EvoSim: A Comprehensive Evolution Simulator Based on Biologically-Inspired Neural Mechanisms

1st Kaiyuan Lou
Dukekunshan University
Suzhou, China
kaiyuan.lou@dukekunshan.edu.cn

I. Introduction

Creating virtual creatures is an interesting and meaningful topic, which has been explored for many years. Inspired by Karl Sims' Evolving Virtual Creatures [1], I designed an entire 2d based framework for evolving virtual creatures, including a virtual environment with physical simulation, creature's gene design and morphology, a brand-new neural network structure to control the virtual creatures, and a proper way to training the neural network using genetic algorithm.

What's more, the simulation is designed with full multi-threading and cross-platform support, and interactive UI.

In the following sections, I will introduce each part of the complex design one by one.

II. Environment & Design Pattern

EvoSim harnesses the capabilities of the Bevy game engine for its core operations. To ensure realistic interactions within the environment, the project integrates the Rapier physics engine. However, to meet the specific demands for precision and efficiency in physical simulations, I have implemented my own tailored fork of Rapier.

At its foundation, the entire project is meticulously crafted using the Rust programming language.

To harness the advantages of parallel computing, the design embraces the Entity Component System (ECS) paradigm over the traditional Object-Oriented Programming (OOP) approach.

The virtual creatures boast a distinct hierarchical neural network. Organizing such a network in a conventional nested manner would immensely complicate parallel computation due to the intricate interdependencies present within the neural network's segments. To navigate this complexity, I've restructured the neural network. Each expansive network is fragmented into its smallest functional units, which are then interlinked. An auxiliary scheduling system is incorporated to dictate the sequence of processing. This reconfiguration significantly simplifies the parallel processing of the numerous neural network units.

For managing other resources and datasets, it leans into Bevy's ECS framework, which offers robust support for parallel operations.

III. Gene & Morphology

For the virtual creatures, affectionately termed "blobs", the goal is to encourage a diverse range of forms. They were built in a significant degree of freedom, allowing them to evolve into any shape or structure, as long as they adhere to certain guidelines:

- **Block Unit:** At its core, every blob is composed of a minimum of two blocks. These blocks, shaped like rectangles, represent the basic rigid-body units within the simulation. Depending on the creature's structure, a block can function as a bone, limb, or any other body part. Detailed documentation can be found here.
- **Neuron:** Each block unit has its own neural network, the id of NN will be stored in the blob's gene.
- **Joint & Anchor Points:** Every block is designed with four anchor points situated at the center of each of its edges. Each of these anchor points can connect to, at most, one other block via a joint.
- **Hierarchy:** The structure of a blob is analogous to a tree. Every block (or "node" in tree parlance) can have up to one parent and up to three children. The tree's depth - that is, the number of generational layers it can have - is configurable.
- **Validation:** For a blob's morphology to be considered valid, it should be possible to depict it in a 2D space without any overlapping components.

This framework ensures that while blobs have the freedom to evolve and diversify, they remain within a coherent and manageable system.

A. Representation of Gene

Given that a blob's structure resembles a tree, I've chosen the **QuadTree** as the data structure for representing a blob's gene. Interestingly, while each block can potentially connect to four other blocks, due to the restriction that one anchor point always links back to its parent, the resultant tree structure is a **ternary tree**.

Yet, I opt for a quad tree over a ternary tree. The reason lies in my need to preserve not only the hierarchical information of each block (i.e., identifying its parent) but also its positional details. Specifically, I need to know which anchor point of the block is linked to its parent. This nuanced representation allows us to identify block locations via indices.

Within the tree design:

- Indexes 1, 2, 3, and 4 correspond to the anchor points: up, down, left, and right, respectively.

- If no children are connected to an anchor point, its value is set to `None`.
- If an anchor is connected to valid block, its value is an enum GenericGenoNode, which can be either `Parent` or `Child`, where parent is an indicator and child is a GenoNode that contains block's information.

