Graph Neural Networks for Dynamic Abstraction Sampling

Vincent Hsiao [1], **Dana Nau** [1], **Rina Dechter** [2]

[1]University of Maryland, College Park
[2]University of California, Irvine

Abstract

Abstraction Sampling (AS) is an extension of importance sampling inspired by the concept of abstractions in automated planning. An important component of Abstraction Sampling is the abstraction function that determines how nodes are grouped together into abstract states. Existing abstraction functions based on context-based heuristics such as RandCB have been proposed, but may not be good for many problems. In this paper, we demonstrate that the problem of finding an optimal abstraction function can be framed as a variance minimization optimization problem. We propose a new method for learning abstraction functions parameterized using graph neural networks. Furthermore, we introduce a novel algorithm, Dynamic Abstraction Sampling that is capable of competitive results with respect to existing abstraction functions.

Introduction

A standard task in probabilistic inference on graphical models is the computation of the partition function, commonly denoted using the variable Z (Darwiche 2009) (Dechter 2013). Because this computation is frequently intractable, it is common to use stochastic methods such as Monte-Carlo sampling to generate estimates of the Z-value. Variance reduction techniques such as Importance Sampling (IS) (Gogate and Dechter 2011) are frequently employed to improve the precision of the Monte-Carlo estimates. In this paper, we are concerned with the task of generating samples for computing the partition function over search trees. Each estimate is generated through the construction of a sample *probe* down the tree. In IS, this probe consists of a path in the search tree, denoting a single configuration of all the variables in the tree. In Abstraction Sampling (AS), this probe can be a subtree, allowing for the representation of multiple configurations of variables in one probe.

For the purpose of this paper, we focus on regular search trees (OR-trees). In order to decide what nodes in the search tree should be included in a subtree probe, (Broka et al. 2018) introduces the notion of an abstraction function that separates nodes level by level in an search tree into abstract states. This process is inspired by abstractions in automated planning and can be traced to work by (Chen 1992) and (Knuth 1975). The idea is also similar to stratified importance sampling, a variance reduction technique in which the sample space is split into distinct groups. The abstraction functions introduced by (Broka et al. 2018) are context-based heuristics inspired by properties inherent to AND/OR-trees of graphical model. However, these abstraction functions can be improved for a given AND/OR-tree.

Contribution

Our contributions are as follows:

1. We propose a novel reformulation of learning abstraction functions as a variance minimization problem within reinforcement learning framework.
2. We demonstrate how RandCB (an abstraction function proposed by (Broka et al. 2018)) can be represented as a single layer graph neural network.
3. We propose an evolutionary algorithm for training abstraction functions parameterized using graph neural networks.
4. We devise a novel algorithm, termed GNN based Dynamic Abstraction Sampling that can be used to learn graph neural network abstraction functions on problems without a known ground truth.
5. We provide empirical results comparing GNN Dynamic Abstraction Sampling to the RandCB abstraction function and demonstrate that it can learn better abstraction functions.

Background

Graphical Models and Probabilistic Inference

A graphical model is a 3-tuple $M = (X, D, \Psi)$, consisting of a discrete variable set $X = \{X_1, X_2, \ldots, X_N\}$, a set of corresponding domains: $D = \{D_{X_1}, D_{X_2}, \ldots, D_{X_N}\}$ with $x_i \in D_{X_i}, \forall i$ (where the lowercase x_i indicates an assignment to the corresponding uppercase variable X_i) and a set of functions $\Psi = \{\Psi_1, \Psi_2, \ldots, \Psi_N\}$ (also known as factors). Each function Ψ_i is defined over a subset $X_{\Psi_i} \subseteq X$ denoted its scope and maps its corresponding subset of domain values D_{Ψ_i} (corresponding to X_{Ψ_i}) to a positive real number: $\Psi_i : D_{\Psi_i} \rightarrow \mathbb{R}^+$. Graphical models are used to represent probability distributions over configurations of the discrete variable set X, where for a configuration $\mathbf{x} =$

$(x_1, x_2, ..., x_N), x_i \in D_{X_i}$, the probability of the configuration $\mathbf{x}$ is:

$$\Pr(X = \mathbf{x}) = \frac{\prod_i \Psi_i(\mathbf{x})}{\sum_x \prod_i \Psi_i(x)} \tag{1}$$

where $\sum_{\mathbf{x}}$ denotes a summation over all possible configurations of X and the denominator $Z = \sum_x \prod_i \Psi_i(x)$ is a normalization constant also known as the partition function. The computation of the partition function is frequently intractable and it is common to estimate its value using stochastic Monte-Carlo methods.

