Machine Learning for Better Combinatorial Algorithm

Sajeeb Roy Chowdhury[1,2], Anton Betten[1]

[1]Department of Mathematics, Colorado State University , Fort Collins 80521

[2]Department of Computer Science, Colorado State University, Fort Collins 80521

Overview

- Problem Description
  - Definition
  - Illustration and Objective
- Motivation
- Existing Solution
- Our Solution
  - Brief Overview
  - Problem Translation
  - State-Action Graph
  - GPU Generated Schreier Tree
  - Graph Embedding on Schreier Trees
  - Objective Functions
  - Q Function and Embedding Parameterization
  - Complete Training Algorithm
- Training
- Testing
- Running Example
  - PGL(2, 11)
  - PGL(2, 23)
- Conclusion

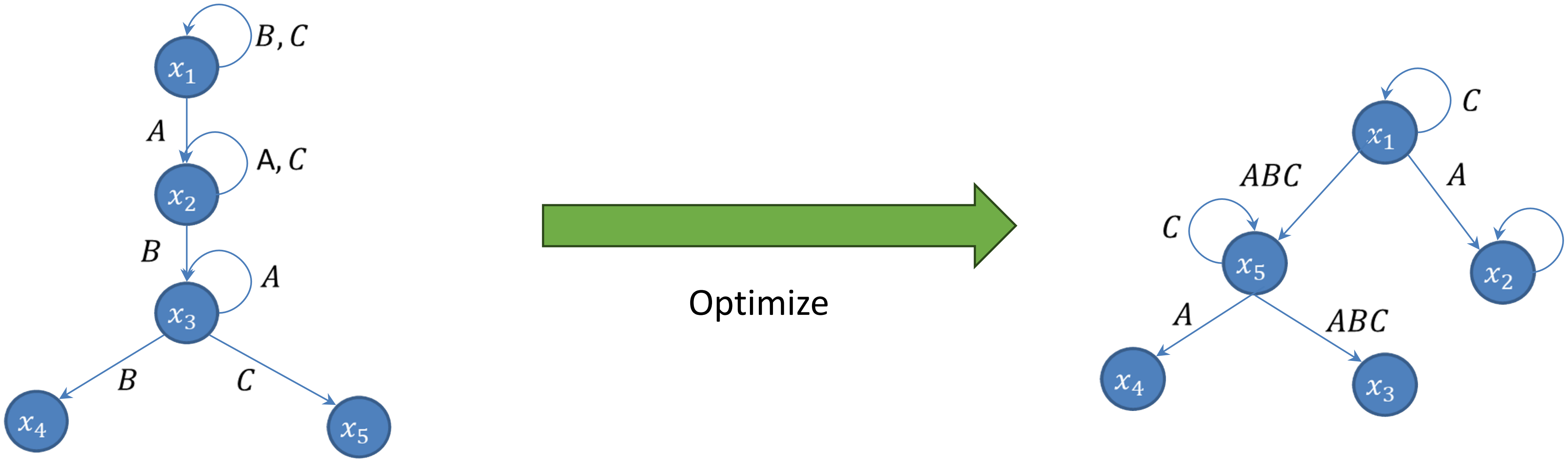
Problem Description: Definition

- Schreier trees are used to represent the orbits of group actions, and allow us to efficiently navigate large group using only the generators.
- These trees are generally "*tall and skinny*" by default, we aim to make these "*shallow and wide*".
- Shallow trees reduce the time to find the specific group element that maps one element of a coset to another.
- This reduction in time is attributed to path length: the longer the path length, the larger the chain of matrix multiplications that must be performed to find the mapping group element and vice-versa.
- Generating shallow Schreier trees requires replacing the generating set with better generators targeting the downstream task.
- Illustration:
  - Let G be a group with a generating set $S = < A, B, C >$
  - Let X be the set on which the G acts. The elements of X are: $X = \{x_1, x_2, x_3, x_4, x_5\}$
  - The following represents the orbits of G on X

Problem Description: Illustration and Objective

Average Word Length: 1.8

Average Word Length: 1.2



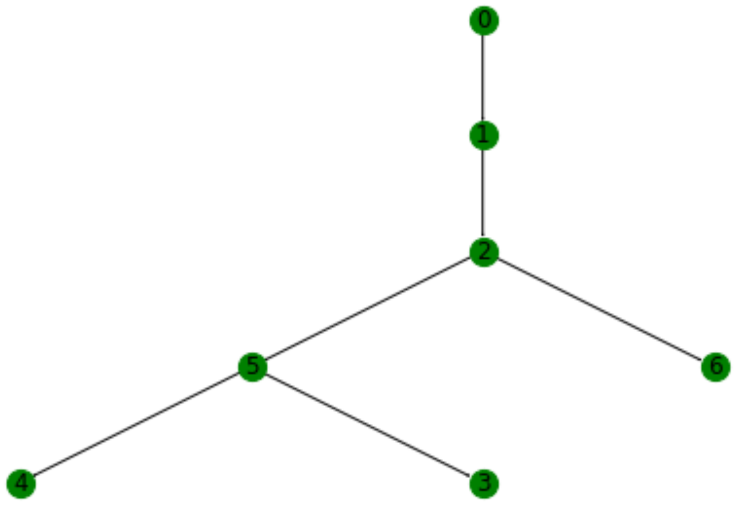
Non-optimal Schreier tree
due to the long node chain

Better Schreier Tree

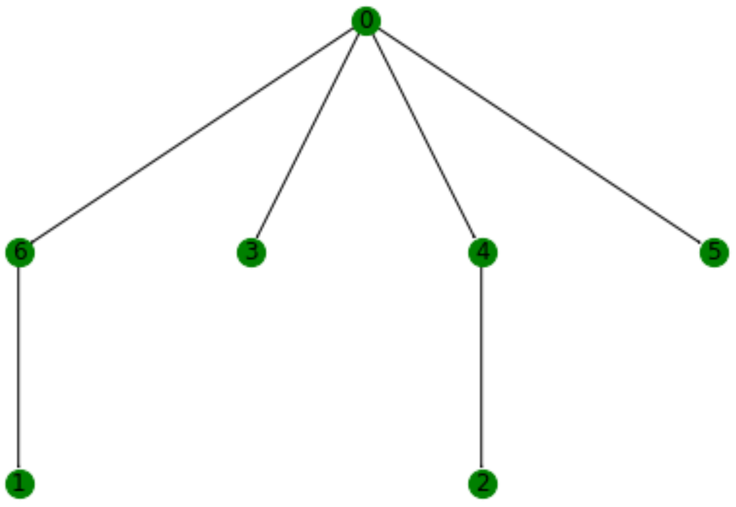
- Our objective is to transform the tree on the left to the tree on the right by reducing the height and increasing the width

Motivation

- Reducing the depth of the Schreier tree reduces the time taken to find the group element that maps a particular element of the coset to another.
- This is achieved by shortening the length of the matrix multiplication chain.
- Example with $PG(2,2)$ group with 4 generators, each of size 3x3:

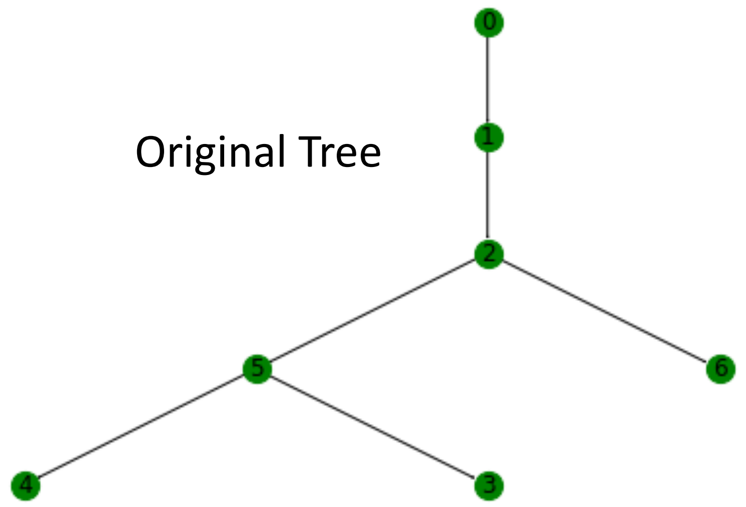


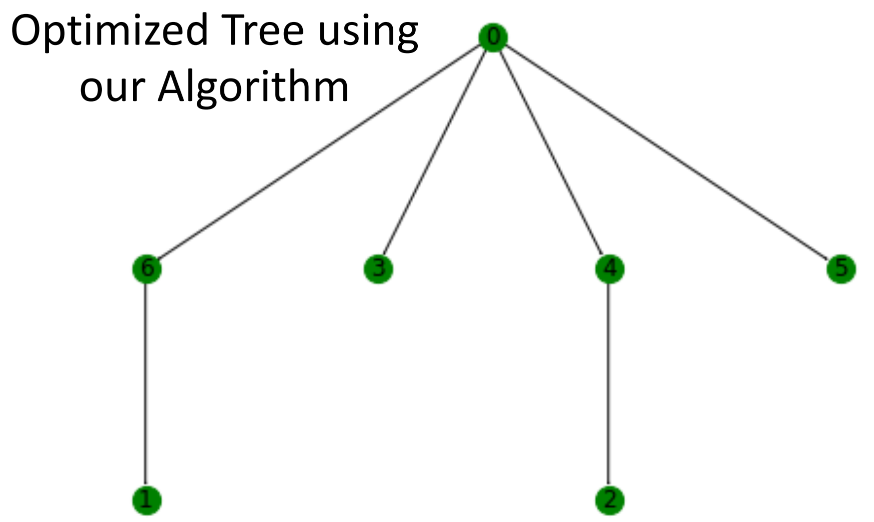
Original Tree



Optimized Tree using our Algorithm

Motivation





- For instance, it takes 336 computations to find the generator that maps node 4 to the root in the original tree, while it takes no computations to do the same in the optimized tree.
- This is because one has to multiply 4 matrices to get the group element in the original tree while no computation needs to be performed to do the same in the optimized tree.
- In the optimized tree, there are more nodes that needs no computation to be mapped to the root, whereas in the original tree, only a single node can be mapped to the root node without any computation.
- This observation attributes to the runtime performance gain when using the optimized tree over the original.

