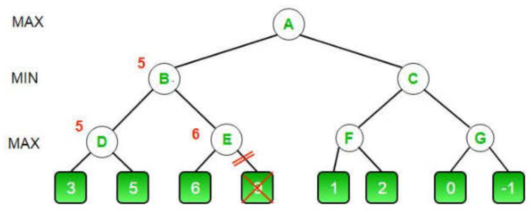
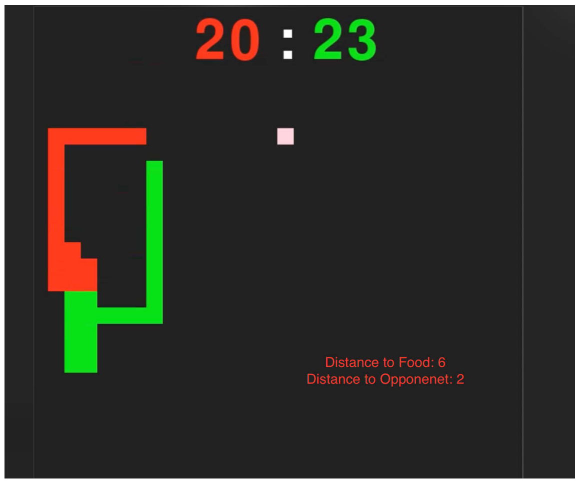
# MiniMax

- The MiniMax Algorithm is a backtracking algorithm which is most commonly used for game playing AI's

- MiniMax can be optimized with Alpha-Beta pruning which is essential in a real time game such as Snake



- We used two main Heuristics: Distance from food & Distance from Opponent
- We later added a penalty heuristic so the Snake would avoid closed areas
- Results
  - Out performed humans at low depth
  - Decided the best strategy was to loop on its tail
  - With Alpha-Beta pruning at 15 fps we were able to run MiniMax at depth 6
  - Depth 6 required us to add the penalty heuristic as it could not see far enough ahead
- Future Optimizations: Monte Carlo tree search would allow us to explore much deeper depths



# A* Search Algorithm

**What is A*?**

A* is a graph traversal algorithm that ensures the snake will always find the optimal route to its food. It combines the actual travel cost and an estimated cost to the goal to find the most efficient route.

**Why Chose A***

- Guarantees the optimal path to the food
- Adaptable: Works well in a dynamic environment, avoiding obstacles like the snakes body
- Efficiency: Can be calculated in real time on limited hardware while playing

**How it works**

G-Score: Movement cost from the starting point to a given square

H-Score: Heuristic cost estimate to move from a given square to the final destination

F-Score: Total Cost of G Score + H Score used to determine the next best move

<u>Brief Steps:</u>

1. Initialization: Create a frontier list containing the starting node
2. Cost Calculation: Calculate the g-cost, h-cost, and f-cost of the
3. Node Selection: Select the node with the lowest f-cost from the frontier list
4. Goal Check: Check if the selected node is the target
5. Node Expansion: Expand all of the adjacent nodes and add them to the frontier list unless the are already in the visited list or a cheaper path has been found
6. Repeat: Continue this process until the target node is reached or the frontier list is empty

# Custom Heuristic

**1. Distance Metric**

Some of the popular distance metrics include Manhattan Distance, Diagonal Distance, and Euclidean Distance. In our game of snake the player can only move north, east, south, and west. Thus we decided to use Manhattan distance.

Manhattan Distance = $|x_1 - x_2| + |y_1 - y_2|$

**2. Player Avoidance**

Penalizes paths that bring the snake closer to the opponent or into likely collision paths. To calculate the weight for this metric we found the

**3. Head-on Collision Detection**

Penalizes paths that would put the snake in a position of head-to-head collision with the opposing snake

**4. Free Space Detection**

Considered the amount of free space around potential paths to ensure mobility and escape routes

# NEAT Algorithm

***N**euro**E**volution of **A**ugmenting **T**opologies*

What is **NEAT**?

- **NEAT** is a combination of the concepts of Genetic algorithms and Neural Networks (NN).
- It attempts to "grow" an NN from a simple start through random changes

Link to the NEAT Paper

Why did I pick **NEAT**?

