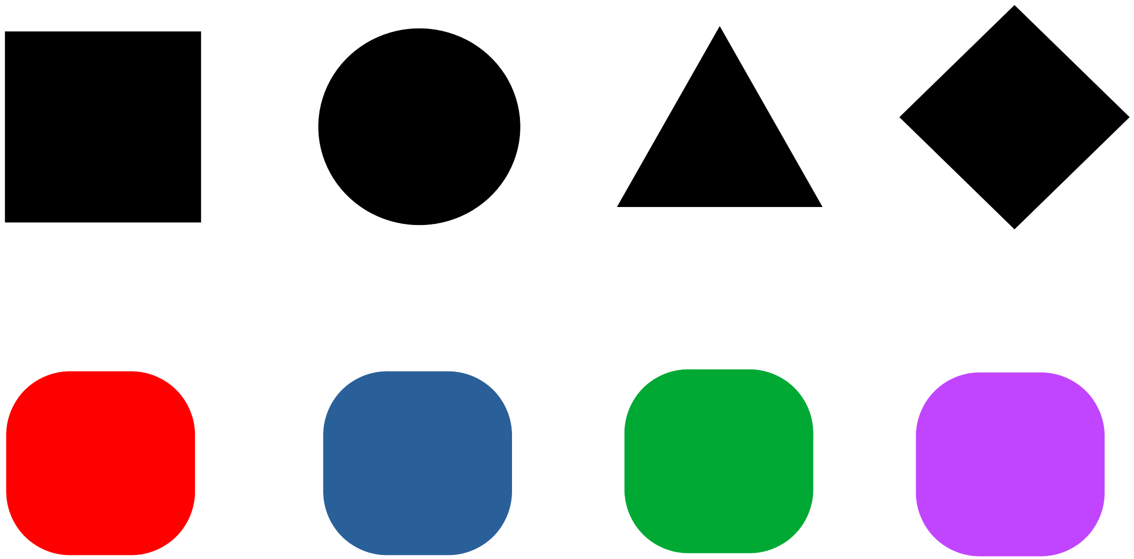
```
seed
 1512398
playing 10000 rounds
[(0, 0.36931642437364676), (1, 0.3244664398391587), (2, 0.21517682235281987), (3, 0.06485204660274255), (4, 0.026188266831632126)]

seed
 4963463
playing 10000 rounds
[(0, 0.252043112637688), (1, 0.34869122349875636), (2, 0.2618737415610565), (3, 0.09463460855146275), (4, 0.04275731375103636)]

seed
 1283218
playing 10000 rounds
[(0, 0.3471475271538543), (1, 0.339238637561953), (2, 0.22113255298956028), (3, 0.06485289465359063), (4, 0.027628387641041863)]

seed
 8374263
playing 10000 rounds
[(0, 0.2569620253164557), (1, 0.357307249712313), (2, 0.2518987341772152), (3, 0.09194476409666283), (4, 0.04188722669735328)]

seed
 6712362
playing 10000 rounds
[(0, 0.28474988933156264), (1, 0.34462151394422313), (2, 0.24369189907038513), (3, 0.08919876051350155), (4, 0.037737937140327575)]

seed
 4387523
playing 10000 rounds
[(0, 0.3286858807616273), (1, 0.34460513994381436), (2, 0.226303194256581), (3, 0.07137654770575383), (4, 0.029029237332223495)]

seed
 6734345
. . .
```