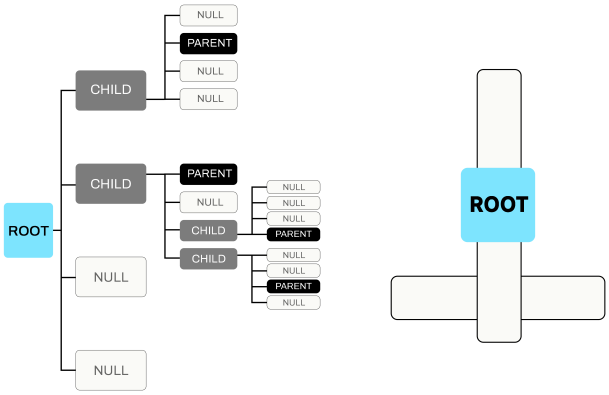

Fig. 1: Gene and its relate morphology

As shown in Figure 1, the left-hand side QuadTree represents the right-hand side blob structure.

B. Gene Validation

In this structure, every blob's morphology can be represented by a unique gene, but not every possible gene relates to a valid structure. So, when a gene is randomly generated, or mutated, I need to check if the new gene corresponds to a valid morphology structure.

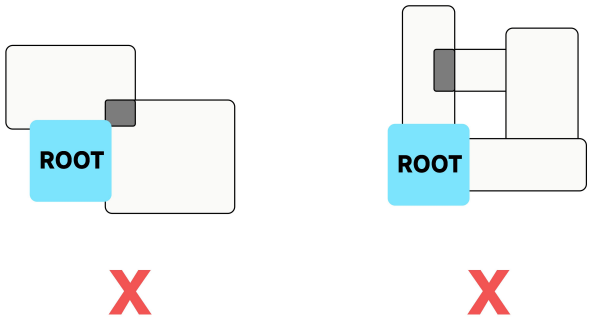

Fig. 2: None-valid genes

According to Figure 2, the first type of invalidation is easy to prevent if I simply limit the size of child blocks, but the second one is hard to prevent by limiting the generation rule. So, instead of modifying the genration rule, I just check the validation each time there are morphological changes.

IV. Physical Simulation

To enable the virtual creatures to engage with their surroundings and other entities, a robust physical simulation is implemented. This simulation allows collisions between objects, enabling creatures to exhibit behaviors like paddling, scratching, pushing, and pulling.

For foundational physical simulations, the [Rapier](https://rapier.rs/) physics engine is chosen to manage collision events. However, there are encountered challenges, especially with joint functionalities.

For full documentation, please visit document for Module evosim::physics.

A. Joints

Our morphology design accentuates the importance of joints in shaping the creatures' appearance and behavior. The joint capabilities offered by Rapier did not align entirely with the design requirements, particularly in:

- **Joint Limit:** Ideally, a joint should possess the flexibility to move within a defined range, analogous to how the human arm can't point in every conceivable direction. Yet, Rapier's joint limits only span from -90 to 90 degrees, restricting the possibility of a joint moving within a 180 to 360-degree range. For a comprehensive explanation, review this issue.
- **Joint Motor:** Echoing the limitations of the joint limit, Rapier's joint motor is also confined to a 180-degree range. This doesn't suit the virtual creatures since joint motors act as their muscles, necessitating a broader range. Further details can be found in this issue page.

To address these constraints without compromising the benefits of Rapier, I developed a own fork of the engine.

B. Constraint and Constraint Solver

Physics engines like Rapier don't exclusively deal with collisions. For instance, the force experienced between two rigid bodies connected by a joint isn't a direct result of collision. In such scenarios, the joint serves as a constraint that delineates the permissible behaviors of the connected entities.

Rapier leverages the Jacobian matrix for evaluating and resolving these constraints. The constraint solver operates as follows:

- Predict the upcoming positions of objects based on existing velocities and forces.
- Assess how much the predictions violate the constraints.
- Adjust the object velocities using iterative relaxation to minimize constraint violations, with the Jacobian facilitating these corrections.

The rapier also use **Sequential Impulse** that iteratively applying impulses to objects to resolve constraint.

Our focus was to alter how Rapier interprets and manages joint limits and motors. To achieve this, I delved deep into Rapier's constraint builder and solver.

