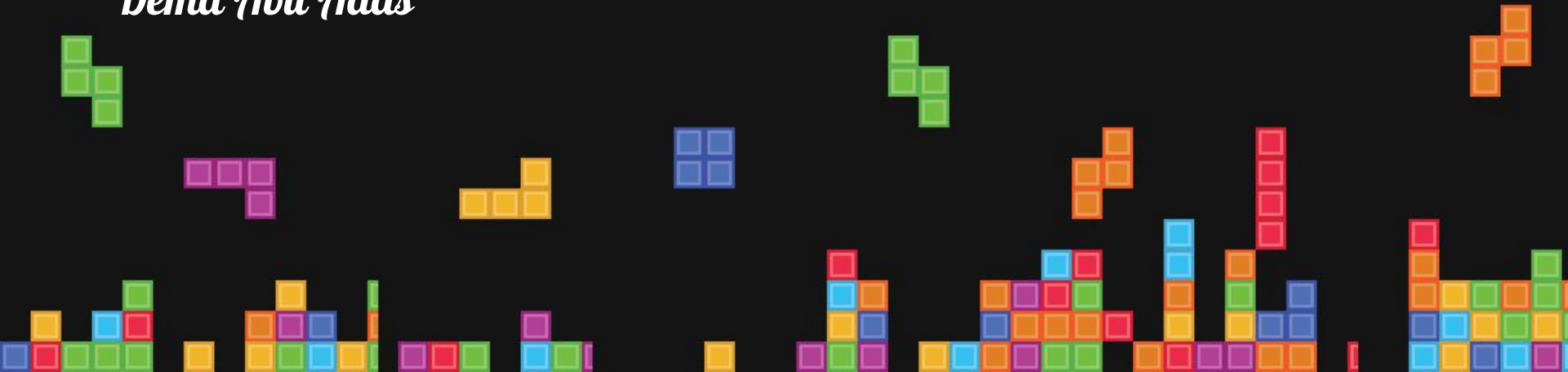
Designing an Evolutionary Algorithm for Tetris!

Dema Abu Adas



Recap of what i'm doing

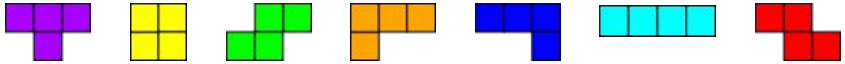
What Makes a Genetic Algorithm

Based on Darwinism: A theory of biological evolution stating that all species of organisms arise and develop through the natural selection of small, inherited variations that increase the individual's ability to **compete, survive** and **reproduce**.

Phases in a genetic algorithm:

1. Initial population
2. Fitness Function
3. Selection
4. Crossover
5. Mutation

What is Tetris?



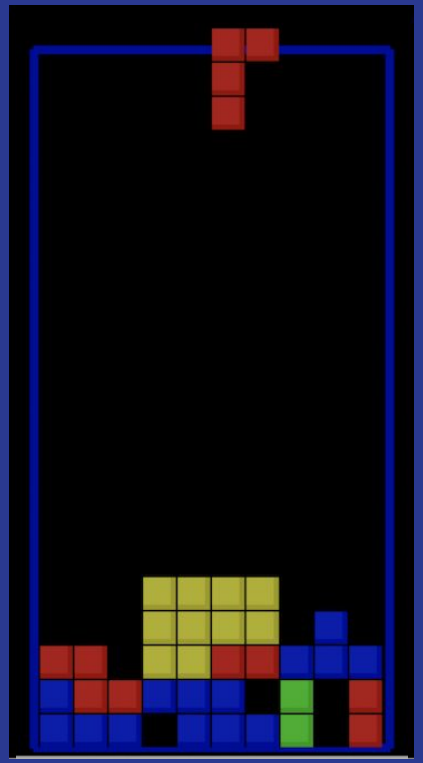
- Tile matching game
- Strategically rotate, move and drop Tetriminos that fall into a 10 x 20 Matrix and increasing speed
- Clear lines by completing horizontal rows of blocks without empty space!
- Lose if the Tetriminos go over the Matrix height

