Towards generating complex programs represented as node-trees with reinforcement learning

Andreas Reich and Ruxandra Lasowski

Hochschule Furtwangen University
reich@digitalmedia-design.com
ruxandra.lasowski@hs-furtwangen.de

Abstract. In this work we propose to use humanly pre-built functions which we refer to as nodes, to synthesize complex programs. As e.g. in the node-tree programming style of Houdini (sidefx.com), we propose to generate programs by concatenating nodes non-linearly to node-trees which allows for nesting functions inside functions. We implemented a reinforcement learning environment and performed tests with state-of-the-art reinforcement learning algorithms. We conclude, that automatically generating complex programs by generating node-trees is possible and present a new approach of injecting training samples into the reinforcement learning process.

Keywords: Neural program synthesis; node-trees; machine learning; reinforcement learning; supervised learning; sample injection

1 Introduction

Many modern computer programs, especially in the 3D sector, offer users to interact with their software via so-called nodes, constructing so-called node-trees. Nodes are atomic parts of node-trees. Every node represents a function, a pre-programmed sequence of computer code visually. Node-trees are a visual high-level representation of non-linear sequences of nodes and are simpler to understand then regular code [1]. Due to the increasing prevalence of nodes in computer software, the idea arose to utilize machine learning to automatically generate node-trees with neural networks. Because nodes represent computer code, automatically generating node-trees with AI means automatically generating computer code with AI. Therefore, our approach is classified in the field of neural program synthesis.

The task of program synthesis is to automatically find programs for a given programming language that satisfy the intent of users within constraints [2]. Researchers like Bunel et. al. use supervised and reinforcement learning techniques to generate programs by concatenating low-level nodes linearly [3]. In contrast to their approach we propose to to generate programs concatenating high-level nodes non-linearly because this allows for more complex programs when using the same count of nodes. Our approach aims for automating the manual node-tree generation pipeline with AI that uses non-linear and high-level nodes. Using high-level nodes in the generation process is beneficial:

When compiled, all non-linear node-trees are transformed into linear sequences of code. The use of high-level nodes that consist of many low-level nodes is beneficial for certain use cases since more complex tasks can be solved with high-level nodes than low-level nodes (if the correct nodes are available) because more code is executed.

Most existing node-driven software solutions have a highly optimized set of high-level nodes that are software-specific. Consequently, users can perform a great variety

of tasks solely using few domain-specific nodes. Automatically creating and combining these specialized nodes with the help of AI would save users a lot of time and resources, while keeping full editability of generated node-tress.

Developing algorithms that search for valid node-trees in a search space (space of all possible connections) is challenging because the search space grows exponentially when adding nodes or depth and best regular program synthesis algorithms like Chlorophyll++ are currently able to find programs in a search space of 10^{79} [4]. This means if the aforementioned algorithm would work with nodes it could reliably find programs out of a pool of 100 individual nodes and a depth of 39 nodes. According to Bodík finding programs in a search space of 10^{79} is not sufficient to solve complex problems, e.g. like implementing the MD5 hashsum algorithm, located in a search space of 10^{5943} [4], since the algorithm works with low-level functions.

However, the generation of programs becomes easier if few high-level functions, that achieve the same result as many low-level functions, are combined. For instance, a program could just consist of 5 pre-assembled, high-level functions that themselves consist of hundreds of lines of code. Most algorithms could find such a program, just consisting of 5 nodes, with ease. The complexity of finding valid programs is constrained to the size of the search space. This means the complexity of finding node-trees built from high-level and low-level nodes are equal since the complexity is constrained to valid connections. Nonetheless, training an AI with high-level nodes is more costly computationally since more code needs to be executed.

This paper strives to point out the benefits of working with high-level over low-level functions. Performing some tests with the 3D Engine Blender and utilizing a node-based modeling tool showed that a specific node-tree in Blender with as little as 4 nodes already represents more than 1000 lines of computer code, whereas approaches from the Google Brain team are able to reliably generate programs with up to 25 lines of code with AI [5]. Figure 1 visualizes the difference between the count of nodes with the complexity of the resulting programs.