game



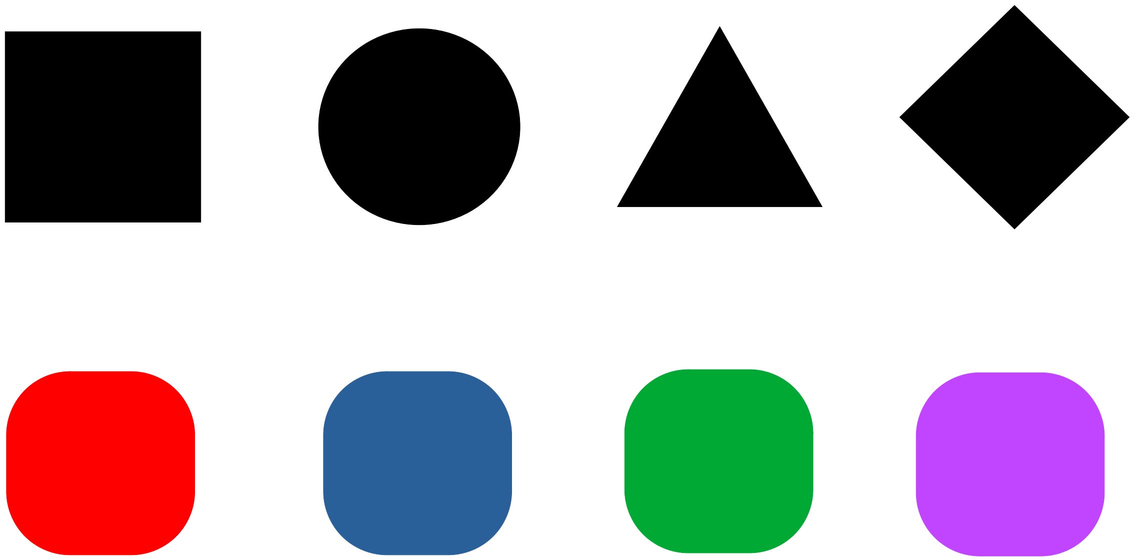
game

- 4 of each kind of card in the deck



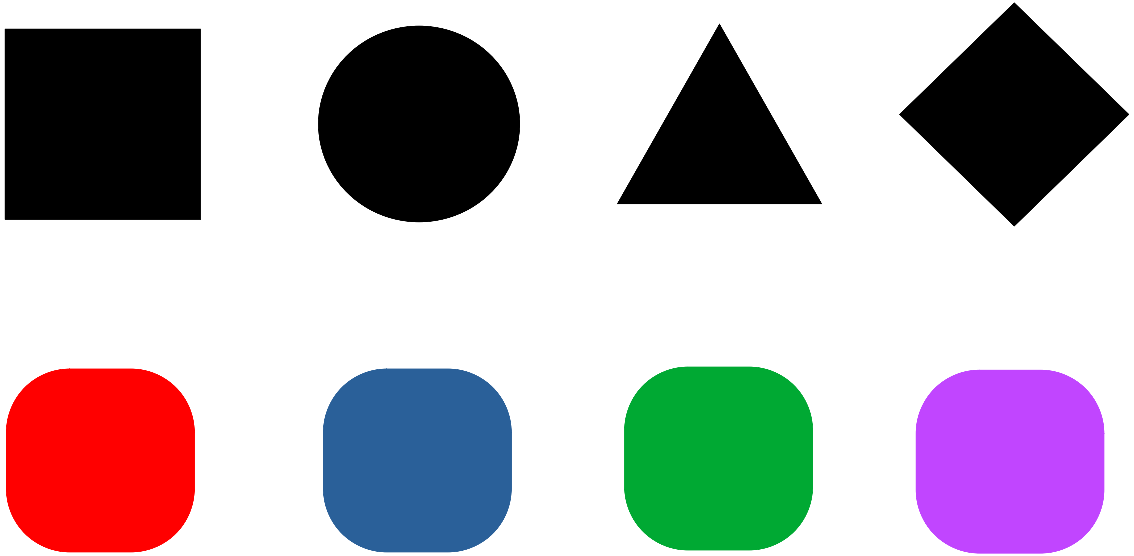
game

- 4 of each kind of card in the deck
- Each player has 2 cards in their hand



game

- 4 of each kind of card in the deck
- Each player has 2 cards in their hand
- 2-3 cards on table, “river”