Existing Solutions

- As of now, there's only one algorithm for generating shallow Schreier trees. This is the Seress Shallow Schreier tree generator algorithm illustrated below:

Algorithm 1: Seress Shallow Schreier Tree Algorithm

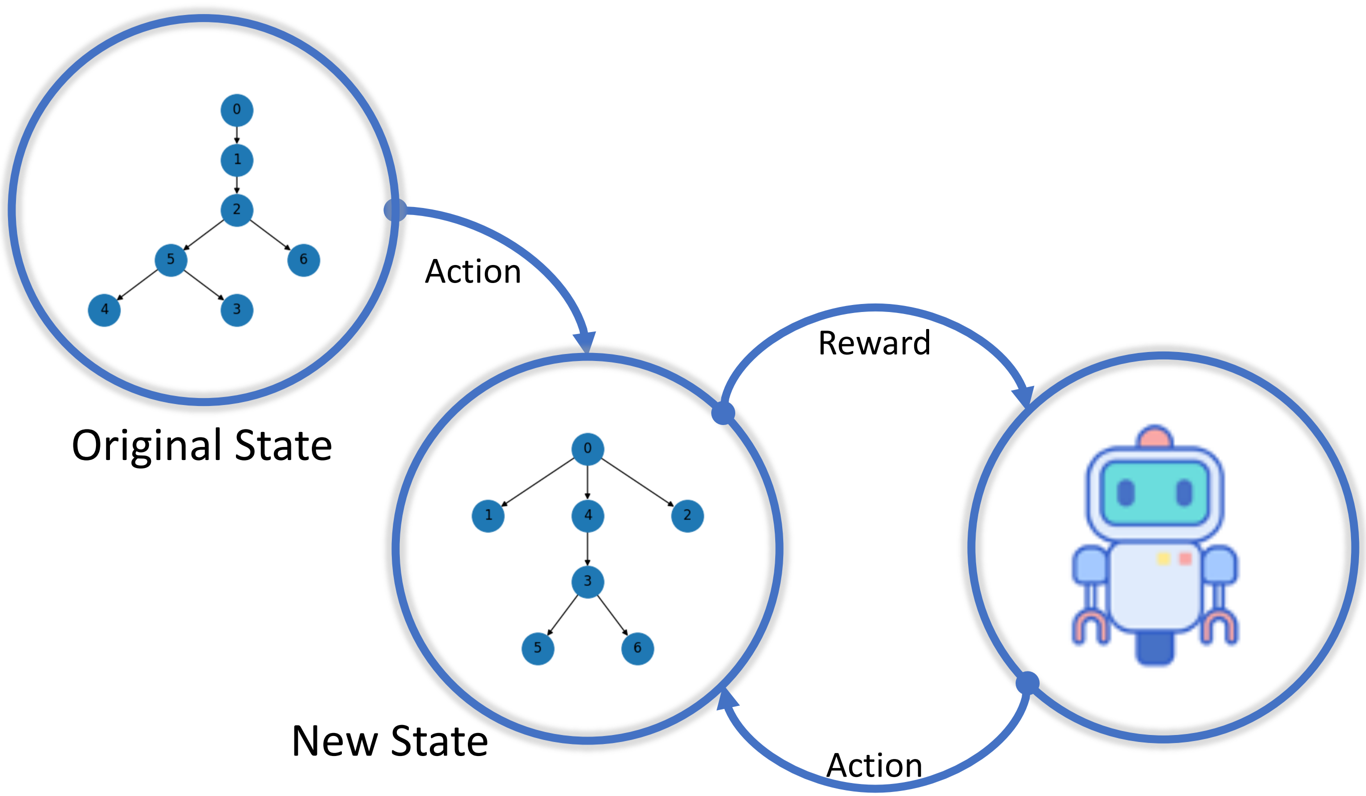
Input : Generating Set S, Coset Elements X, Depth Limited Schreier Tree Generator Function F_{sd}, Function to generate the complete Schreier Tree F_s, Transporter Function F_T, Function to get the set of nodes in a tree P

Output: New Generating Set, S'

$S' \leftarrow \{\}$;
$T \leftarrow F_s(S, X)$;
$P_T \leftarrow P(T)$;
$len \leftarrow |P_T|$;
$T' \leftarrow F_{sd,depth=|S'|}(S', X)$;
$l \leftarrow |P_{T'}|$;
while $l \neq len$ **do**
    $P_{T'} \leftarrow P(T')$ $e \leftarrow P_T \setminus P_{T'}$;
    $s \leftarrow F_T(P_{T'0}, e_0)$;
    $S' \leftarrow S' \cup s$;
    $T' \leftarrow F_{sd,depth=|S'|}(S', X)$;
    $l \leftarrow |T'|$;
end
return S'

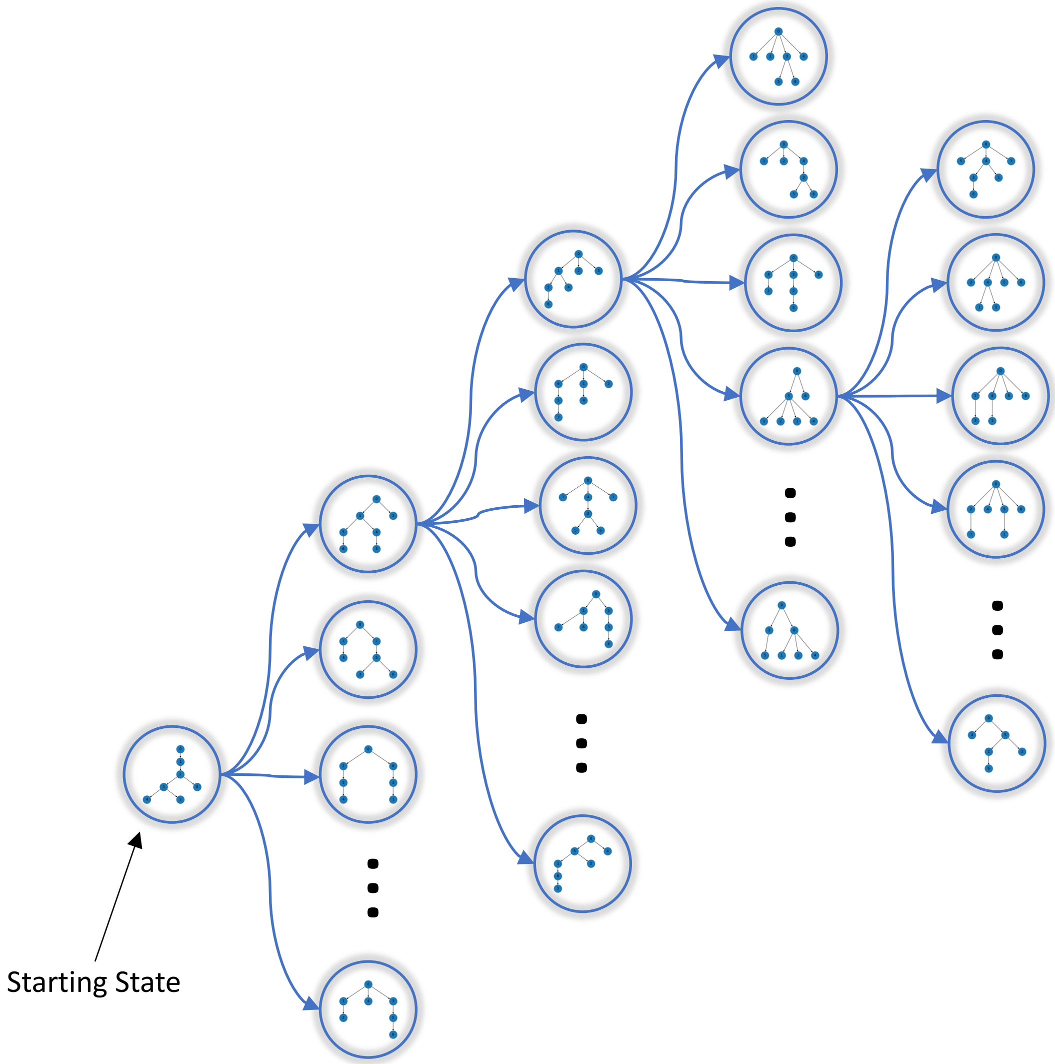
- The algorithm builds a new, hopefully shallow, Schreier tree by picking generators from the old tree
- It builds an empty tree, and adds the transporter of the first missing node to the generating set.
- It then builds a tree with this new generating set.
- This process of generating partial trees continues till we get a tree that has all the nodes as the original tree.