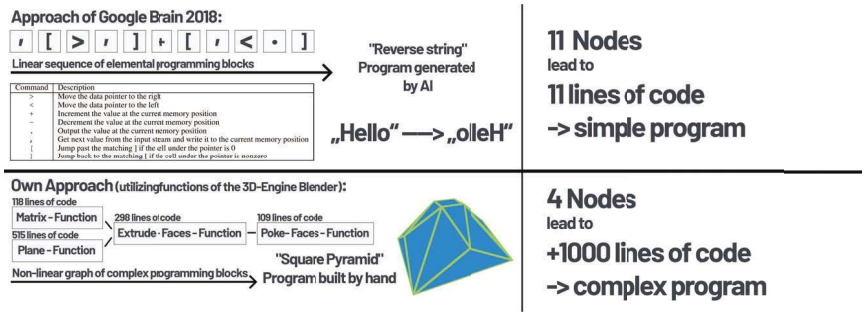

figure 1: complexity of high-level and low-level nodes

A possible real-world use case for using automatically synthesized node-trees could be the software Substance Designer with a completely node-driven workflow for texture and material creation. The Substance Designer documentation states that there are about 300 individual (high-level) nodes, yet also that most of these nodes only find application in rare, specific use cases [6]. Assuming one could build basic substance materials with

50 different nodes and a node-tree length of 40, the complexity of the search space would be $50^{40} \approx 10^{68}$ – less than what is currently possible in regular program synthesis. Nevertheless, a lot of the Substance Designer nodes have internal states that need to be adjusted, so the number of nodes necessary increases since changing an internal state can also be represented by the creation of a specific node.

Assuming one could create nearly all common Substance Designer materials by using all 300 nodes and 300 extra nodes that represent internal state changes, the creation of node-trees with lengths of 100 nodes leads to a search space of $600^{100} \approx 10^{278}$. Compared to solving MD5 with low-level functions and a search space complexity of 10^{5943} [4], a search space with the complexity of 10^{278} would still be way out of scope, yet more realistic to be achievable in the upcoming decades.

Besides Substance Designer, there are many node-based software solutions that offer a set of highly specialized nodes for different use cases. Making use of a predefined set of nodes is beneficial since most applications with node collections are capable of performing almost all tasks in a specific domain. We therefore identified certain areas where generating nodes with AI could be useful:

- Parameterization of 3D meshes (input: 3D mesh, output: node-tree approximating the modelling steps necessary to generate the mesh)
- Parameterization of photoscanned point clouds (input: point cloud, output: node-tree approximating the modelling steps necessary to generate the point cloud)
- Creation of 2D materials from images (input: 2D image, output: node-tree approximating a PBR-material)
- Reverse-engineering of functions and programs for optimization (input: program/function, output: optimized node-tree that approximates the program/function)

2 Purpose and Methods

We start our research for node-tree synthesis with the task of calculating a number with combinations of mathematical low-level nodes, e.g. plus and multiplication nodes, that are organized in a tree. In this way, we first heavily abstract more complex tasks to show feasibility. Figure 3 shows a node-tree reliably generated by our AI. We do not directly try to synthesize node-trees for complex graphic industry software because they require pipeline API's that could be implemented once the feasibility is shown. The purpose of our research is to show that we are able to automatically create node-trees from nodes with AI. Furthermore we want to point out the benefits of creating more complex programs by using high-level nodes instead of low-level nodes. Positive results could enable node-driven software solutions to utilize AI to automate processes that currently are solely performed by human users.

We use the qualitative method to investigate the state-of-the-art through literature research, especially concerning the comparison of publications and results of other researchers. Furthermore, we use the quantitative method to build experiments to prove that creating node-trees with supervised and reinforcement learning is possible.

To investigate the feasibility of generating node-trees with AI we implement an OpenAI Gym [7] reinforcement learning environment for automated node-tree generation and perform about 50 training sessions over the course of nine months. We use the reinforcement learning neural network architecture DDPG [8] and adjust its parameters.