game

```python
def score_hand(self, holdings):
    if len(holdings) != 4:
        return 0
    ret = 0
    cs = set([x.color for x in holdings])
    ss = set([x.shape for x in holdings])
    if len(ss) == 2: # three of a kind, or two pair
        ret += 1
    if len(ss) == 1: # four kind
        ret += 2
    if len(ss) == 4: # four row
        ret += 2
    if len(cs) == 1: #flush
        ret += 2
    if len(cs) == 4: #rainbow
        ret += 2

    return ret

def score(self, player):
    return max([self.score_hand(holdings) for holdings in combinations(player.hand + self.river, 4)])
```

game

```python
def score_hand(self, holdings):
    if len(holdings) != 4:
        return 0
    ret = 0
    cs = set([x.color for x in holdings])
    ss = set([x.shape for x in holdings])
    if len(ss) == 2: # three of a kind, or two pair
        ret += 1
    if len(ss) == 1: # four kind
        ret += 2
    if len(ss) == 4: # four row
        ret += 2
    if len(cs) == 1: #flush
        ret += 2
    if len(cs) == 4: #rainbow
        ret += 2

    return ret

def score(self, player):
    return max([self.score_hand(holdings) for holdings in combinations(player.hand + self.river, 4)])
```

hand+river -1

game

1. Deck is shuffled

game

1. Deck is shuffled
2. Each player draws 2 cards, two cards are place face up in the river

game

1. Deck is shuffled
2. Each player draws 2 cards, two cards are place face up in the river
3. P1 makes a bet. P2 can match it, or fold, or raise. P1 has to match a raise, or fold

game

1. Deck is shuffled
2. Each player draws 2 cards, two cards are place face up in the river
3. P1 makes a bet. P2 can match it, or fold, or raise. P1 has to match a raise, or fold
4. A third card is drawn from the deck to add to the river. Called “The Flop”

game

1. Deck is shuffled
2. Each player draws 2 cards, two cards are place face up in the river
3. P1 makes a bet. P2 can match it, or fold, or raise. P1 has to match a raise, or fold
4. A third card is drawn from the deck to add to the river. Called “The Flop”
5. Another round of betting ensues, in the same order

game

1. Deck is shuffled
2. Each player draws 2 cards, two cards are place face up in the river
3. P1 makes a bet. P2 can match it, or fold, or raise. P1 has to match a raise, or fold
4. A third card is drawn from the deck to add to the river. Called “The Flop”
5. Another round of betting ensues, in the same order
6. Scores are tallied, the winner takes the pot

game

1. Deck is shuffled
2. Each player draws 2 cards, two cards are place face up in the river
3. P1 makes a bet. P2 can match it, or fold, or raise. P1 has to match a raise, or fold
4. A third card is drawn from the deck to add to the river. Called “The Flop”
5. Another round of betting ensues, in the same order
6. Scores are tallied, the winner takes the pot
7. If there’s a tie, the pot carries over to the next round
8. Go to 1

project

- The game – components, rules, simulation ✔
- The player – perception, decision making
- The learning algorithm – feedback, learning

player

- “action weights”
- “environment”
- “strategy representations”
- “percept functions”