Our Solution: Brief Overview



- We rephrase the problem of finding Shallow Schreier Trees as a Reinforcement Learning Problem, where train an agent to heuristically search the group using the existing generators to find better generators that lead to shallower trees.
- We propose combining Graph Embedding with Deep Reinforcement Learning
- To accelerate the training process, we are using GPUs to compute our Schreier tree after every operation.
- Our terminal state is when all the generators are replaced with new generators

Problem Translation: State Action Graph



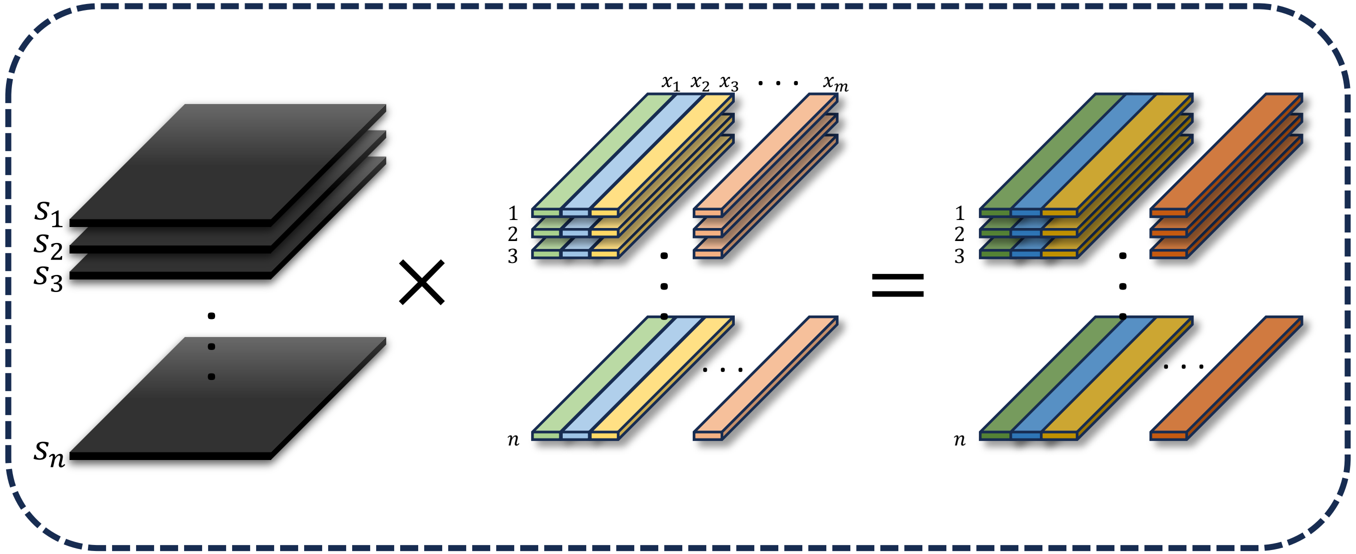
Partial State-Action graph for PG(2,2)

- Each state is a Schreier tree, and the action is a selected node
- For the problem set of PG(2,2) there are 4 generators. This implies that the depth of the state action graph can be at most 4+1=5.
- This is also the maximum number of steps the agent is allowed to take on the original tree. The game is terminated once all the generators are replaced.
- At every transition, at most $|S| - |adj(0)|$ actions can be taken. S is the set of nodes in the tree, and $adj(0)$ returns the set of nodes that are adjacent to the root node
- When a node is selected as the action for transition, it is marked with a special vector denoting this.
- Each transition from the previous state to the next state is done through selecting a node and connecting it with the root and adding the corresponding generator to the generating set.

Problem Translation: GPU Generated Schreier Tree

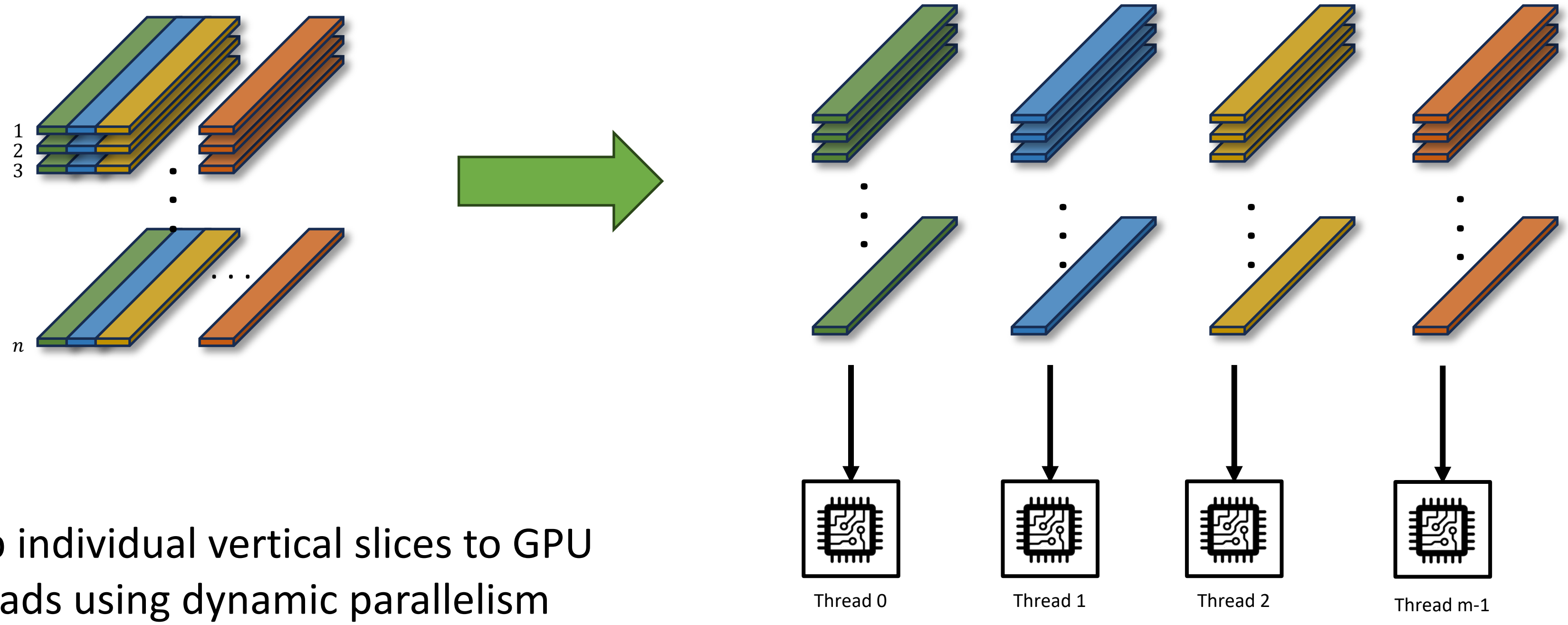
- Since generating Schreier Trees are expensive, we are re-implementing the generation algorithm targeting GPU-acceleration
- We use a two step solution to accomplish this

$$S = < s_1, s_2, s_3, \dots, s_n >$$
$$X = \{x_1, x_2, x_3, \dots, x_m\}$$



- Step 1:
  - Apply every generator to each element of the coset
  - This returns a 3D tensor containing the mappings

Problem Translation: GPU Generated Schreier Tree

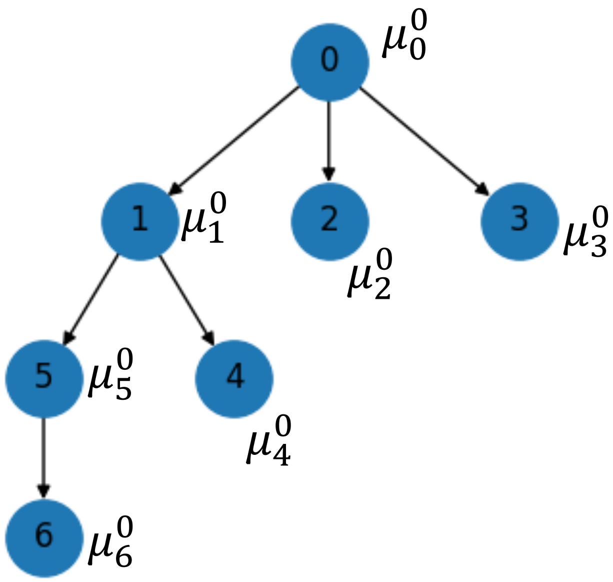


- Step 2:
  - Map individual vertical slices to GPU threads using dynamic parallelism
  - Each slice represents a point in the tree
  - Each thread gathers the descendants of each point and puts it in an adjacency list
  - This gives us the Schreier Tree

Problem Translation: Graph Embedding on Schreier Trees

- We use graph embedding to design a feature vector for each node for state space representation
- Initially each node in the tree is marked with a vector representing the embedding at $t = 0$: $\mu_v^0 = 0$
- At time $t + 1$, the embedding evolves as dictated by the following function:

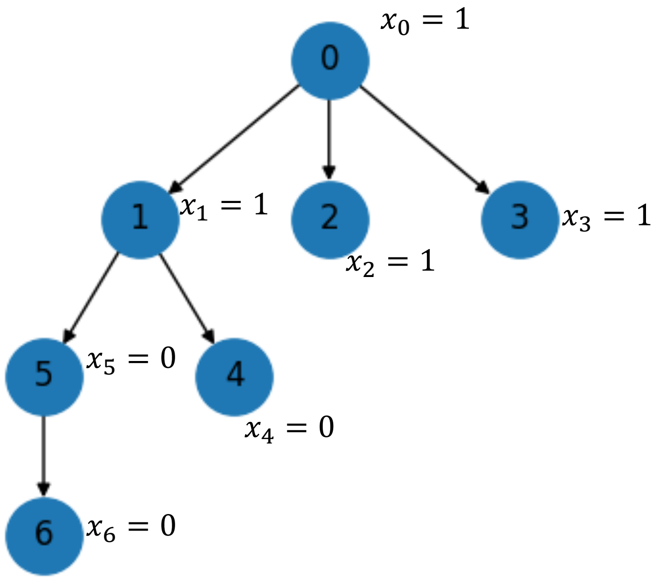
$$\mu_v^{t+1} = F\left(x_v, \{\mu_u^t\}_{u\in\mathcal{N}(v)}, \{w(u,v)\}_{u\in\mathcal{N}(v)}; \theta\right)$$



- At time T, the node embedding function takes the following form: $\mu_v^T = F\left(x_v, \{\mu_u^{T-1}\}_{u\in\mathcal{N}(v)}, \{w(u,v)\}_{u\in\mathcal{N}(v)}; \theta\right)$
- w is a function that returns the edge weight between two nodes. For our purposes, $w(x, y) = 1$
- Initially, the embedding iteration is run until the following condition is met: $\forall v \in V;\ \left|\left|\mu_v^t - \mu_v^{t+1}\right|\right|_2 < \epsilon$
- Once the above condition is met, the agent can start taking actions to minimize the tree

Problem Translation: Graph Embedding on Schreier Trees

- In order to mark the selected nodes by the agent used to perform the state transition, we use a special vector x_v



- Initially, the root node and all nodes that are the direct descendants are marked with $x_v = 1$

- After an action, once a node is no longer the direct descendant it is marked with $x_v = 0$

- After an action, if a node becomes the direct descendant of the root node, it is marked with $x_v = 1$

Problem Translation: Objective Functions

- We are using the following objective functions to reward or punish the agent

$$AWL(S) = \frac{\sum_{d=0}^{D} d \times K(d, S)}{N - 1}$$

S is the current state, AWL is the average word length, N is the number of nodes in the tree, and K is a function that returns the number of nodes at a current depth for a given state

- The reward function at state S, given an action v is:

$$r(S, v) = AWL(S') - AWL(S)$$

This reward function is defined as the change in the cost function when transitioning to a new state S' through action v.