To detect valid programs our actor explores a space of 8^{25} ($\approx 3.7 * 10^{22}$) in which it can perform 25 actions out of a pool of 8 individual actions. The following types of actions can be performed by the network:

- creating and automatically switching to new nodes
- changing internal states of nodes
- switching back to the previous node

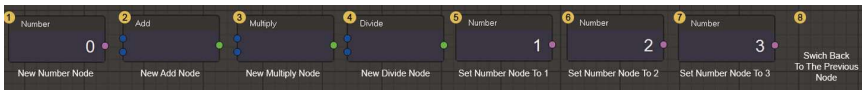

figure 2: All actions that can be performed by the network

The following states are observable by the network:

- specification/goal number
- relative distance to target
- number of nodes in total
- count of open connections
- type of the active node
- id of the active node
- internal state of the active node

Since our actor is not capable of finding valid samples without guidance we use an approach which we refer to as sample injection (figure 4). During exploration we randomly inject a valid action sequence (complete episode), which fulfils a given specification, into the current batch instead of the actors own chosen action sequence. This yields good results, since the actor learns from these optimal action sequences and is still capable of exploring the environment. This method is a combination of supervised- and reinforcement learning. The supervised samples are random sequences of valid connections between nodes that lead to a result (node-tree) which can be used as a specification for training. Therefore the actor can learn how to use which node in which context in order to fulfill a specification. Due to this on-the-fly generation process no data collection is necessary. Moreover the actor can learn how to use all available nodes in different contexts.

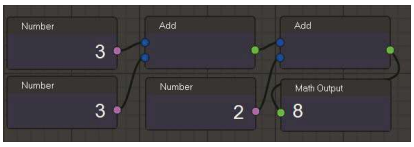

figure 3: node-tree found reliably in a search space of the size of 8^{25}
(8 individual actions and max. 25 actions)

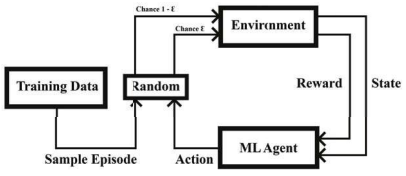

figure 4: sample injection

Because the search space in which valid node-trees can be found is very large our machine learning agent only receives sparse rewards and needs to learn from many samples. Therefore we decrease the learning rate to 10^{-5} and increase the replay buffer size to 10^7. For the most successful training sessions we train the network for ≈ 20 million steps. We use the following reward function, which rewards the actor for approximating a specification. Since we use mathematical nodes the goal number serves as specification and the current result of a node tree can be used to calculate the current distance which

is minimized:

$$\frac{((goalNumber - currentDistance) - (goalNumber - previousDistance))}{goalNumber} * incentive$$

DDPG uses the random Ornstein-Uhlenbeck process [9] which results in few valid programs to be found. We therefore complement the process with a supervised process to inject samples to enhance results. For most training sessions we inject samples 10% of the time instead of the action of the neural network. To reduce the impact of outliers we use the Huber loss [10] to enhance regression quality.

Figure 5 shows the network architecture of DDPG for the actor (left side) and the critic (right side). Figure 6 shows the parameters used for training. Figure 7 shows the pseudocode of DDPG with sample injection.

```
actor = Sequential()
actor.add(Flatten(input_shape=(1,) + env.observation_space.shape))
actor.add(Dense(16))
actor.add(Activation('relu'))
actor.add(Dense(16))
actor.add(Activation('relu'))
actor.add(Dense(16))
actor.add(Activation('relu'))
actor.add(Dense(nb_actions))
actor.add(Activation('linear'))
```

```
x = Concatenate()([action_input, flattened_observation])
x = Dense(32)(x)
x = Activation('relu')(x)
x = Dense(32)(x)
x = Activation('relu')(x)
x = Dense(32)(x)
x = Activation('relu')(x)
x = Dense(1)(x)
x = Activation('linear')(x)
```