(,) =

player

(,) =

“action weights”

```
bet = random.choices(possible_bets, weights=w)[0]
```

(,) =

player

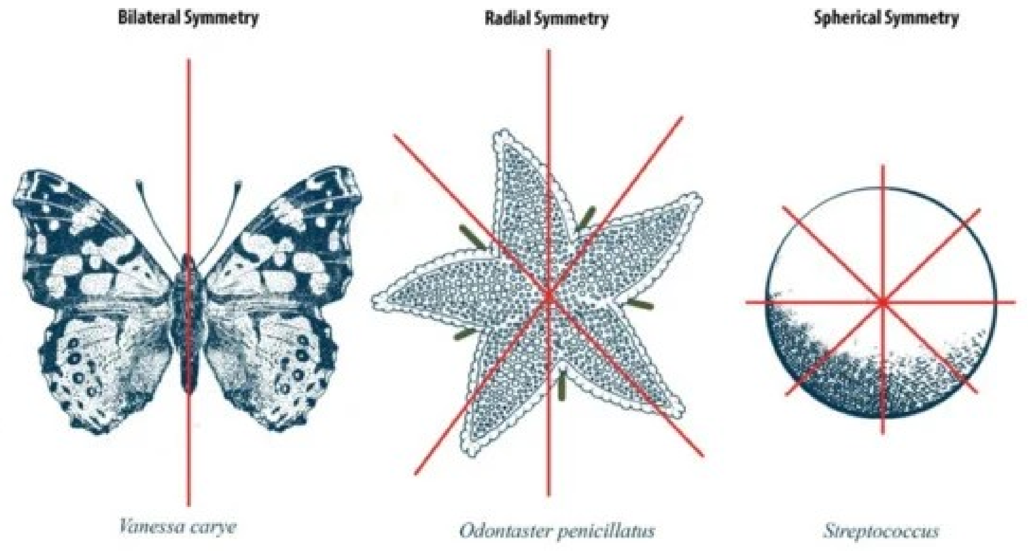
“action weights”

```
def make_move(self, cur_score, river_score, pot, last_bet, can_raise=True):
    # domain-specific part of the action selection process
    w = self.strategy.action_weights([cur_score, river_score, pot, last_bet]) # 4 input dimensions
    betr = max(0, last_bet)
    w = ( w[0:1] +
        [0 for _ in range(betr)] + (
            w[betr + 1:]
            if can_raise else
            ( [w[betr + 1]] +
                [0 for _ in range(3 - betr)]
            )
        )
    )

    # fold -> -1
    # check -> 0
    # call -> bet_to_match
    # raise -> int > bet_to_match
    possible_bets = [-1, 0, 1, 2, 3] # 5 output dimensions
    bet = -1
    if sum(w) != 0:
        bet = random.choices(possible_bets, weights=w)[0]
    return bet
```

player

(,) =

“environment”



player

(,) =

“environment”

```
w = self.strategy.action_weights([cur_score, river_score, pot, last_bet]) # 4 input dimensions
```

player

(,) =

“environment”

```
def score_river(self):
    rs = set([c.shape.value for c in self.river])
    rc = set([c.color.value for c in self.river])

    s = (len(rs), len(rc))
    st = { # These are computed emprircally using a million rounds of simulation
        (2, 1): 0,
        (1, 1): 1,
        (2, 2): 2,
        (1, 2): 3,
        (3, 2): 4,
        (3, 1): 5,
        (2, 3): 6,
        (3, 3): 7,
        (1, 3): 8
    }
    return st[s]
```

player

(,) =

“strategy representations”

```
w = self.strategy.action_weights([cur_score, river_score, pot, last_bet]) # 4 input dimensions
```

```
possible_bets = [-1, 0, 1, 2, 3] # 5 output dimensions
```

$$\begin{bmatrix} \# & \# & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \end{bmatrix}$$