- We only touched upon Genetic Algorithms during class, and I wanted to see how they worked. When I found out about **NEAT** I wanted to play around with it to see how effective it was.

What was my experience with **NEAT?**

- **NEAT** was *incredibly* sensitive to the fitness equation, even more than the Min-Max was. A tremendous amount of time was spent just tuning it until it aligned with my intentions.
- Designing the inputs to the NN required a few iterations before I found one that trained quickly enough
- Simple tasks (e.g. eating food) were easy and quick to train. But advanced tasks greatly increased the training time with each additional level of complexity.
- Overall, designing an implementation for a competitive snake game was slow and difficult.

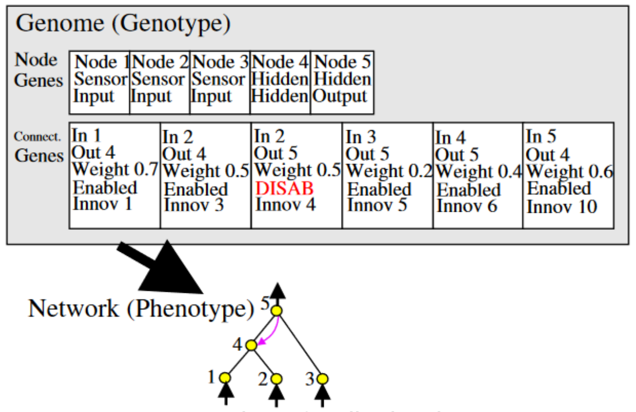
What implementation of **NEAT** did I go with**?**

- These results were made with NEAT-Python, which is built on the PyTorch framework.

# How Does NEAT Work?

The **NEAT** algorithm builds a structure referred to as a “genome,” which has **node genes** and **connection genes.**

The **node genes** represent the individual neurons, and the **connection genes** contain the connecting edges and activation functions for the **node genes.** They also contain additional information that **NEAT** uses to work.



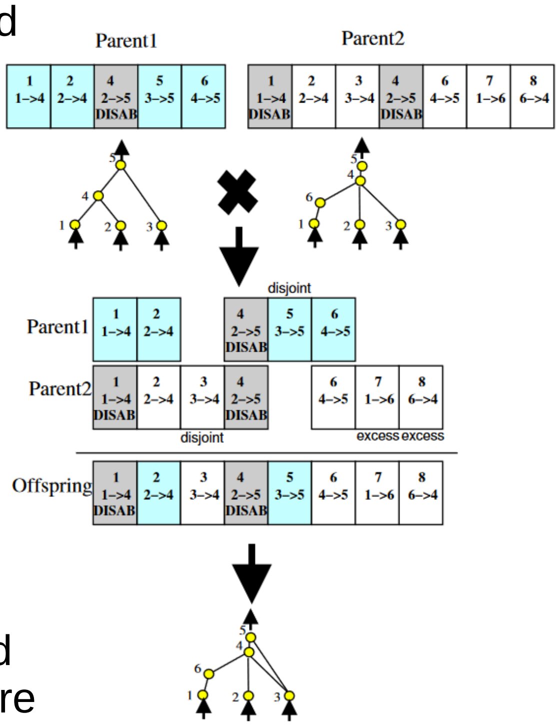
Genome (Genotype)
Node Genes
Node 1 Sensor Input
Node 2 Sensor Input
Node 3 Sensor Input
Node 4 Hidden Hidden
Node 5 Hidden Output
Connect. Genes
In 1 Out 4 Weight 0.7 Enabled Innov 1
In 2 Out 4 Weight 0.5 Enabled Innov 3
In 2 Out 5 Weight 0.5 DISAB Innov 4
In 3 Out 5 Weight 0.2 Enabled Innov 5
In 4 Out 5 Weight 0.4 Enabled Innov 6
In 5 Out 4 Weight 0.6 Enabled Innov 10
Network (Phenotype)
5
4
1
2
3

Demonstration of a disabled gene
*Image taken from the NEAT paper*

A fitness evaluation is checked for each generation, and more fit models are increased in population, less fit are reduced, and Unchanging “stagnant” models are eliminated.