C. Our Modification

Rapier is a finely tuned physics engine that manages diverse constraints with dedicated solvers. As an illustrative example, let's consider the modifications to the `limit_angular`

function. For a comprehensive view of all modifications, refer to the github repo.

The original `limit_angular` function accepts two physical units connected by a joint and their respective joint limits as inputs. It then calculates the angular error, sets impulse bounds, constructs the angular Jacobian, computes the Right-Hand Side (RHS), derives the angular Jacobian for the connected bodies, and ultimately integrates the constraint into the `JointVelocityConstraint` structure. I modified a limit resizing process to make the functioning angular extended from $(-90, 90)$ degrees to $(-180, 180)$ degrees.

D. SIMD programming

Adapting the function wasn't straightforward due to its parallel operation optimizations. Notably, the function is **devoid of branching**. It employs **SIMD** (Single Instruction, Multiple Data) to bolster parallel support, wherein each variable represents a list of elements.

Given the SIMD approach, traditional branching using `if` on booleans isn't feasible, as the boolean variable encapsulates an entire list of boolean values. Thus, bitwise calculations is used for branching. Consequently, the modified version is more extensive as I needed to compute different impulses and Jacobian values based on whether the angle exceeds 90 degrees or not.

E. Fluid Simulation

To breathe life into the virtual creatures, it's essential they interact seamlessly with their environment. But when it comes to fluid simulation, which is pivotal for creatures that move in water, the challenge mounts. Fluid dynamics can be computationally intensive, demanding significant resources.

I weighed two primary methods during the initial phase of implementation:

- **particle based simulation:** As the name suggests, this method relies on the simulation of individual particles to mimic fluid behavior. It boasts of being one of the most accurate methods available, and the simplicity of its implementation is an added advantage. However, the method is not without its drawbacks. High computational costs are a significant barrier. Using larger particles can compromise the accuracy of the fluid behavior, while opting for smaller ones can restrict the size of the world aimmed to simulate.
- **Viscosity Effect:** A viscosity effect is used for the simulations in underwater environments. For each exposed moving surface, a viscous force resists the normal component of its velocity, proportional to its sur-face area and normal velocity magnitude. According to Karl Sims, "This is a simple approximation that does not include the motion of the fluid itself, but is still sufficient for simulating realistic looking swimming and paddling dynamics." Compared to particle-based simulations, it's faster but demands meticulous and intricate implementation. Moreover, in this model, creatures can't influence water movement because the water itself doesn't 'move'. A potantial bug case for viscosity effect simulation is that, for example, putting an propeller inside a box, the box can still move once the propeller starts even the box is a confined space.

Considering the pros and cons, I decided to employ the viscosity effect for every object in motion, thereby simulating an underwater environment. You can find the implementation here.

F. Collision Rules

As outlined in the morphology design for the virtual entities, certain collision events must occasionally be deactivated to ensure normal behavior of blobs.

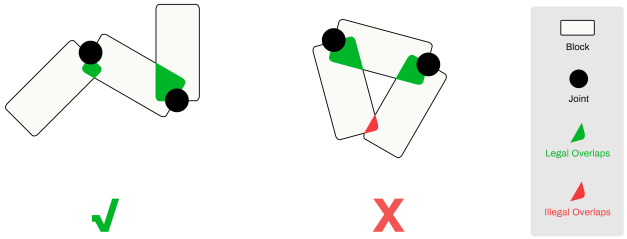

Fig. 3: Rules for collision

Figure 3 delineates the specific collision rules applied to blobs. Notably, each block is designed to avoid collisions with its immediate parent and child nodes, allowing for unimpeded movement around the joint. Yet, when two blocks connected to the same parent come into contact, a collision is triggered.

This design ensures that blobs maintain a tactile awareness of themselves, balancing self-recognition with joint flexibility.

V. Neural Network

Virtual creatures, also referred to as blobs, are endowed with significant morphological flexibility (as elaborated in the Gene section). To bestow these entities with the capability to manipulate their bodies and sense their surroundings, they are equipped with a neural system. This intricate neural network granting blobs both sensory and motor capabilities. You can find the implementation and documentation here.