Search spaces A graphical model can be transformed into an alternative representation known as an AND/OR or an OR search space (Dechter and Mateescu 2007). In this paper we focus on simple OR-trees. Given a variable ordering $o = (v_1, v_2, ..., v_N), N = |X|$, let the root of the search tree on the 0-th level be a dummy node. Each level i in the search tree corresponds to a variable $v_i \in o$ and each node in level i corresponds to a configuration to all variables up to v_i. Let $o_i = (v_1, v_2, ...v_i)$ be the i-th prefix of o. The edges of the search tree are weighted such that each edge into a node n_{v_i} has a weight $c(n_{v_i})$ defined to be the product of all functions $\Psi_i \in M$ that have scope X_{Ψ_i}, $s.t.\ X_{\Psi_i} \subseteq o_i\ and\ X_{\Psi_i} \not\subseteq o_j, j < i$. We define the value of a node n_{v_i} as the following recursive expression:

$$Z(n_{v_i}) = \sum_{n_{v_j} \in ch(n_{v_i})} c(n_{v_i}) Z(n_{v_j}) \tag{2}$$

where $ch(n_{v_i})$ is the set of children of the node n_{v_i}. The partition function of the graphical model M can be computed as $Z = Z(root)$ of the corresponding OR-tree search space (Dechter and Mateescu 2007).

Abstraction Sampling as Trajectories

In this paper, we consider Abstraction Sampling (Broka et al. 2018) on the special case of OR-trees. It would be straightforward to extend to AND/OR-trees. Given a graphical model $M = (X, D, \Psi)$, let $o = (v_1, v_2, ..., v_N), N = |X|$ be some elimination ordering and let $\mathcal{T}_o = (V, E)$ be the search tree generated from ordering o on M.

Abstraction Probe A probe $S_{\mathcal{T}}$ is a subtree of $\mathcal{T}_o$ such that:

1. It includes the root of $\mathcal{T}_o$ and if $n \in S_{\mathcal{T}}$, then it also includes the parent of n, $par(n) \in S_{\mathcal{T}}$.
2. Each $S_{\mathcal{T}}$ has an associated value $Z(S_{\mathcal{T}}) = Z(root)$ where

$$Z(n_{v_i}) = \sum_{n_{v_{i+1}} \in ch(n_{v_i})} c(n_{v_{i+1}}) \cdot Z(n_{v_{i+1}})$$
$$Z(n_{v_N}) = 1 \tag{3}$$

Abstraction Function Let d be a constant denoting the number of abstract states, $A = \{1, 2, \ldots, d\}$. We define an abstraction function $F_A : (M, S_{\mathcal{T}}, \mathcal{C}) \rightarrow A^{|\mathcal{C}|}$ as a mapping from the graphical model M, the current probe iterate $S^i_{\mathcal{T}}$, and a set of nodes $\mathcal{C}$ to a list of abstract states $\{s_1, s_2, ...s_{|\mathcal{C}|}\}$, one for each node $n \in \mathcal{C}$.

Abstraction Sampling In Abstraction Sampling, to generate an abstraction probe $S_{\mathcal{T}}$, we perform the following steps iteratively starting from $S^0_{\mathcal{T}} = \{S_{\mathcal{T},root}\} = \{S_{\mathcal{T},v_0}\}$ where v_0 is a dummy variable with child v_1.

1. Given $S^i_{\mathcal{T}} = \{S_{\mathcal{T},v_0}, \ldots, S_{\mathcal{T},v_i}\}$, let $ch(S_{\mathcal{T}}, v_i)$ be the set of child nodes to be partitioned in the next level v_{i+1}.
2. Given an abstraction function F_A, generate the assignment $S_A = F_A(M, S^i_{\mathcal{T}}, ch(S^i_{\mathcal{T}}, n_{v_i}))$. We place each node $n_i \in ch(S^i_{\mathcal{T}}, v_i)$ into abstract state $s_i \in S_A$.
3. For each non-empty abstract state $A_i \in A$, we perform importance sampling with heuristic $h(n)$ to generate a representative node n_{A_i} and assign a weight:

$$w_{A_i} = \frac{\sum_{n \in A_i} w(par(n)) g(n) h(n)}{w(par(n_{A_i})) g(n_{A_i}) h(n_{A_i})} \tag{4}$$

4. Let $S_{\mathcal{T},v_{i+1}} = \{n_{A_i}, \forall \text{ non-empty } A_i\}$, then $S^{i+1}_{\mathcal{T}} = S^i_{\mathcal{T}} \cup S_{\mathcal{T},v_{i+1}}$.