Then the settings such as weights, connections, or even the activation functions are “mutated” randomly by **NEAT**.

Besides the mutations the structure can also change by mimicking sexual reproduction. Two genomes can have their genes combined in order to synthesize a new structure.



Combining two genomes
*Image taken from` the NEAT paper*

In the past, the differing topologies that occurred from random changes made it difficult to compare the fitness of two different neural nets or “mate” them together, since this would only work if they Were similar enough in structure. **NEAT** handles this by giving each new connection gene an “Innovation” number stored in the node gene.

This can be used to track ancestry of genes, which implies similar structures that can be used for comparison and combination. It also permits us to break genomes down into “species,” which lets a new mutation (which initially lowers fitness) have a chance to flourish without having to compete with higher fitness species already dominating the environment.

# How Long To Learn Features?

With **NEAT** we lack direct control of designing the NN. This means how long a **NEAT** structure takes to learn is a little bit random.
Here's some rough guidelines of how many generations it took for our implementation to learn certain topics.
*Note: Each generation is 800 individuals*

**300 Generations** (**0.83 Hours**) – Navigates to food directly, but dies heading on to the next food, usually by doubling back on itself.

**500 Generations** (**1.3 Hours**) – Can eat food without killing itself, but if food doesn't appear instantly it simply runs into walls or players.

**1200 generations** (**3 Hours**) – Learns walls are fatal, most of the time.

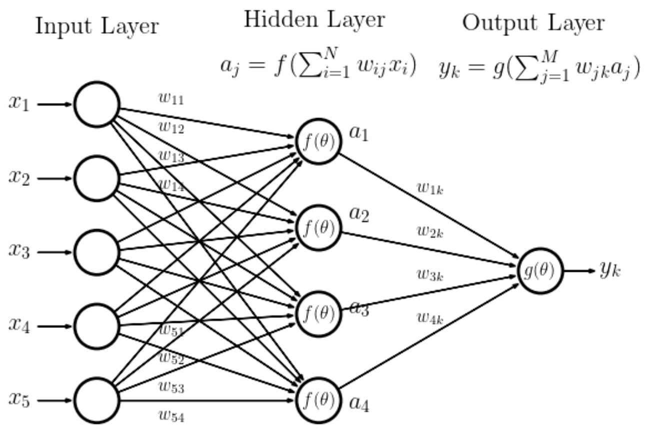
**8000 generations** (**22 Hours**) – Learns to spin in place when no food so that it doesn't die when waiting for food to appear

On average, it took 10-15 seconds to process a generation.

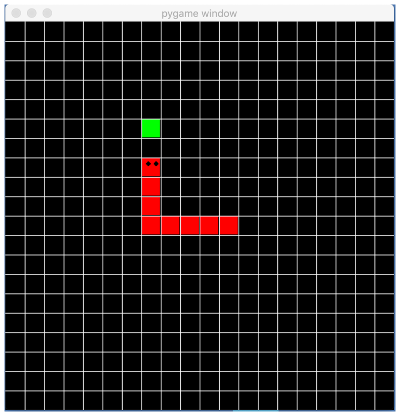
GPU Acceleration for **NEAT** wasn't well developed until as recently as April 2nd, 2024, when **TensorNeat** was published.
**TensorNeat** uses the JAX framework

Link to
**TensorNEAT** Paper

# Traditional NN - Analyzing on top of Minimax



**Data Collection-** Collects data from minimax algorithm, and builds on top of it.



**Architecture:**

The NN architecture comprises multiple fully connected layers, also known as dense layers. These layers enable the NN to learn complex patterns and relationships within the game data. The input layer of the NN accepts the game state features, which are then passed through one or more hidden layers. These hidden layers perform nonlinear transformations on the input data, extracting abstract representations that capture important features for decision-making. Finally, the output layer produces predictions for the next move based on the learned patterns.

**Training:**

The NN is trained using a supervised learning approach, where it learns from a dataset containing examples of game states and corresponding optimal moves. During training, the NN adjusts its internal parameters, known as weights and biases, to minimize the difference between its predictions and the true optimal moves. This optimization process is achieved using an algorithm called back propagation, coupled with an optimization algorithm such as Adam.