The design of a neural network, particularly for controlling the adaptable and evolving virtual blobs, hinges on two pivotal characteristics: structural flexibility and inheritability. These twin pillars shape the entire approach, distinguishing the system from more traditional neural network structures.

A. Structural Flexibility

Blobs vary in size and function, and as a result, they differ in their sensory and motor requirements. For instance, a simpler blob might have just two blocks and one joint, while a more complex one might possess numerous blocks and joints. Hence, the neural network must be flexible, scaling according to the blob's individual requirements.

The architecture must accommodate varying input lengths (sensors) and provide outputs (motors) of differing lengths

accordingly. This dynamic architecture is pivotal for blobs to adequately interact with their environment regardless of their size.

Leveraging structures like RNN, LSTM, or Transformer can achieve such flexibility. These architectures are adept at managing sequential data of variable lengths.

B. Inheritability

Given that the system is driven by genetic algorithms, the neural network must be robust against genetic mutations since the neural network will mutate in both structure and weight. In particular:

- **Resilience to Mutation:** The neural system should function in a modular fashion, where a change or mutation in one section doesn't drastically impact the overall behavior. This modularity ensures that if a blob undergoes a mutation, leading to a loss of a limb, the rest of its body can still function appropriately.
- **Interdependency Management:** While connections and dependencies between nodes in a network are typical, excessive interdependency can be detrimental in this case. If one part of the network gets mutated, and if there's a high degree of interdependence, the entire network might behave erratically. Thus, the architecture should strike a balance, ensuring nodes operate somewhat independently while still maintaining network cohesion.

C. OctopusNet

Emulating the decentralized intelligence seen in real-world cephalopods like octopuses, the "Octopus Net" seeks to solve the dual challenges of inheritability and structural flexibility. This neural architecture, crafted specifically for the blobs, ensures adaptability without compromising integrity.

The architecture comprises a **Central Brain (CB)** positioned at the root block, paired with numerous **Peripheral Neural Units (PNU)** present in all other blocks. The functional dynamics of the Octopus Net are categorized into **Inward Propagation** and **Outward Propagation** phases.

A detailed explanation of OctpousNet will be in the next section.

D. Neural Signals

To interact with and respond to their environment, blobs are equipped with sensors and muscles. The data generated by active sensors and the signals that can regulate muscles are termed as neural signals. Within the simulation, three distinct neural signals have been implemented: `BrainSignal`, `InwardNNInputSignal`, and `OutwardNNInputSignal`. More details can be found in the documentation.

In the neural architecture of the blobs, muscles are operational during the Outward Propagation process, while sensors play a role in both inward and outward propagation.

Signals for Inward Propagation:

- **Collision Type:** Information about the type of collision is collected when a block interacts with any rigid body other than its immediate parent or offspring. This could involve collisions with walls, other blobs, or even its own separate limbs.
- **Collision Vector:** For any collision involving a block, its PNU captures data describing the collision's direction and force.
- **Collision Magnitude:** Even though the neural network can derive the intensity of a collision based on its vector, a single float value representing the collision's magnitude is provided to ensure quicker and more efficient information processing.
- **Current Joint Motor's Data:** The joint motor, acting as the blob's muscle, relays information that includes its target position and velocity.
- **Joint Information:** The blob is also sensitive to its joint's metrics, such as its current angular velocity and position.
- **Block's Location (under implementation):** PNUs may require the geographical positioning of their respective blocks.
- **Children's Outputs:** When a block has children blocks, the synthesis of its own data with its progenies' is crucial. Therefore, a child's output is a necessary input for the current block, ensuring that the Central Brain (CB) receives comprehensive data.

For the **Outward Propagation** via the PNU's OutwardNN, the input signals remain identical with those of the InwardNN, except that the outputs from children are replaced with those from the parent. This is because the outward propagation follows a top-down approach.

Additionally, the OutwardNN of the PNU not only relays general commands meant for the child nodes but also outputs signals governing the joint motor. Each OutwardNN solely produces two control signals: the joint motor's target position and its target velocity. These two outputs determine the joint motor's force and direction.