- The policy function for deciding on the next action

$$\pi(v|S) = argmax_{v' \in \overline{S}} \hat{Q}(S, v')$$

$$\overline{S} = \{i | i \in S; x_i \neq 1\}$$

Problem Translation: Q function and Embedding Parameterization

- Parameterization of our embedding:

$$\mu_v^{t+1} = relu\Big(\theta_1 x_v + \theta_2 \sum_{u \in \mathcal{N}(v)} \mu_u^t + \theta_3 \sum_{u \in \mathcal{N}(v)} relu(\theta_4 w(u,v))\Big)$$

- The Q function in Deep Reinforcement Learning (DRL) framework is often a neural-network function approximator that approximates the Q table

$$\hat{Q}(S, v; \theta) = \theta_5^T relu\Big(\big[\theta_6 \sum_{\mu \in v} \mu_u^T, \theta_7 \mu_v^T\big]\Big)$$

Problem Translation: Complete Training Algorithm

Algorithm 2: Q-Learning Algorithm

Input : Generating set Z, Function T for generating the tree, Initial Parameters Θ, Set of cosets X, n for n-step Learning

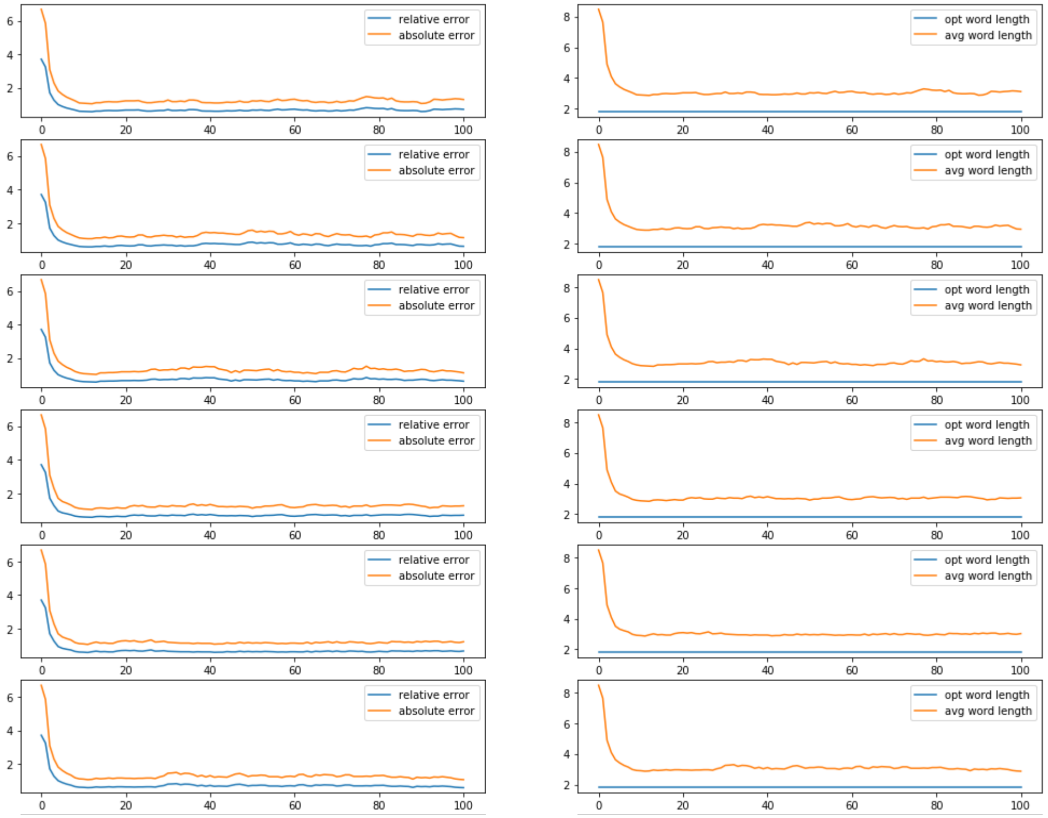
Output: Updated Parameters, Θ

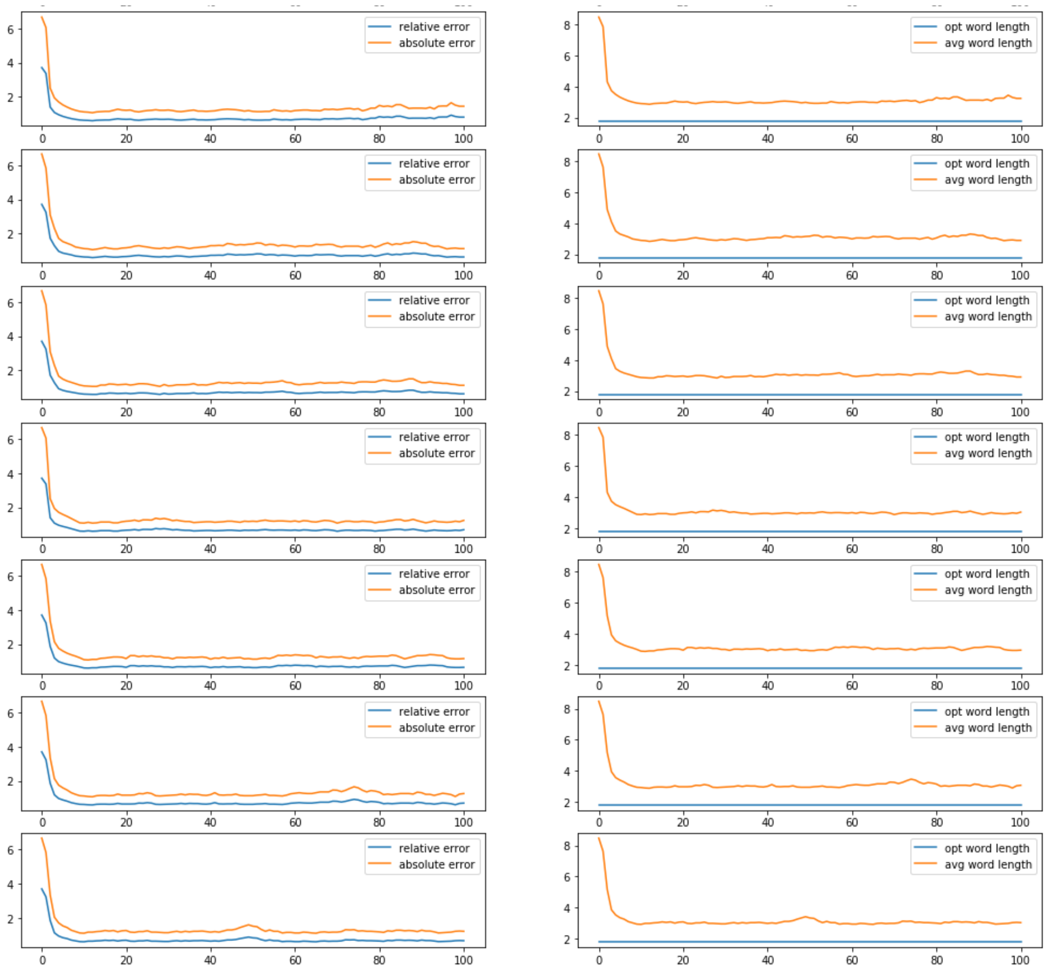
$G \leftarrow \{T(Z,x) | x \in X\}$;
$\epsilon \leftarrow 10^{-5}$;
for *episode* $e = 1$ *to* L **do**
    Pick a Graph, $g \in G$;
    $t \leftarrow 0$;
    $\forall v \in g,\ \mu_v^0 = 0$;
    $\forall v \in g; \mu_v^{t+1} = F\big(x_v, \{\mu_u^t\}_{u \in \mathcal{N}(v)}, \{w(u,v)\}_{u \in \mathcal{N}(v)}; \Theta\big)$
    while $\mu_v^t - \mu_v^{t+1} \geq \epsilon\ \forall v \in g$ **do**
        $t \leftarrow t+1$;
        $\forall v \in g; \mu_v^{t+1} = F\big(x_v, \{\mu_u^t\}_{u \in \mathcal{N}(v)}, \{w(u,v)\}_{u \in \mathcal{N}(v)}; \Theta\big)$
    end
    $E \leftarrow \{\}$;
    for $t = 1$ *to* $|Z|$ **do**
        $v_t = \begin{cases} random(v) \in \overline{S_t} & \epsilon_{Decay} \\ argmax_{v \in \overline{S_t}} \hat{Q}(S_t, v) & Otherwise \end{cases}$
        if $t \geq n$ **then**
            $R_{t-n,t} = \sum_{i=0}^{n-1} r(S_{t+i}, v_{t+i})$;
            $y = R_{t-n,t} + \gamma max_{v'} \hat{Q}(S_t, v', \theta)$;
            Add (S_{t-n}, v_{t-n}, y) to E;
            Draw a random sample from E;
            Update Θ by SGD with loss $L = (y - \hat{Q}(S_{t-n}, v_{t-n}))^2$;
        end
    end
end
return Θ;