We repeat the above steps from $i = \{0, \ldots, N\}$. It is also possible to terminate this early if we know heuristic $h(n_{v_i})$ is exact at v_i and we set the value $Z(n_{v_i}) = h(n_{v_i})$.

Z-estimate Given a full probe $S_{\mathcal{T}}$, the Z-estimate of $S_{\mathcal{T}}$ denoted $Z'(S'_{\mathcal{T}})$, where all leaves are located at the lowest level N, is defined by:

$$Z'(n_{v_i}) = \sum_{n_{A_i} \in ch'(n_{v_i})} c(n_{A_i}) \cdot Z'(n_{A_i}) \frac{w_{n_{A_i}}}{w_{n_{v_i}}}$$
$$Z'(n_{v_N}) = 1 \tag{5}$$

where $ch'(n_{v_i})$ denotes the children of n_{v_i} in the pruned tree.

Optimality Let θ be some parameterization of a family of abstraction functions denoted $F_{A,\theta}$ and $\hat{Z}_\theta$ be a Z estimate generated using the abstraction function $F_{A,\theta}$. Our goal is to minimize the variance of our Z-estimates over the family $F_{A,\theta}$:

$$\theta' = arg \inf_\theta E[(\hat{Z}_\theta - Z)^2] \tag{6}$$

where $E[(\hat{Z}_\theta - Z)^2]$ denotes the expected squared difference between our Z-estimates and a ground truth Z value. Since AS is an unbiased sampling algorithm, minimizing this quantity is equivalent to minimizing the variance of our abstraction sampling procedure.

Reinforcement Learning

We formalize the task of choosing the optimal abstraction function as a reinforcement learning (RL) problem where an RL agent is tasked with learning a mapping $F_A : (M, S_{\mathcal{T}}, ch(S^i_{\mathcal{T}}, v_i)) \rightarrow A^{|ch(v_i)|}$ that minimizes the variance of the Z-estimates (Eq. 6). At each step i, given the current state $(M, S^i_{\mathcal{T}}, ch(S^i_{\mathcal{T}}, v_i))$, the task is to return an action $\alpha \in A^{|ch(v_i)|}$ which is the assignment of $ch(S^i_{\mathcal{T}}, v_i)$ to d abstract states.

Markov Process We denote the stochastic process $S_\mathcal{T}^i \to S_\mathcal{T}^{i+1}$ induced by our importance sampler, $\text{Pr}_{h,\alpha}(S_\mathcal{T}^{i+1}|S_\mathcal{T}^i)$ as a Markov process parameterized by the heuristic function h and abstract state assignment α.

Reward Function Let Z be the value of the partition function of M. At step N, a full probe $S_\mathcal{T}^N$ has been sampled having a Z-estimate of $Z'(S_\mathcal{T}^N)$. We assign a reward:

$$r_N = -\|Z - Z'(S_\mathcal{T}^N)\|^2 \tag{7}$$

to our learned probe. Since M is constant and $ch(S_\mathcal{T}^i, v_i)$ is a deterministic function of $S_\mathcal{T}^i$, the task is to learn a value function $V : S_\mathcal{T} \to \mathbb{R}$ specified by the following Bellman Equation:

$$V(S_\mathcal{T}^i) = \max_{\alpha \in A^{|ch(v_i)|}} \{\sum_{S_\mathcal{T}^{i+1}} V(S_\mathcal{T}^{i+1})\text{Pr}_{h,\alpha}(S_\mathcal{T}^{i+1}|S_\mathcal{T}^i)\}$$

$$V(S_\mathcal{T}^N) = r_N = -\|Z - Z(S_\mathcal{T}^N)\|^2 \tag{8}$$

where the value function specifies the negative expected variance at a given probe and can be computed recursively from the full probes. Unfortunately, due to the size of the state space of possible probes, this computation is intractable and we will need to rely on a different method for computing the optimal abstraction function F_A.