Signals for **Central Brain (CB)**:

- **Collision Data:** Positioned at the blob's root block, the CB is susceptible to collisions. Therefore, it's vital for the CB to process collision-related inputs, encompassing aspects like collision type, vector, and magnitude, akin to the PNUs.
- **Blob Metrics:** Given the CB's role in issuing overarching directives, it's equipped to discern holistic blob details. Key metrics such as the blob's center of mass and current velocity fall under its purview.
- **Number Generator (under implementation):** Emulating real-world organisms, the virtual entities will incorporate a random number generator and oscillator. This feature allows them to make stochastic decisions and introduces an intrinsic rhythm, facilitating recurring movements.

VI. OctopusNet

Octopus Net is a neural architecture, crafted specifically for the blobs, ensures adaptability without compromising integrity.

Granular & Hierarchical Structure:

- **Peripheral Neural Units (PNU):** Associated with every block of the blob, these are the smallest computational units. Each PNU processes sensory input unique to its respective block and issues commands to its specific joint motor. This allows for autonomous decision-making at a micro level.
- **Central Brain (CB):** Positioned in the root block, the CB operates as a coordinator. It receives sensory information from all blocks, processes it, and emits overarching commands. These signals can modify or influence behaviors across all blocks, ensuring coordination.

Signal Transmission & Behavior Modulation:

- **Bottom-up Transmission (Inward Propagation):** PNUs send specific sensory information hierarchially to the CB. This aids in holistic decision-making by the CB, considering inputs from all over the blob's body.
- **Top-down Modulation (Outward Propagation):** The CB issues general commands to it children blocks, and hierarchially passes to every block. While a block's PNU makes independent decisions, these decisions are modulated based on its parents' overarching signals, ensuring harmony in the blob's behavior.

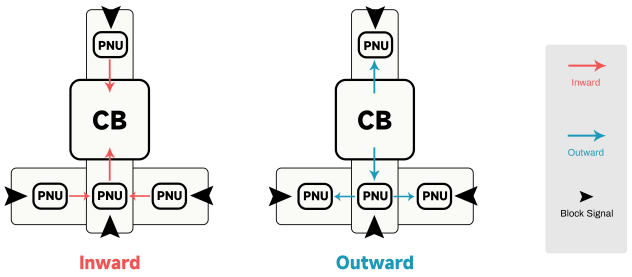

Fig. 4: Inward & Outward Propagation

Figure 4 shows the general information flow in both Inward Propagation and Outward Propagation on a single blob.

Advantages:

- **Decentralized Processing:** Like an octopus, where each tentacle can act semi-independently but is influenced by a central brain, the blob benefits from a blend of localized decision-making and coordinated control.
- **Adaptability:** If a block or limb gets mutated or is non-functional, its corresponding PNU gets impacted, but the overall neural architecture remains resilient. The CB can adjust its strategies based on inputs from functioning PNUs.
- **Evolutionary Fitness:** In the world of virtual evolution, Octopus Net's modular design ensures that mutations affecting one part don't lead to the complete breakdown of the entire neural system, offering better survival and adaptability chances.

A. Flat vs. Hierarchical: A Comparative Discussion

When I ponder the design of Neural Processing Units (NPU) in the blobs, two structural possibilities emerge: maintaining a hierarchical tree structure or adopting a flattened approach where all NPUs directly interface with the Central Brain (CB). Both designs have their own merits and challenges:

Hierarchical:

Strength:

- Mutation Resilience: This design showcases superior resilience to structural mutations. Whether a limb is gained or lost, the CB remains unaffected in terms of its structural integrity and functioning.
- Harmonized Decision-Making: The hierarchical structure supports synchronized decision-making. Each node emits general commands to its immediate descendants. As a result, the CB can release broad directives without being entangled in the intricacies of specific block levels.

Challenges:

- Signal Distortion: As signals traverse a labyrinth of nodes before reaching the CB, they can get distorted. This limits the CB's ability to accurately gauge stimuli from distal blocks, potentially making the blob less responsive to subtle, yet vital cues.
- Processing Complexities: Given the intertwined dependencies within NPUs, some function almost as mini-brains, reliant on outputs from other NPUs. This can complicate parallel processing efforts, delaying responses.