A
Matrix

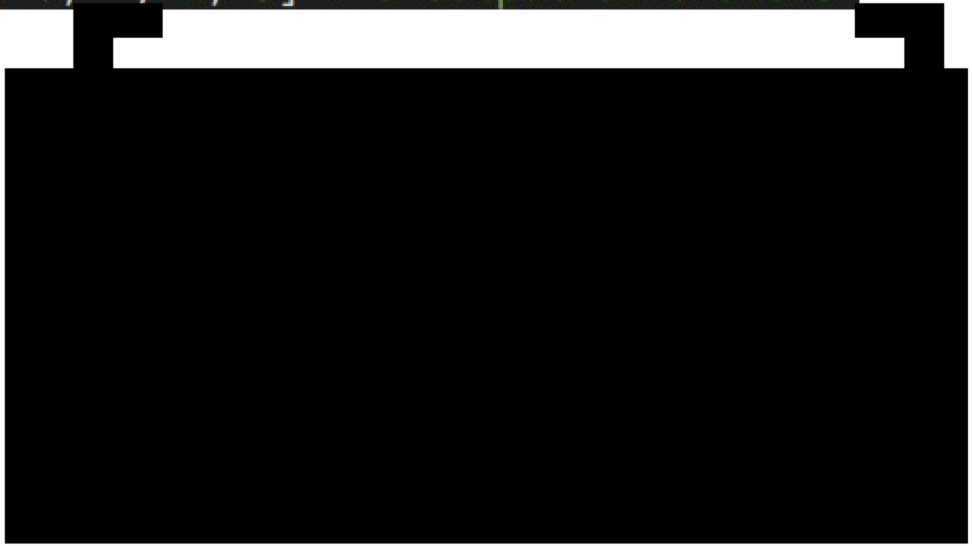
player

(,) =

"strategy representations"

```
w = self.strategy.action_weights([cur_score, river_score, pot, last_bet]) # 4 input dimensions
```

```
possible_bets = [-1, 0, 1, 2, 3] # 5 output dimensions
```



A
Matrix

player

(,) =

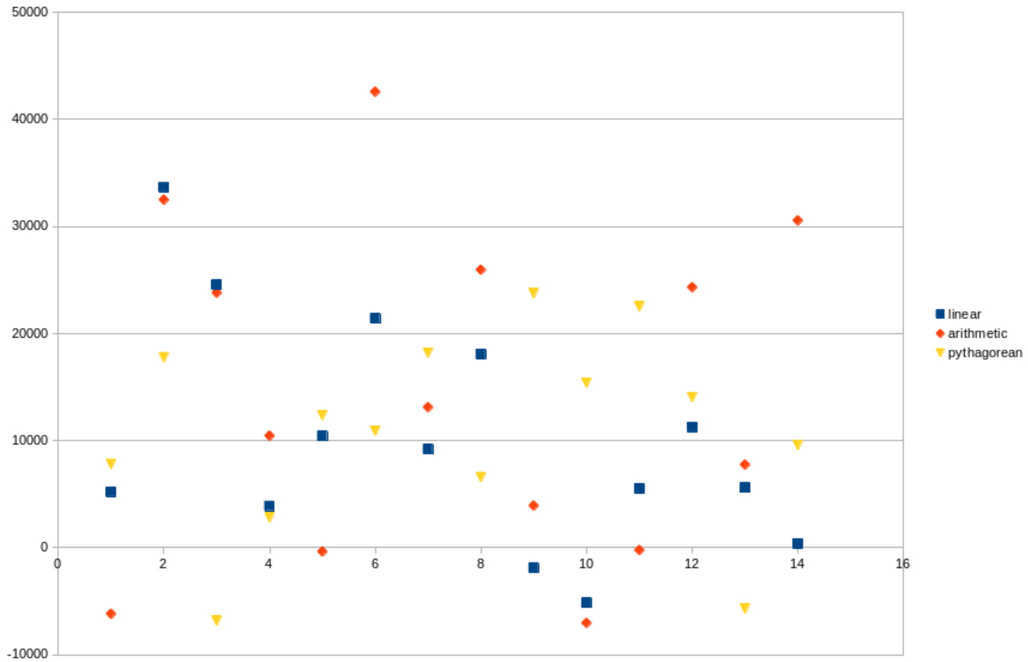
“percept functions”

```
# Various formulas to 'combine' Dna with the input dimensions. These are what I call "percept functions" in my paper,
# because they determine how the agent makes decisions with respect to its environment.

linear_combine = lambda chrom, env: \
    [abs(sum(a*b for a,b in zip(row,env + [1]))) for row in chrom]

arithmetic_combine = lambda chrom, env: \
    [abs(sum(a*b for a,b in zip(row,env + [1])))/len(row) for row in chrom]

pythagorean_combine = lambda chrom, env: \
    [math.sqrt(abs(sum((a*b)**2 for a,b in zip(row,env + [1])))) for row in chrom]
```

player

(,) =

“percept functions”