Evolution Strategies

First developed for continuous parameter optimization for the control of nonlinear systems, evolution strategies are a class of heuristic search procedures, similar to stochastic local search, that comprises a major subset of evolutionary algorithms. The general idea is an iterative process applied to a population of randomly generated individuals. Each individual is a set of continuous valued parameters θ and at each iteration (termed generation of the population), the fitness of each individual is evaluated, and the most fit individuals are selected from the current population and perturbed to generate the next population of individuals.

Evolution strategies appear to have some success in application to difficult RL environments (Salimans et al. 2017). More concretely, we propose the following evolution strategies approach to evolve a population of parameterized abstraction function:

1. Let $\{\theta_1, ...\theta_L\}$ be a population of L abstraction functions that map a context representation $(M, C(v_i), v_i)$ to an abstract state $A^{ch(v_i)}$. As an example, one possible parameterization is a neural network, where each θ_i would correspond to the weights and biases of a given network.
2. Evaluate each θ_i as an abstraction function F_{A,θ_i} for m rollouts obtaining estimates $\hat{Z}_{i,1}, ...\hat{Z}_{i,m}$ for each member of the population θ_i.
3. Evaluate the fitness $f(\theta_i)$:

$$f(\theta_i) = -\frac{1}{m}\sum_{j}^{m} \|Z - \hat{Z}_{i,j}\|^2 \tag{9}$$

   for each abstraction function θ_i where $\hat{Z}_{i,j}$ are the rollouts mentioned in the previous step.
4. Let the next population be composed of the following networks:
   - r (in practice $r = 0.2 * L$) individuals from the current population with the best fitness. These individuals are called parents.
   - $N - r - 1$ children $\theta_j = \theta_i + \mathcal{N}(0, \sigma), i \sim [1, r]$ where the parameters of a randomly chosen parent θ_i are perturbed with Gaussian noise $\mathcal{N}(0, \sigma)$ to produce a child θ_j. For numerical stability, we constrain parameter values to be within $[-1, 1]$.
   - A copy of the individual with the highest fitness through all population iterations (this is a convention known as elitism).
5. Repeat from step 1.

This algorithm is a standard $(r/1 + \lambda)$-ES scheme (Beyer and Schwefel 2002) with an additional slot for elitism (Baluja and Caruana 1995).

Dynamic Abstraction Sampling

So far, we have assumed that we know the ground truth partition function Z when we compute a reward/fitness function. Clearly, this is not the case when we would like to apply our algorithms to practical problems. Instead of using Z as our target value, we can use a running estimate Z_{est} of the ground truth as our target:

$$Z_{est} = \frac{1}{m \cdot L \cdot g}\sum_{k}^{g}\sum_{i}^{L}\sum_{j}^{m} \hat{Z}_{k,i,j} \tag{10}$$

where g denotes the generation of the evolution strategy and $\hat{Z}_{k,i,j}$ is the Z-estimate from the j rollout of the i member of the k-th generation of the evolution strategy.

Graph Neural Networks

Graph neural networks (GNNs) are a type of neural network architecture that has found recent success in its application to structured data. Compared to other architectures such as multi-layer perceptrons (MLPs), one of the primary advantages of a graph neural network is its flexibility in handling arbitrary relational data structures. Notably, unlike MLPs, a GNN does not need an input that is of a fixed length. Furthermore, the structure of the data directly impacts the computations performed in an GNN unlike in an MLP where such information is typically discarded. Additionally, GNNs are typically more efficient than MLPs for training since the weights in a GNN are shared across the nodes for an input graph. This advantage can also be found in convolutional neural networks (CNNs) and GNNs can be seen as a method of extending the local computations that are present in CNNs towards computation on arbitrary graph structures.

An important subclass of graph neural networks are aggregate-combine graph neural networks (AC-GNN) (Barceló et al. 2020). The AC-GNN consists of two operations, the graph aggregation operation, and the combine operation. To be concrete, consider a graph $G = (V, E)$, and let $N(v)$ denote the neighborhood of a vertex $v \in V$. Let each $v \in V$ be associated with some real vector $x_v \in \mathbb{R}^n$. For a given node v, we define the following:

- An *aggregation* operation is a function that maps a set of vectors to a single vector. Namely $Agg : \{x_j, j \in N(v)\} \rightarrow \mathbb{R}^n$. An example of an aggregation operation is the mean of a set of vectors:
$$Agg(v) = \frac{1}{|N(v)|} \sum_{j \in N(v)} x_j \tag{11}$$
Alternatively, this operation can be parameterized using a recurrent neural network (e.g. through an LSTM in GraphSage (Hamilton, Ying, and Leskovec 2017)).
- A *combine* operation maps the result of the aggregation operation and the current node's vector x_v to a real vector: $Comb : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$. Typically this is done using a simple feed forward neural network:
$$Comb(x_v, Agg(v)) = f(Cx_v + A[Agg(v)] + b) \tag{12}$$
where A and C are trainable weight matrices, b is a trainable bias, and f is some non-linear function (a trainable MLP or just a non-linear activation function such as ReLU).