Flat:

Strength:

- Simpler Parallel Processing: Absence of internal NPU dependencies enables simultaneous processing of all NPUs. This can lead to faster reactions and adaptability.
- Clearer Signal Transmission: Direct connections to the CB eliminate the long circuitous paths of the hierarchical setup. This means that the CB receives more pristine, less-distorted signals, enhancing its sensitivity to nuanced stimuli.

Challenges:

- Lack of Gradation: While signals are clearer, there's a potential loss in the depth and gradation of information. Everything is relayed directly to the CB, which could overwhelm it with a barrage of data.
- Inconstant CB structure: Due to the direct linkage, the BC's shape and size changes while morphological mutation happens, which might leads to a wide disable of blocks.

In conclusion, the Octopus Net offers a harmonious blend of decentralization and coordination, drawing inspiration from nature's evolutionary marvels and tailoring it to the unique requirements of the virtual blobs.

While in this situation, I finally choose to use the hierarchical structure since there is a higher prority of mutation resilience for training. Keeping a relatively stable CB structure is essencial to the project regards of other benefits provided by flat design.

B. Inward & Outward Propagation

Given the selection of a hierarchical neural network model, all illustrations within this section pertain to the hierarchical architecture.

As previously highlighted, each Peripheral Neural Unit (PNU) is adept at both inward and outward propagation—capabilities beyond the scope of a singular neural network. Consequently, every PNU, representing the smallest independent neural network unit, houses two distinct networks: Inward NN and Outward NN. Refer to the implementation details.

The Inward NN receives input from two sources: signals gathered from blocks and joints, and the outputs from the child nodes' Inward NNs. This network generates a singular output, a synthesized signal derived from all its inputs. The output's dimensionality is notably smaller than its input, as only crucial information is relayed to parent nodes. Should a block be equipped to process specific data independently, the Outward NN decides on the course of action without escalating every detail to higher-tier nodes or the Central Brain (CB).

Conversely, the Outward NN accepts input from its block and joint and amalgamates it with directives descending from its parent nodes. It produces a dual-faceted output: one segment directs the joint motor—serving as the blob's muscle—while the other conveys commands to its child nodes.

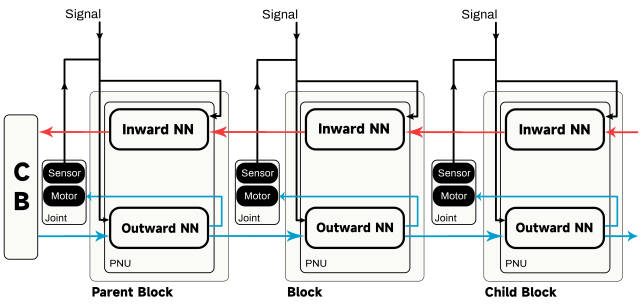

Fig. 5: Information transmission

Figure 5 delineates the intricate pathways of information transfer from one block to another. It's essential to note that, for clarity in the illustration, each block is depicted with a single child. However, this simplification doesn't typically reflect reality. In actual simulations, a block can have up to three child blocks. Consequently, the PNUs both receive and relay information to all its associated child blocks.

VII. Mutation

For an AI project that utilizes a genetic algorithm for training, the mutation strategy can never be more important. In the mutation process, the neural network's mutation and the gene's mutation are done separately.

A. NN Mutation

This network must possess mutable characteristics to facilitate the exploration of novel solutions. Nevertheless, caution must be exercised to ensure that mutations aren't overly aggressive, as excessive mutation may jeopardize inherent features vital for evolutionary progress.

Given the intricate and variable nature of a blob's neural network, mutation efforts are confined to the most granular units of the network. The procedure for mutation involves generating random values from a normal distribution with a mean of 0, which are then added to the respective weights and biases of the neural network.

To mirror the unpredictability of mutations in the natural world, certain constants—`MUTATE_NN_PORB`, `MUTATE_NN_WEIGHT_PROB`, and `MUTATE_NN_BIAS_PROB`—govern the likelihood of mutations occurring at the neural network, weight, or bias levels, respectively.