Initial Population

= Penalize heights

+ Reward line clearing

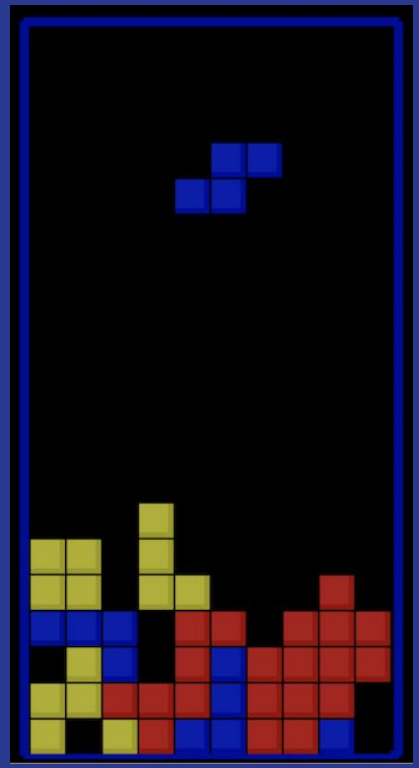
Height



Initial Population

= Penalize heights
+ Reward line clearing
- Penalize holes

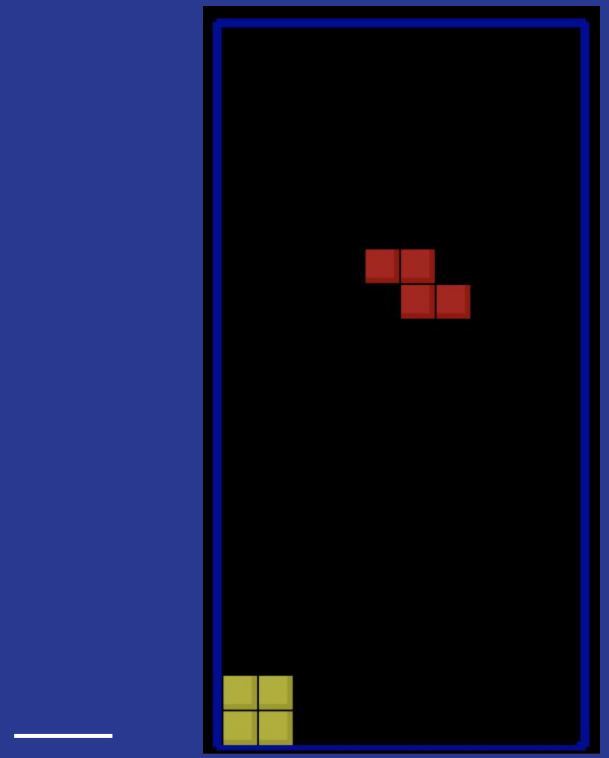
Holes



Initial Population

= Penalize heights
+ Reward line clearing
- Penalize holes
- Penalize blockades
+ Reward Touching Pieces

Piece Location



Objective Function

= # of Lines Cleared w/ 500 Tetrominos + blocks remaining in our game Matrix

Fitness Test

= (k1 * Penalize heights)

+ (k2 * Reward line clearing)

- (k3 * Penalize holes)

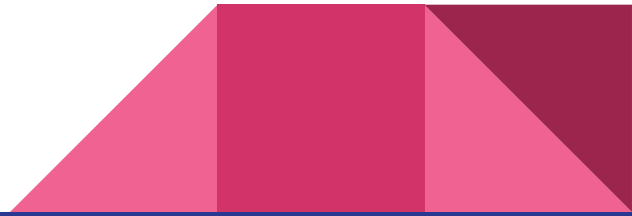
- (k4 * Penalize blockades)

+ (k5 * Reward Touching Pieces)

Where k1, k2, k3, k4, k5 are random variables that we will optimize

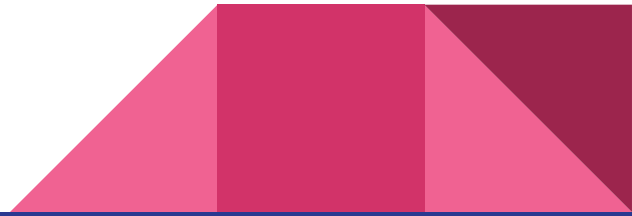
Crossover and Mutation

- Single point crossover
- Creep mutation



Goals I'm Trying to Achieve!