In a graph neural network, each layer represents an aggregation operation followed by a combine operation that is computed for each node in an input graph.

Let $x_v^{(0)}$ denote the vectors associated with $v \in V$ in the input graph G. Let $x_v^{(i)}$ denote the vector associated with vertex v after the i-th layer of the graph neural network. A graph neural network with l layers obeys the following equation:

$$x_v^{(i)} = Comb^{(i)}(x_v^{(i-1)}, Agg^{(i)}(v)), \ \forall v \in V, \ \forall i \in [1, ..., l] \tag{13}$$

Additional Details

- *Loss Function*: The output of the final layer of a graph neural network with l layers is a vector $x_v^{(d)}$ for each vertex in the graph G. GNN loss functions can directly use this information for node-level loss functions (e.g. approximating node classifiers as in (Barceló et al. 2020)) or include an additional global aggregation layer for graph-wide loss functions (approximating a value function in (Ståhlberg, Bonet, and Geffner 2022)).
- *Global Readout*: In (Barceló et al. 2020), there is an additional component called a readout that aggregates all the nodes in the graph. The result of this readout operation is fed as an additional argument to the combine operation. This global readout improves the expressivity of a given GNN (Barceló et al. 2020) and allows them to represent a larger class of node classification functions.

For this paper, we are only learning the parameters for the combination operation. This means that each graph neural network is parameterized by $\theta = \{\theta^{(1)}, ...\theta^{(l)}\}, \theta^{(i)} = \{C^{(i)}, A^{(i)}, R^{(i)}, b^{(i)}\}$, where C, A, R, b are trainable weight matrices and biases.

Graph Neural Networks for Abstraction Functions

Our goal is to learn a graph neural network that approximates the optimal abstraction function over the family of abstraction functions parameterized by l layer GNNs as in Eq. 6. To describe our approach, we begin with the following two motivations:

- Consider a set of nodes: $ch(S_{\mathcal{T}}^i)$. Recall that at a given iteration i, an abstraction function maps the set of nodes $ch(S_{\mathcal{T}}^i)$ to a list of abstract states $\{s_1, s_2, ..., s_{|ch(S_{\mathcal{T}}^i)|}\}$. This can be thought of as a node classification problem.
- By definition, the assignment to the context of a variable uniquely determines the value of its rooted subtree (Broka et al. 2018) (Kask et al. 2020). For each node $n_j \in ch(S_{\mathcal{T}}^i)$, we can encode the context assignment into the structure of a bipartite constraint graph, similar to factor graphs for constraint satisfaction problems.

Given a set of nodes $ch(S_{\mathcal{T}}^i)$ for variable v_i with context $C(v_i) = \{v_j, j \in C(v_i)\}$ and domains $D_{v_j}, v_j \in C(v_i)$, we define a bipartite context graph as $G = (U, Y, E)$ where:

- U: $ch(S_{\mathcal{T}}^i)$
- Y: each vertex in $y_{j,a} \in Y$ corresponds to one value $a \in D_{v_j}$ for each $v_j \in C(v_i)$
- E: there is an edge from a vertex $\in U$ to a vertex $y_{j,a} \in Y$ if the assignment for the child node n_{v_i} for the variable v_j is a.

An Example Let $ch(S_{\mathcal{T}}^i) = \{n_1, n_2\}$ associated with v_i, where $C(v_i) = \{X_1, X_2\}$, and $C(n_1) = [X_1 = 1, X_2 = 2], C(n_2) = [X_1 = 1, X_2 = 1]$. The bipartite context graph is:

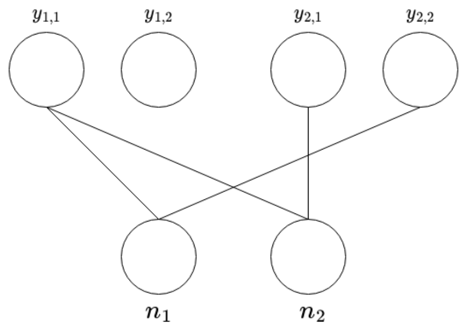

Figure 1: Example bipartite context graph for two child nodes.