- We are using n-step reinforcement learning to efficiently train our model.
- N-step Q learning looks at the current step and predicts the future reinforcement n steps ahead
- This is because we might encounter a state where the quality of the tree is worse than the previous state
- Since we do not have a terminal state, we set the constraint that $n \leq |S|$, where S is the number of generators in the generating set. We force the agent to terminate when all the generators in the generating set are replaced with new generators in succession

Training

- We trained our RL agent on PG(2,2) and PG(5,2) problem spaces. This gave us 46 GB of graph training data.
- Each row shows one unit of 10,000 iterations performed by the agent on the environment

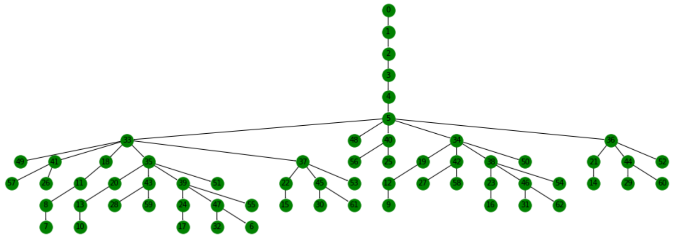




Testing

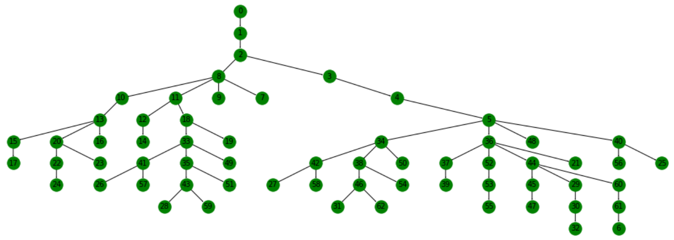
- The following represents an example from PG(5,2) testing set