(,) =

project

- The game – components, rules, simulation ✔
- The player – perception, decision making ✔
- The learning algorithm – feedback, learning

learning algorithm

=

learning algorithm

=

```
def crossover(self, other):
    assert self.inputs == other.inputs
    assert self.outputs == other.outputs

    ret = [[t for t in row] for row in self.chrom]
    col_cross = random.randint(0, self.inputs+1 - 1)
    row_cross = random.randint(0, self.outputs - 1)

    flip1 = random.randint(0, 1)
    r1 = range(row_cross, self.inputs+1)
    if flip1:
        r1 = range(0, row_cross)
    flip2 = random.randint(0, 1)
    r2 = range(col_cross, self.outputs)
    if flip2:
        r2 = range(0, col_cross)

    # splice in other's values after col_cross and row_cross
    for i in r1:
        for j in r2:
            ret[i][j] = other.chrom[i][j]

    return Dna(self.inputs, self.outputs, chrom=ret, mut_rate=self.mut_rate, max_gene_val=self.max_gene_val, combine_formula=self.combine_formula)

def mutate(self):
    how_many_times_to_alter = random.randint(0, self.mut_rate)
    for n in range(how_many_times_to_alter):
        col = random.randint(0, self.inputs+1 - 1)
        row = random.randint(0, self.outputs - 1)
        adj = random.uniform(-1,1)
        if self.chrom[row][col] == self.max_gene_val:
            adj = min(adj, 0)
        elif self.chrom[row][col] == -self.max_gene_val:
            adj = max(adj, 0)
        self.chrom[row][col] = self.chrom[row][col] + adj
```

learning algorithm

Roulette Selection in Genetic Algorithms

Last modified: October 19, 2020

by Gabriele De Luca

Algorithms

1. Overview

In this tutorial, we'll study the roulette wheel selection method for genetic algorithms.

2. Genetic Algorithms

The selection of chromosomes for recombination is a mandatory step in a genetic algorithm. The latter is, in turn, an algorithm that's inspired though not reducible to the evolutionary process of biological species.

Genetic algorithms find important applications in machine learning. For example, we use them in the selection of policies in reinforcement learning. But also, in the optimization of parameters for deep learning, in the subset sum problem, in pathfinding, or, in general, in the solution to many search problems in reasoning and learning.

The real-world problems that they help solve span from the discovery of new materials to the identification of biomarkers in computational biology. But also, the disease screening and drug discovery in medicine. Therefore, they're important tools in the toolbox of any data scientist.

For genetic algorithms to work as intended, it's necessary however to solve the related problem of recombination between chromosomes first.

3. Recombination

A typical definition of a chromosome considers it as a fixed-length array that contains a binary variable:

0	1	2	3	4	5	6	7
0	1	0	1	0	1	1	0

Each bit of the variable then maps to a parameter or characteristic of some type. **In this manner, we can describe an individual possessing a finite set of binary characteristics exclusively in terms of a chromosome**. The chromosome, then, allows the representation of a population that contains as many as 2^n different types of individuals.

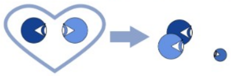
This lets us represent complex agents composed of multiple subsystems. In the case of policies for reinforcement learning, for example, the chromosome normally corresponds to either the agent's perceptual system, or its movement controller, or both.

All individuals, each with their own chromosome, start to operate in the environment. As they do, **each receives a measure of fitness** f_i, according to its performances. Then, after we compute the fitness for all individuals in a population, we proceed with the recombination phase.

During this phase, we select some individuals to act as parents. These parents, in turn, mix their chromosomes according to a procedure called crossover or recombination.