We can now define the input and outputs of our graph neural network to approximate an abstraction function:

- *Input*: a bipartite context graph
- *Output*: a label for each node $u_{n_j} \in U$

Recall that in order to use a GNN, we need a vector $x_v^{(0)}$ associated with each vertex in the input graph. We will discuss the considerations behind the initial values of these vectors in a later section, but for now we assume that $x_{u_{n_j}}^{(0)} = \mathbf{0}$ for every vertex $u_{n_j} \in U$ and $x_{y_{j,a}}^{(0)} \sim \mathcal{N}(0, 1)^d$ for every vertex $y_{j,a} \in Y$, where d is the number of abstract states that can be used in our abstraction function.

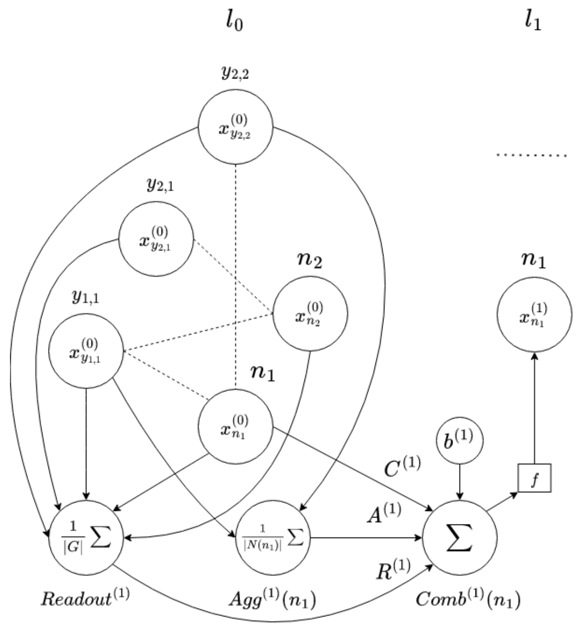

Figure 2: Example computational graph to compute $x_n^{(1)}$ for the bipartite context graph in Fig. 1. Dotted lines represent the connections in the bipartite context graph. Similar substructures are present for the computation of every other node in layer l_1. The node $y_{1,2}$ is removed as it is disconnected from the rest of the graph.

Given a spatial partitioning function H that maps $\mathbb{R}^d \rightarrow [1, ..., d]$ (e.g. argmax), a graph neural network abstraction function is defined in Algorithm 1. The graph neural network $(Comb^{(i)}, Agg^{(i)})$ can be learned through reinforcement learning using evolution strategies on a variance reduction objective as discussed in Section as in Alg. 2. For this paper, we choose the following Aggregation and Combination functions:

$$\frac{1}{N(v)} \sum_{k \in N(v)} x_j$$

$$f(C^{(i)} x_v + A^{(i)}[Agg(v)] + R^{(i)}[Readout] + b^{(i)}) \quad (14)$$

where *Readout* is the global aggregation operations discussed in (Barceló et al. 2020).

RandCB as a graph neural network

It is possible to formulate the RandCB abstraction function as a special case of a graph neural network. Consider the example in Fig. 1. In RandCB, we generate a random integer $C_j \in k$ for each variable $v_j \in C(v_i)$. In the corresponding graph aggregation operation, let each variable $x^{(0)}_{y_{j,a}} \sim a \cdot [1, ...k]$ such that $x^{(0)}_{y_{j,a}} = a \cdot C_j$. The bipartite context graph is shown in Fig. 3.