Starting Tree



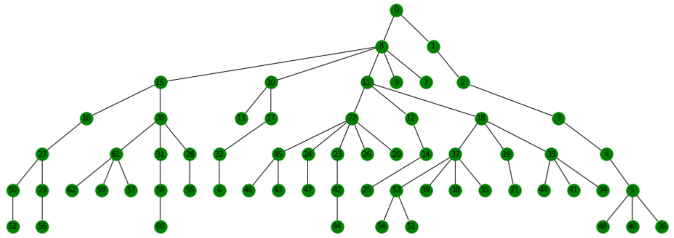
AWL: 8.47619
OWL: 1.79934
RE: 3.71072

1[st] Action



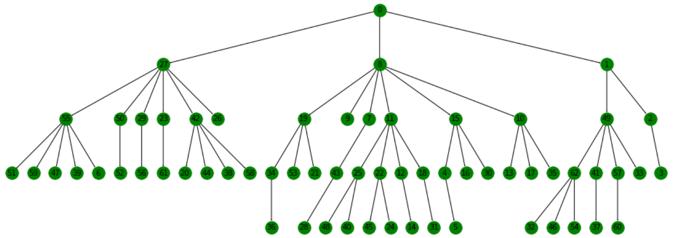
AWL: 7.63492
OWL: 1.79934
RE: 3.24318

2[nd] Action



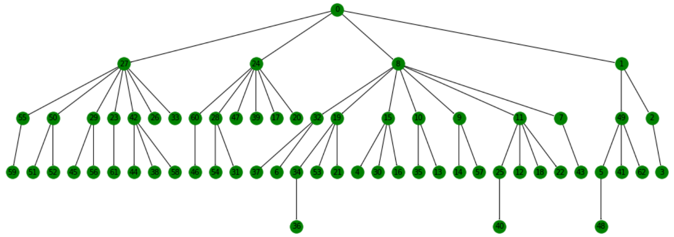
AWL: 5.15873
OWL: 1.79934
RE: 1.86701

3[rd] Action



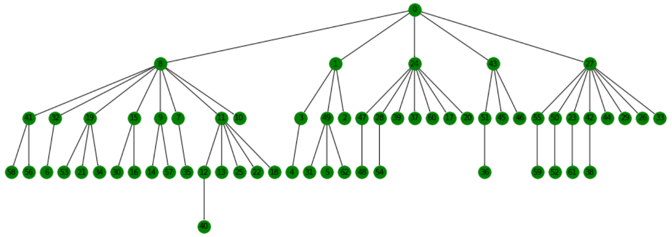
AWL: 3.85714
OWL: 1.79934
RE: 1.14364

4[th] Action



AWL: 3.52381
OWL: 1.79934
RE: 0.958389

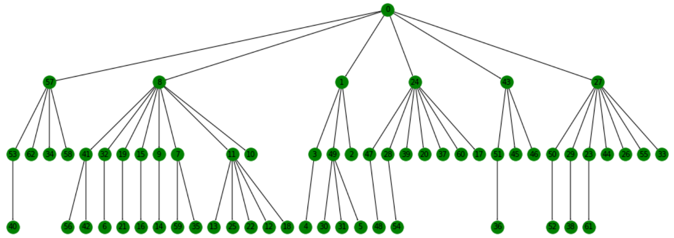
5[th] Action



AWL: 3.34921
OWL: 1.79934
RE: 0.861352

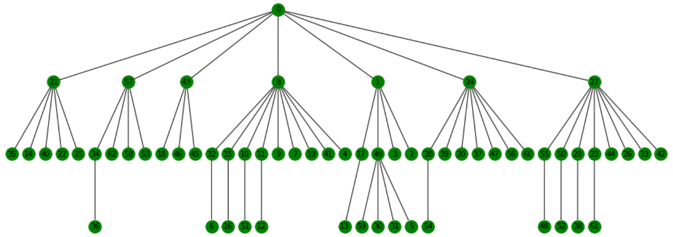
Testing

6th Action



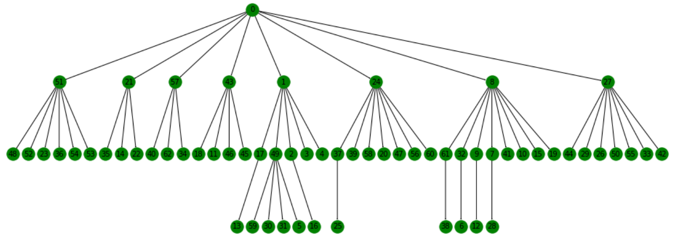
AWL: 3.25397
OWL: 1.79934
RE: 0.808423

7th Action



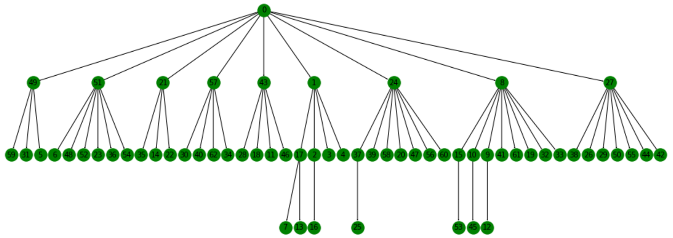
AWL: 3.09524
OWL: 1.79934
RE: 0.720207

8th Action



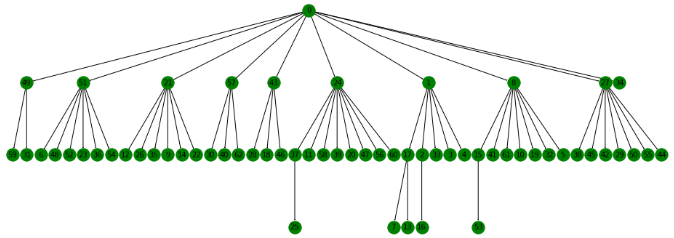
AWL: 3.01587
OWL: 1.79934
RE: 0.676099

9th Action



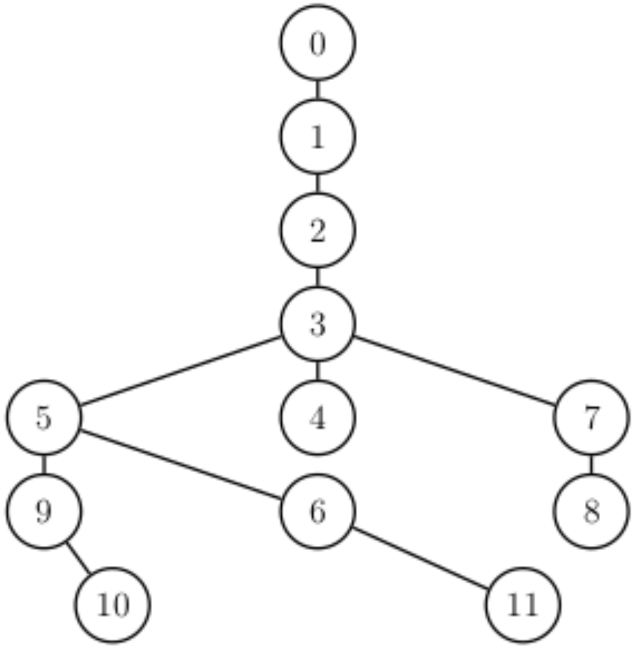
AWL: 2.93651
OWL: 1.79934
RE: 0.631991

10th Action

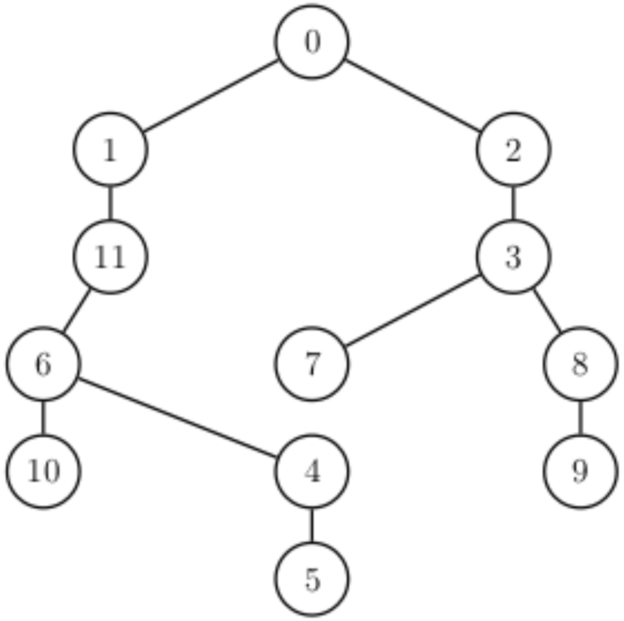


AWL: 2.88889
OWL: 1.79934
RE: 0.605526

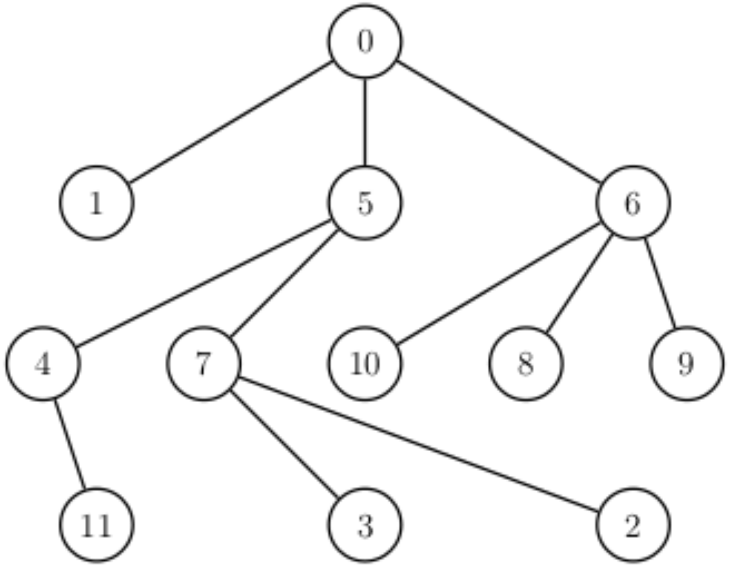
Running Example: PGL(2, 11)



Not-Optimal Tree
Size of Generating Set: 3
Average Word Length: 4.75

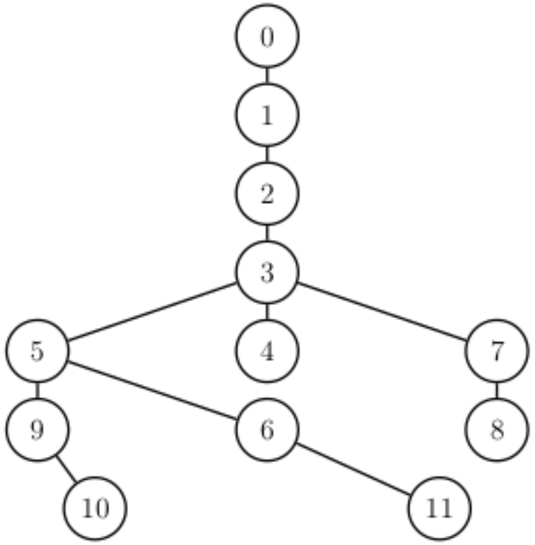


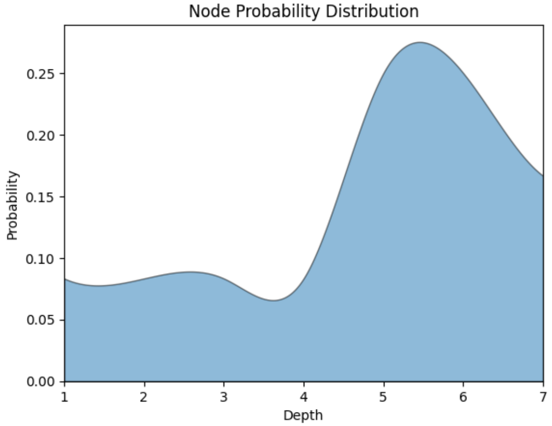
Seress Algorithm
Average Word Length: 3.667
Size of Generating Set: 4



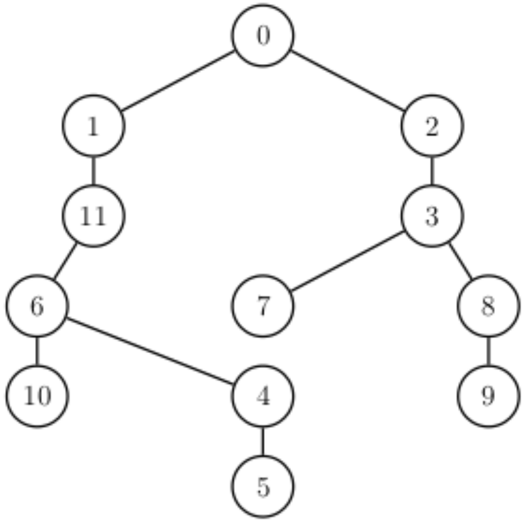
DL Heuristic
Average Word Length: 2.83
Size of Generating Set: 3

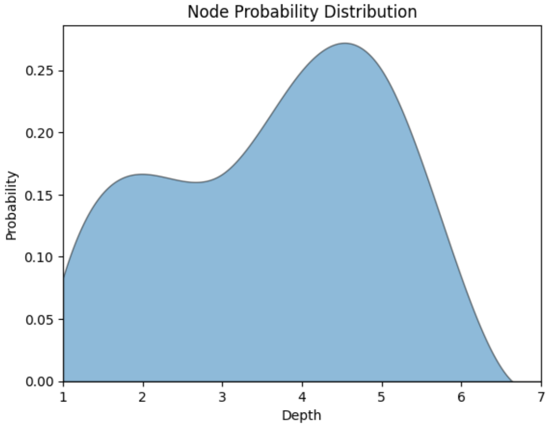
Running Example: PGL(2, 11) – Depth Probability Distribution



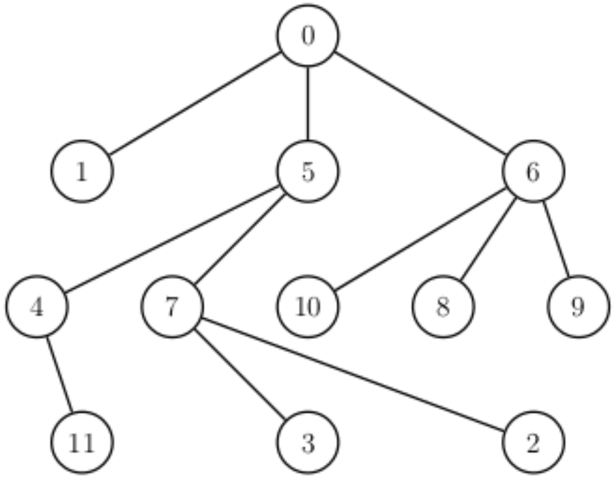


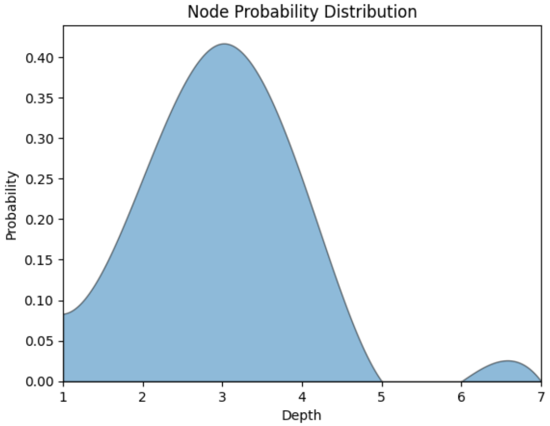
$\sigma = 1.83, \mu = 4.75, \sigma^2 = 3.35$





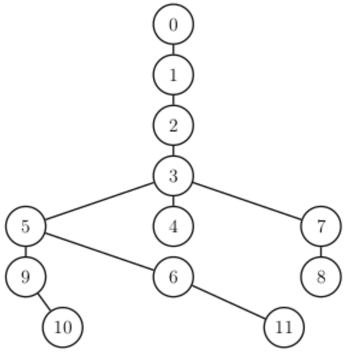
$\sigma = 1.43, \mu = 3.67, \sigma^2 = 2.06$



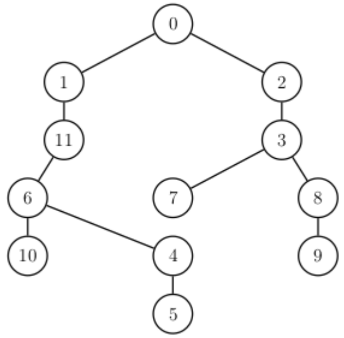


$\sigma = 0.89, \mu = 2.83, \sigma^2 = 0.81$

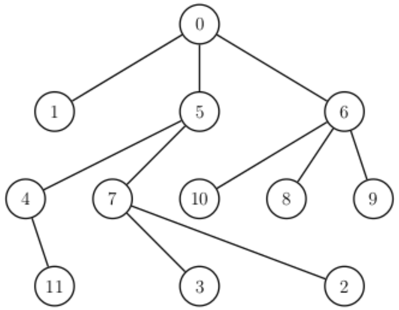
Running Example: PGL(2, 11) – Depth Probability Distribution