- To be able to generate a genetic algorithm in Tetris
- Bonus: Be able to implement the hold function as apart of my Genetic Algorithm



What I've Achieved

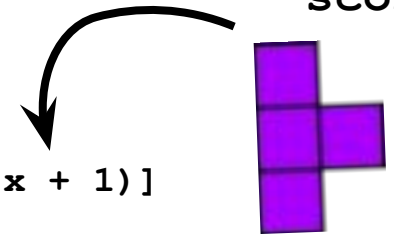
1. Moving Pieces Around

- Using the PyGame module (A game library)
- “Mocked” events like they’re coming from the keyboard

2. Calculating Best Move - Getting All Options

1. Check which piece I have (every piece works well in different surfaces)
2. Create an option for every possible move (* rotation)

```
option =
{ 'shapeList': sL,
  'index': indexTo,
  'rotation': 1,
  'score': 0 }
```



```
[(y - 2, x), (y - 1, x), (y, x), (y - 1, x + 1)]
```

1. Moving Pieces Around

1. Took an x coordinate and rotation offset
2. Calculated how many times the user would “hit” left or right to get the piece to go to the x coordinate
3. Calculated how many times the user would “hit” up to rotate the block
4. Returned the sequence of moves!

Why scoring is important?

- Before scoring I would take a piece and try to find the surface that corresponds to its “ideal” position and place it
- Helps in case there is no “optimal” place to put the piece
- Need it for optimization..

2. Calculating Best Move - Scoring Options!

We go through and check to see how ideal it is to make the piece go to that option

- Height
- Hole
- Touch Wall
- Touch Ground
- Touch piece
- Clear line

2. Calculating Best Move - Scoring Options!

- Check how many y coordinates the piece has and multiple it by a constant
- This forces the algorithm to favour rotations that don't take up too much height

- Height
- Hole
- Touch Wall
- Touch Ground
- Touch piece
- Clear line

2. Calculating Best Move - Scoring Options!

- Take a reduced form of the board
- “Place” the piece (“T”)
- Check underneath the piece if there is a hole
- Multiple hole penalty * # of holes

- Height
- Hole
- Touch Wall
- Touch Ground
- Touch piece
- Clear line

2. Calculating Best Move - Scoring Options!

- Touch wall:
  - Check if x coordinates are -2 or 7
- Touch ground:
  - Check if y coordinates are 19

- Height
- Hole
- Touch Wall
- Touch Ground
- Touch piece
- Clear line

2. Calculating Best Move - Scoring Options!

- Take a reduced form of the board
- “Place” the piece (“T”)
- Check if its touching a piece

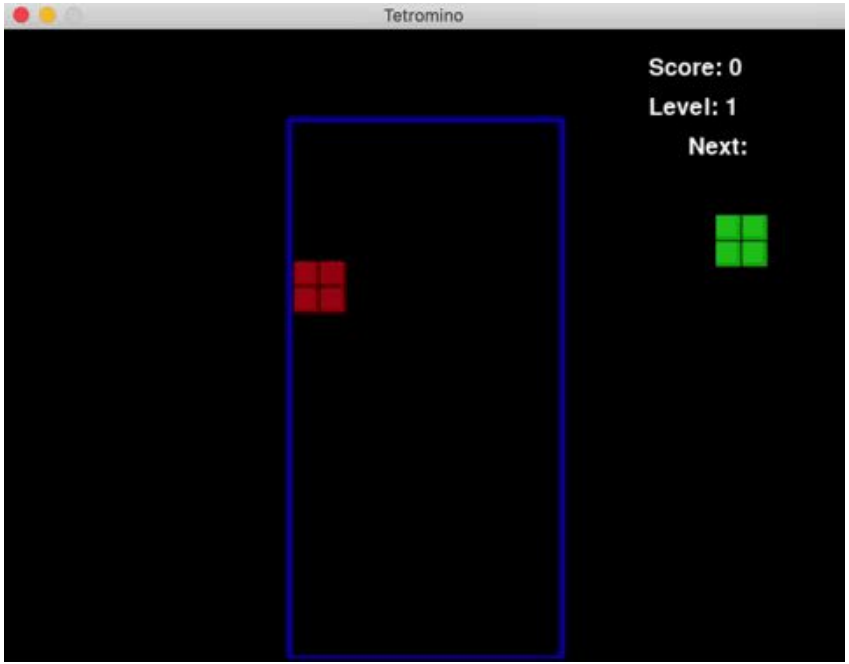
- Height
- Hole
- Touch Wall
- Touch Ground
- Touch piece
- Clear line