figure 5: DDPG network architecture: left: actor, right: critic

```
memory = SequentialMemory(limit=10000000, window_length=1)
random_process = OrnsteinUhlenbeckProcess(size=nb_actions, theta=.15, mu=0., sigma=.3)
agent = DDPGAgent(nb_actions=nb_actions, actor=actor, critic=critic, critic_action_input=action_input,
                  memory=memory, nb_steps_warmup_critic=1000, nb_steps_warmup_actor=1000,
                  random_process=random_process, gamma=.99, target_model_update=1e-3,
                  supervised_learning=True, action_process=env.Supervised_Action_Process(env))
agent.compile(Adam(lr=.000001, clipnorm=1.), metrics=['mae'])
```

figure 6: parameters used for training DDPG

3 Results

Our experiments show that our trained AI agent is capable of generating certain node-trees reliably from low-level nodes when given a specification. Some generated node-trees are found with 100% accuracy in regard to a given specification whereas other node-trees approximate a given specification to some degree. Our results therefore show the feasibility of generating node-trees from low-level nodes with AI. Hence, We observe that more complex specifications, that need more nodes to be solved, lead to decreasing accuracy of our agent (table 1). This is due to the fact that the search space grows exponentially.

Algorithm 3 DDPG algorithm with sample injection

Provide valid action trajectories for injection learning
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  With the probabilty of 1 - Ɛ select an episode for injection learning
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    If episode is selected for injection learning, sample all episode actions from valid trajectory
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

figure 7: DDPG with sample injection pseudocode

4 Limitations

Due to time and hardware limitations we are not able to find programs in a search space of 10^{79} and therefore work with a smaller search space. Furthermore, we are only able to set up our experiments using low-level nodes. However, the size of the search space of high- and low-level nodes are equal when the same count of actions and nodes are available. Therefore, the results of our experiments can be transferred and therefore also apply to high-level nodes.

Table 1. Node-trees reliably generated by our algorithm in a search space with the size of 8^{25} and their accuracy in relation to a given specification (sorted by accuracy)

Specification	Exemplary action trajectory	Accuracy
1,2,3,4,5,6,9	add, num, set(2), back, num, set(2)	100%
10	add, multiply, num, set(3), back, num, set(3)	90%
7,8	add, add, num, set(3), back, num, set(3), back, num, set(2)	86%
11	add, multiply, num, set(3), back, num, set(3)	81%
12	add, multiply, num, set(3), back, num, set(3)	75%

5 Conclusions

In this work we use reinforcement learning in combination with supervised learning and the technique of sample injection to tackle the problem of solving the combinatorial search over node-trees that lead to the user specified result. We think that our technique could be used to ease many tasks in computer graphics like 3D mesh generation, VFX compositing, material generation and node-tree based scripting. We show that it is possible to automatically create node-trees from nodes with AI. Furthermore we point out that the size of the search space when using low-level nodes and the size of the search space when using high-level nodes are equal. This means that one can create more complex programs with the same amount of high-level nodes since high-level nodes consist of more lines of code. We therefore conclude that our technique will find application in node-driven software solutions and will leverage academia's interest in node-based neural program synthesis.

References

1. Blackwell, A.F.: Metacognitive theories of visual programming. IEEE symposium on visual languages (1996) 240–244
2. Gulwani, S.: Program synthesis. FNT in Programming Languages **4** (2017) 1–2
3. Bunel, R., Hausknecht, M., Devlin, J., Singh, R., Kohli, P.: Leveraging grammar and reinforcement learning for neural program synthesis. ICLR 2018 (2018)
4. Bodík, R.: Program synthesis. opportunities for the next decade. (2015)
5. Abolafia, D., Norouzi, M., Shen, J., Zhao, R., Le, Q.V.: Neural program synthesis with priority queue training. (2018)
6. Adobe: Substance designer - node library. (2021)
7. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym (2016)
8. Lillicrap, T.P., et al.: Continuous control with deep reinforcement learning. (2015)
9. Uhlenbeck, G.E., Ornstein: On the theory of the brownian motion. (1930)
10. Huber, P.J.: A robust version of the probability ratio test. The Annals of Mathematical Statistics (1965)