B. Gene Mutation

While neural networks follow a unified mutation approach, the mutation of morphology is decidedly more intricate. A blob's physical structure is made up of individual blocks, each of which can undergo size mutations. The joints, pivotal for connecting these blocks, can experience alterations in their movement limits. Additionally, during the mutation process, blobs have the potential to either gain or lose blocks.

- **Gain & Lose Limbs:** Blobs can either gain or lose up to one block during mutation. This dynamic allows blobs to experiment with varied bodily configurations.
- **Joint Mutations:** Blocks are interconnected via joints. During mutation, the range of joint movement can change, but it's always restricted to lie between 0 and 360 degrees.
- **Block Size Mutation:** Blocks can vary in size, provided the alterations remain within a predefined range. There's a caveat: size changes must not lead to internal structural conflicts. To avoid this, a maximum number of retry attempts is established. Another tricky problem for blocks' size mutation is that it affects the position of all connected child blocks. The direction in which a mutated block attaches to its parent is significant. If mutated, children blocks in one particular direction will shift twice as much as those in the other two directions. This cascading movement can, in turn, introduce further potential for structural conflicts. The implementation of solving those questions is partially located in struct BlobGeno since lots of mutations are directly modifying the blob's gene.

C. Synchronization

Ensuring synchronization between the neural network (NN) and the blobs, as well as the blocks within these blobs, is paramount. Given that blobs can gain or lose limbs during mutation, it's necessary to generate new neural networks during this process and subsequently remove the outdated ones. Additionally, as neurons are associated with blocks based

on their indices, special measures must be put in place. This prevents disruptions to the indexing caused by the addition or removal of neural networks.

The intricacies of synchronization are encapsulated within the sync_mutate function. While this function is succinct, it houses complex operations.

VIII. Training

Due to the different tasks the virtual creatures need to do, different training strategies can be designed.

Distinct tasks can demand vastly different training approaches. In the design, all training tasks are modular, allowing for seamless transitions between them by merely modifying the configuration file.

To achieve this flexibility, the BlobControlPlugin is introduced. This essential component orchestrates all aspects of the application, encompassing resources, systems, and the sequence of function executions (determining which functions execute first, and which ones can run concurrently).

As with many genetic algorithms, the training process incorporates tournament selection. The pivotal question is: how do we execute this? And how do we determine if one blob is superior to another?

In the tournament selection process, I combine two metrics for decision-making: a custom metric and the crowding distance.

The custom metric is tailored based on the specific objective of the training. For instance, in the experiment where blobs were trained to navigate underwater, the cumulative distance traveled served as the primary metric. This encouraged the blobs to cover greater distances.

Depending on the task at hand, this custom metric can be redefined to align with the desired outcomes.

Crowding distance gauges the similarity of an individual in relation to the broader population. A larger crowding distance indicates that the individual is markedly different from the majority. Applying crowding distance as a measurement can ensure the diversity of the population, which can prevent the solution from falling into a local solution.

In the framework, the crowding distance for a blob is determined using the tree edit distance (TED) for its gene since the blobs gene are represented by a QuadTree. The function document can be found here. It is a dynamic programming approach.

Unlike widely used genetic algorithms such as NSGA-II or NSGA-III, which employ crowding distance to maintain population diversity, my approach places greater emphasis on crowding distance, as I do not implement non-dominance selection.

At the conclusion of each epoch, two tournaments (the custom metric tournament and the crowding distance tournament) operate concurrently, producing two sets of survivors. The final set of survivors is derived from a random selection among these candidates. Individuals who rank higher in each tournament have an increased likelihood of being chosen. Notably, if an individual emerges as a survivor in both tournaments, its chances of selection in the final round are significantly amplified.

The constant `HYBRID_RATE` determines the balance between the custom metric tournament and the crowding distance tournament. This flexibility empowers users with more control over the training process.

IX. Conclusion

In conclusion, a brand-new approach to evolving virtual creatures and a highly flexible morphological bonded neuron network structure is proposed.

Since it is only a capstone report to show the project structure, a more complete paper in terms of training results and references will be provided later in the future.

References

[1] K. Sims, "Evolving virtual creatures," in *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*, 2023, pp. 699–706.