4. Selecting by Fitness

We thus need a method for identifying the parents whose chromosome we subject to recombination:



This method needs to use the fitness of individuals in the population. **Or otherwise, there's no learning between one generation and the next**.

There are two main categories of methods for using fitness in order to support selection:

- deterministic methods
- stochastic methods

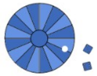
Deterministic methods involve, for example, the selection of the n-most fit individuals in a population for recombination. They're generally discredited because they tend to develop a population that reaches a local maximum and then stops evolving.

Alternatively, we can also use stochastic methods for selecting parents. **The most extreme of these methods select individuals randomly with uniform probability**, and thus completely disregards their individual fitness.

A good middle way, instead, is the roulette wheel selection, which creates a discrete probability distribution from which we identify the chromosomes for crossover.

5. Principles of Roulette Selection

Roulette selection is a stochastic selection method, where the probability for selection of an individual is proportional to its fitness. The method is inspired by real-world roulettes but possesses important distinctions from them. As we know from the movies on casinos and gamblings, roulettes always have slots with the same size:



That means, however, that all slots have the same probability of being selected. Instead, we can implement a weighted version of the roulette. **With it, the larger the fitness of an individual is, the more likely is its selection**:

The first component of the roulette selection method is, therefore, that **individual fitness is proportional to its likelihood of selection**.

This isn't sufficient though. This is because, if the population has n individuals, then the summation of probabilities $\sum_{i=1}^{n} p_i$ of their selections equals one. **As a consequence, we have to also normalize all values for the individual probabilities** to the interval $[0, 1]$.

6. Roulette Wheel Selection

Finally, we can sum up the considerations made above and develop a method that satisfies the requirements we set.

https://www.baeldung.com/cs/genetic-algorithms-roulette-selection

learning algorithm

The building block hypothesis [edit]

Genetic algorithms are simple to implement, but their behavior is difficult to understand. In particular, it is difficult to understand why these algorithms frequently succeed at generating solutions of high fitness when applied to practical problems.

https://en.wikipedia.org/wiki/Genetic_algorithm#The_building_block_hypothesis

learning algorithm

```python
class Strategy:
    def __init__(self, inputs, outputs):
        pass

    def action_weights(self, env):
        pass

    def update(self, score): # score should be the marginal benefit gained by choosing this strategy for this play session
        return False # returns True at the completion of each evolutionary cycle

    def best_chrom(self):
        pass

    def worst_chrom(self):
        pass

    def refresh_metaparams(self, p):
        pass
```

project

- The game – components, rules, simulation ✔
- The player – perception, decision making ✔
- The learning algorithm – feedback, learning ✔

voila!

linear	28239	105915	78249	28989	37614	67809	49341	71405	23631	-15168	19915	53119	36806	13735
arithmetic	5923	103866	84520	37934	5097	129606	50790	88620	15381	-6523	7249	94355	31874	97615
pythagorean	27057	61139	-13826	16631	32606	40239	93924	26165	71092	64033	75037	50518	1658	26149

- Ways to improve more quickly
- Ways to improve sooner
- Ways to improve more
- Ways to improve against another learning opponent
- Document other search kinds (e.g. ant colony) and their effectiveness
- A priori understanding of search, and theoretical optimality
- The effectiveness of stochastic methods
- Inverse problems and solution heuristics
- Various ways to code a solution as DNA*
- Various ways to assimilate experience, including attention-based
- Persisting knowledge gained across games
- Realtime
- Games that require logical consistency in addition to strategy
- The elimination of blind, nonsensical behaviors; intuition or fuzzy reasoning
- Games that require short-term vs long-term tradeoffs i.e. resource management
- Applied parallel or hardware-accellerated computation in an effective way
- Population/ecosystem dynamics

https://defensemaven.io/warriormaven/air/why-an-air-force-6th-gen-stealth-fighter-is-here-almost-10-years-early-Q9gApEljfk-3iXPKqvmH5Q

thanks!

Seems like an excellent place to work.