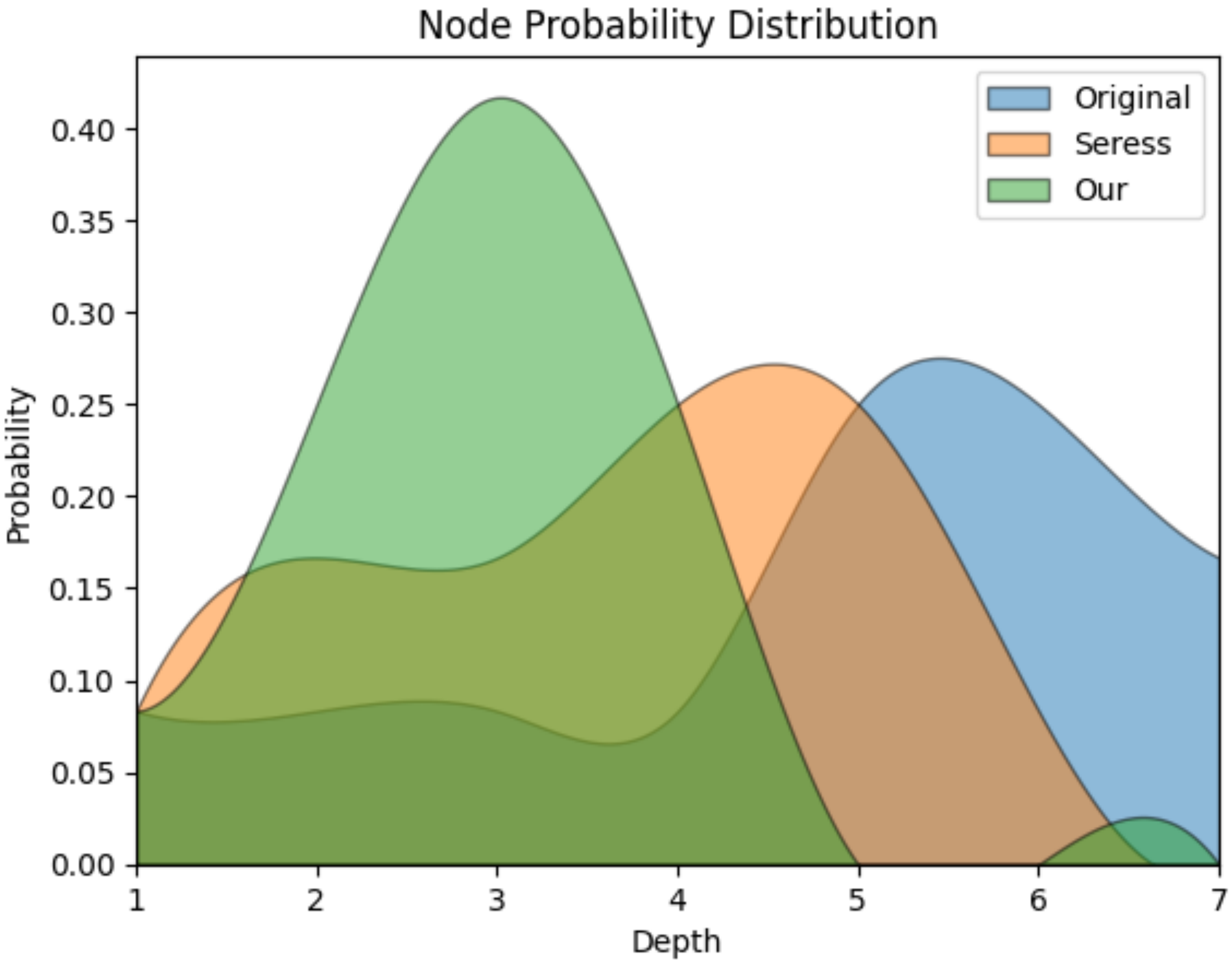
Original Tree



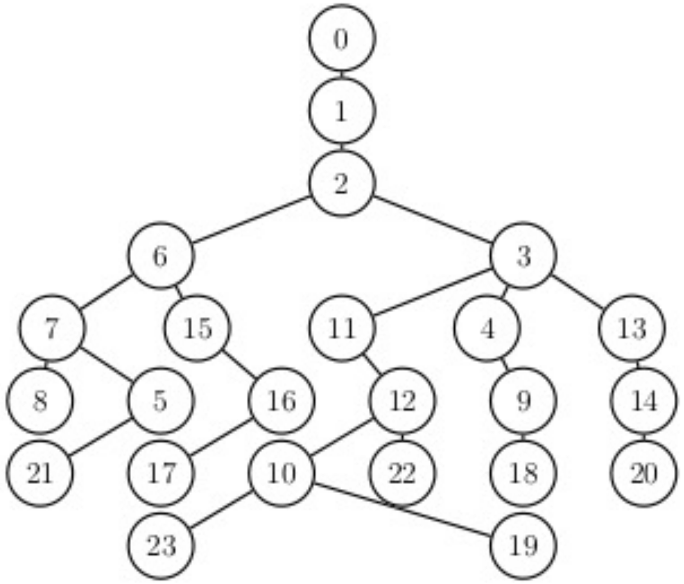
Seress Tree



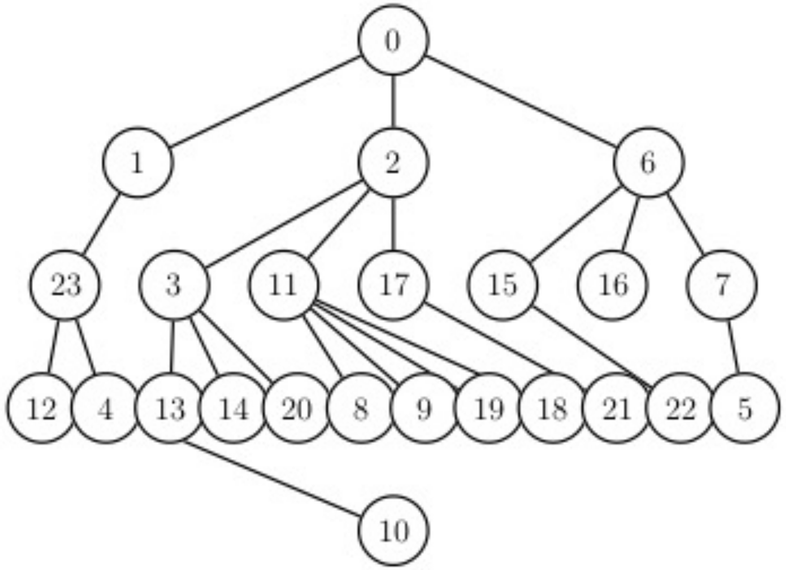
Our DL Heuristic Tree



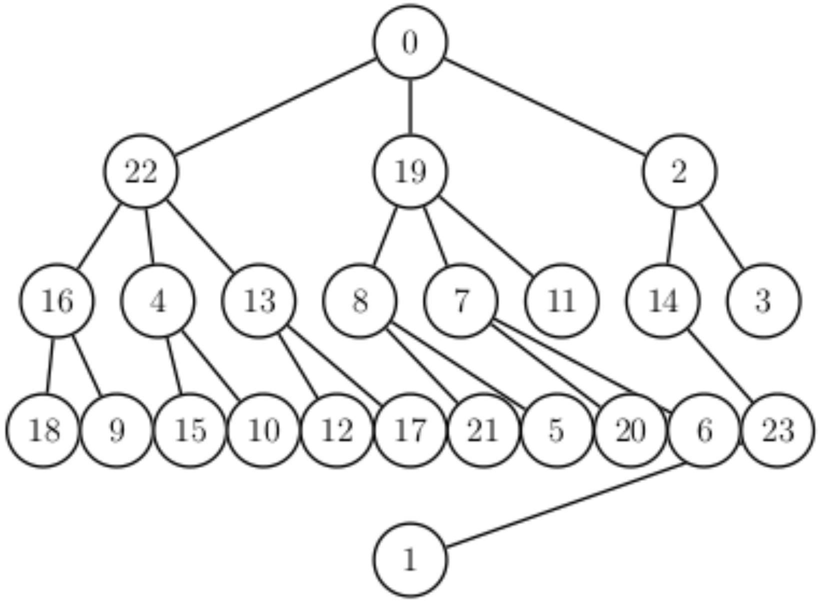
Running Example: PGL(2, 23)



Non-Optimal Tree
Size of Generating Set: 3
Average Word Length: 5.54167

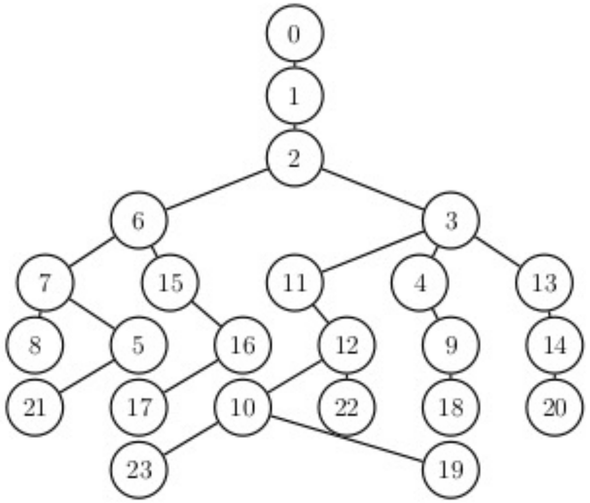


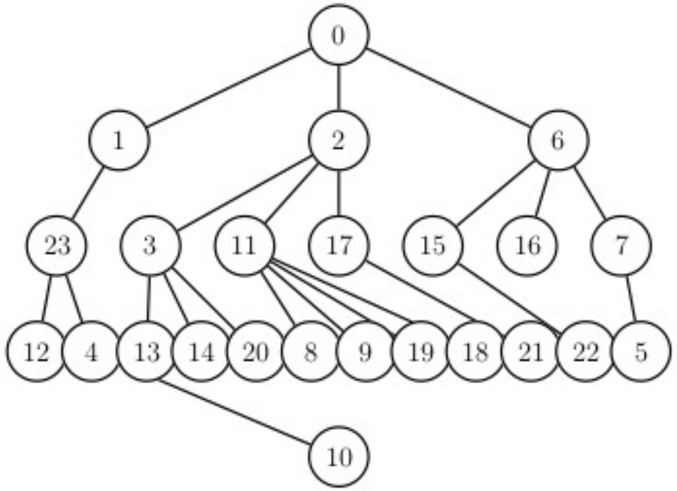
Seress Algorithm
Average Word Length: 3.375
Size of Generating Set: 6