2. Calculating Best Move - Scoring Options!

- Take a reduced form of the board
- “Place” the piece (“T”)
- Check how many horizontal lines don’t contain an empty space (“.”)

- Height
- Hole
- Touch Wall
- Touch Ground
- Touch piece
- Clear line

Non scoring vs scoring movements



Implementing the Genetic Algorithm

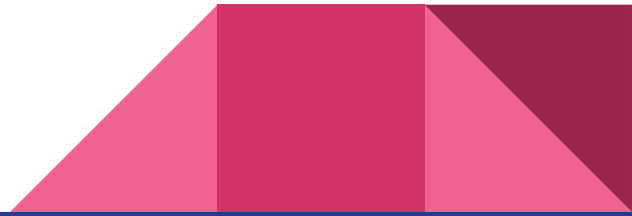
Initial Population

- Created a population of 10 chromosomes! Each with random numbers of what the scoring could look like

```
{ "height": -random.randint(10, 20),
    "hole": -random.randint(10, 20),
    "touchPiece": random.randint(0, 1),
    "touchWall": random.randint(0, 2),
    "touchFloor": random.randint(1, 10),
    "clearLine": random.randint(10, 25),
    "totalScore": 0}
```

Fitness Test

- Height * k1
- Hole * k2
- Touch Wall * k3
- Touch Ground * k4
- Touch piece * k5
- Clear line * k6



Selection

- Selected top half of the population
- Paired each one up randomly to produce offsprings

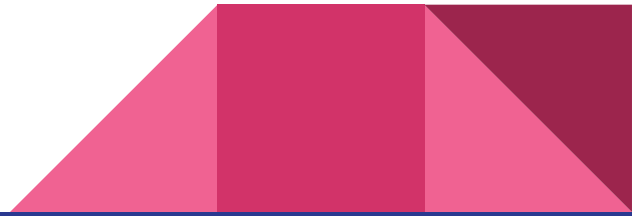
Crossover

- Created new offspring by switched over which genes which chromosome has

Crossover

```
def crossover(pop1, pop2):
 pop1['clearLine'], pop2['clearLine'] = pop2['clearLine'],
pop1['clearLine']
 pop1['height'], pop2['height'] = pop2['height'], pop1['height']


 return pop1, pop2
```

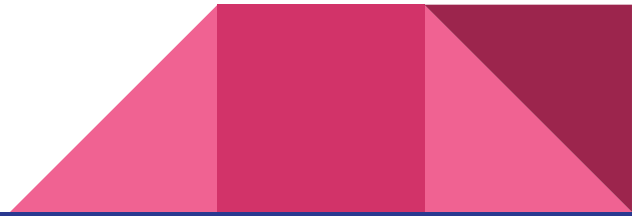


Mutation

- Took the “less important” scoring features and added/subtracted a random small offset

Mutation

```
def mutate(pop):
 pop['touchFloor'] += random.randint(-3, 3)
 pop['touchPiece'] += random.randint(-1, 1)
 pop['touchWall'] += random.randint(-1, 1)
```



Results..

Results!

Generation 1:

[21, 19, 19, 18, 17, 17, 17, 15] Avg: 17.875

Generation 2:

[21, 21, 20, 20, 20, 20, 18, 16] Avg: 19.5

Generation 5:

[32, 24, 22, 19, 18, 16, 16, 15] Avg: 20.25

It kind of worked?

- Didn't really clear lines, but tried its best
- Piece indexes for movements were weird depending on rotation, sometimes off by 1
- sometimes choked on where it was supposed to go

Questions, Comments, Concerns?!