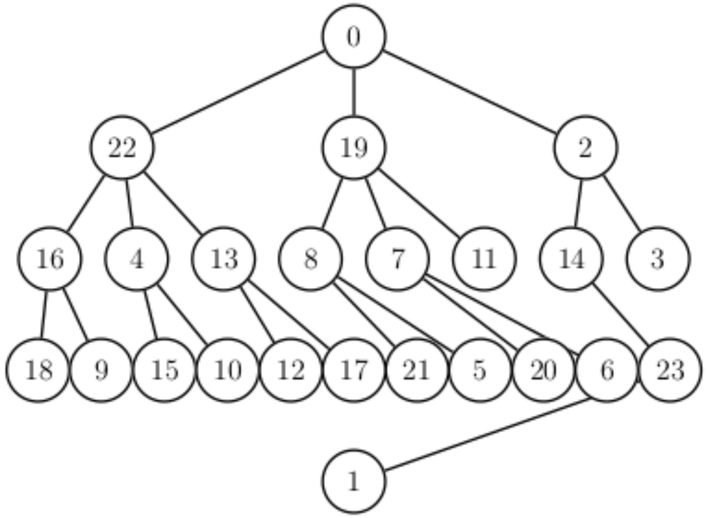


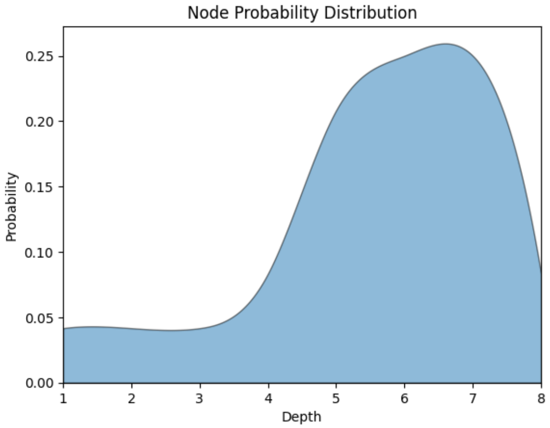
Machine Learned Algorithm
Average Word Length: 3.33
Size of Generating Set: 3

Running Example: PGL(2, 23) - Depth Probability Distribution

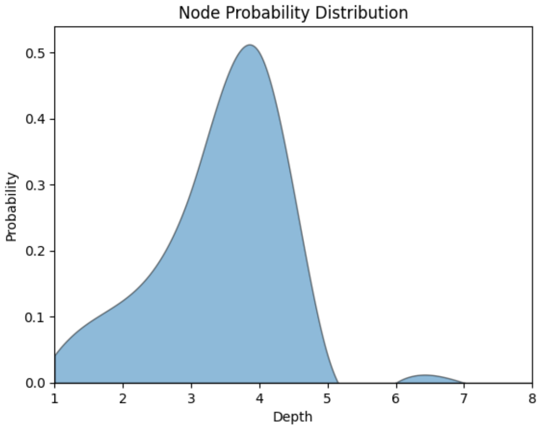




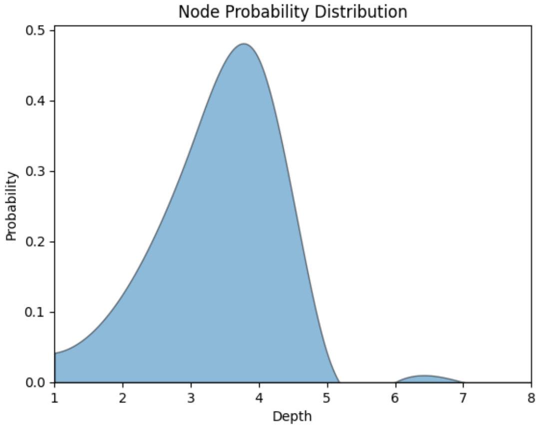




$\sigma = 1.73, \mu = 5.54, \sigma^2 = 3$

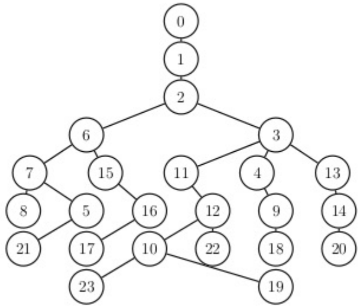


$\sigma = 0.9, \mu = 3.34, \sigma^2 = 0.82$

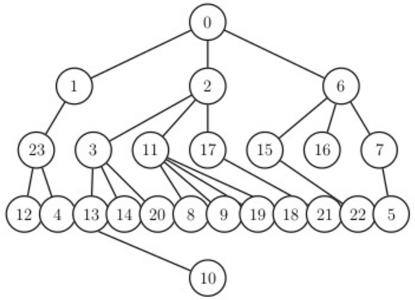


$\sigma = 0.9, \mu = 3.33, \sigma^2 = 0.81$

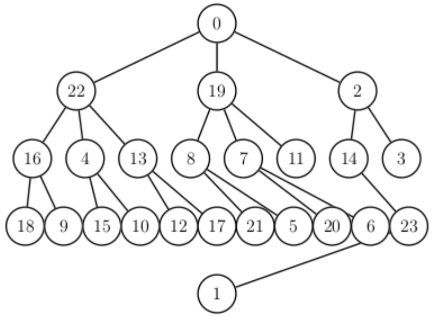
Running Example: PGL(2, 23) - Depth Probability Distribution



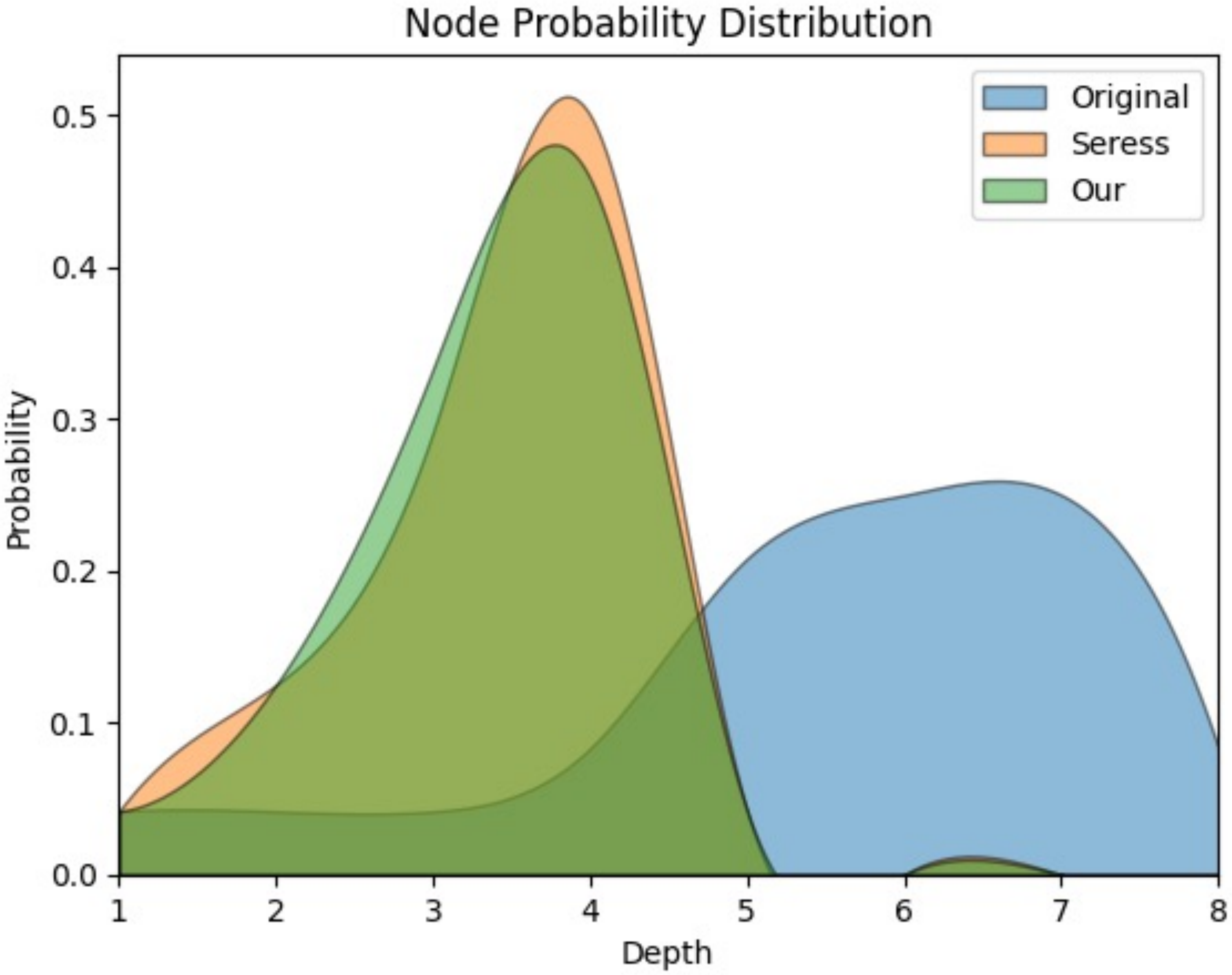
Original Tree



Seress Tree



Our DL Heuristic Tree



Conclusion

- In this talk, we have provided a framework for generating better heuristics for shallow Schreier trees
- Currently, there's only one algorithm for generating shallow trees
- The shallow Schreier trees generated by our model often outperforms those generated by the Seress algorithm
- From the real world running examples, we have discovered that generating shallow Schreier trees is equivalent to shifting the node probability distribution to the left.
- Shifting the node probability distribution to the left ensures that there are more nodes at lower depths of the tree.
- If the probability of discovering a node at a lower depth of the tree increases, this also decreases the path length between that node and the root node
- This decrease in path length leads to a reduction in the number of computations that need to be performed to find the transporter group element
- This increases the performance of the program using the Schreier tree data structure