Suppose we define the following graph neural network with 1 layer and spatial partitioning function:

Algorithm 1: Graph Neural Network Abstraction Function
Input: A graphical model M, a partial probe $S_\mathcal{T}$, a set of child nodes to be partitioned $ch(S^i_\mathcal{T})$
Parameters: A graph neural network $(Comb^{(i)}, Agg^{(i)}), i \in [1, ..., l]$ parameterized by θ, a spatial partitioning function $H : \mathbb{R}^d \rightarrow \{1, ..., d\}$
Output: Labels $s_i \in \{1, ..., d\}, \forall n_i \in ch(S^i_\mathcal{T})$

```
Generate a bipartite context graph $G = (U, V, E)$
  from $(M, S_\mathcal{T}, ch(S^i_\mathcal{T}))$ ;
Initialize $x^{(0)}_{u_{n_j}} = \mathbf{0}, \ \forall u_{n_j} \in U$ ;
Initialize $x^{(0)}_{y_{j,a}} \sim \mathcal{N}(0,1)^d, \ \forall y_{j,a} \in Y$ ;
for $i = 1 \rightarrow l$ do
    for $v \in U \cup V$ do
        $x^{(i)}_v = Comb^{(i)}(x^{(i-1)}_v, Agg^{(i)}(v))$
    end
end
return $\{H(x^{(l)}_{u_{n_j}}), \ \forall u_{n_j} \in U\}$
```

Algorithm 2: Dynamic Abstraction Sampling
Input: Graphical model M and corresponding search tree $\mathcal{T}_o$
Parameters: A spatial partitioning function $H : \mathbb{R}^d \rightarrow \{1, ..., d\}$, g number of generations or T time-limit, L number of individuals per generation, m number of rollouts per individual
Output: Estimate of the partition function Z_{est}

```
Randomly initialize population of parameters $\theta_i$,
  $i \in [1, ..., L]$ ;
$n = 0, s = 0$ ;
for number of generations g or time-limit $T$ do
    for $i = 1 \rightarrow L$ do
        for $j = 1 \rightarrow m$ do
            Call Abstraction Sampling with (Alg. 1
              using parameter $\theta_i$ and function $H$) as
              the abstraction function to sample a
              probe $S_\mathcal{T}$ on $\mathcal{T}_o$ ;
            $\hat{Z}_{i,j} = Z'(S'_\mathcal{T})$ (Eq. 5) ;
            $s = s + Z(S'_\mathcal{T}), n = n + 1$ ;
        end
    end
    $Z_{est} = s/n$ ;
    for $i = 1 \rightarrow L$ do
        $f_i = -\frac{1}{m}\sum_j (max(Z_{est} - \hat{Z}_{i,j}, 0))^2$ ;
    end
    Evolve population $\{\theta_1, ..., \theta_L\}$ using fitness
      values $f_i$ according to bullet 4 in Section  ;
end
return $Z_{est}$
```

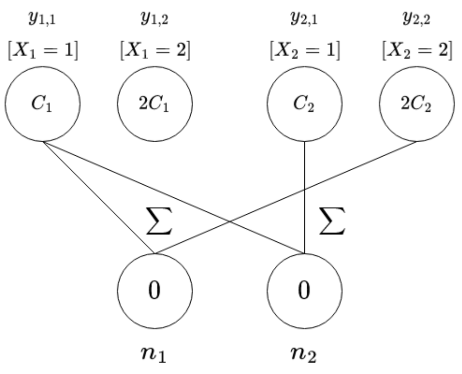

Figure 3: Bipartite context graph for RandCB

- $Comb^{(1)} = (C = \mathbf{I}, A = \mathbf{I}, b = \mathbf{0})$, where $\mathbf{I}$ is the identity matrix
- $Agg^{(1)} = \sum$
- $H = mod\ d$

If we run Algorithm 1 and evaluate the above graph neural network on the biparatite context graph in Fig. 3, we get the exactly the same computations present in RandCB. Essentially, the graph neural network specified above is equivalent to RandCB.

In general, a single layer graph neural network parameterizes a multidimensional hash function. Let $\mathbf{c}_{n_j}$ be the assignments to $C(v_i)$ for a node n_j. We have for any n_j, n_k:

$$\mathbf{c}_{n_j} = \mathbf{c}_{n_k} \rightarrow Agg(n_j) = Agg(n_k) \rightarrow \mathbf{x}^{(i)}_{n_j} = \mathbf{x}^{(i)}_{n_k} \quad (15)$$

It has been shown in past work that RandCB is an effective abstraction function for many problems. Since RandCB can be written as a special case of a graph neural network, we hypothesize it can be possible to learn even better abstraction functions for a given problem using more expressive graph neural networks.

Spatial Partitioning

One important aspect of designing a graph neural network as an abstraction function is the spatial partitioning function H. Before we discuss this, we first provide a geometric understanding of what the computations performed in a graph neural network abstraction function actually do.

The graph neural network abstraction function can be seen as a multivariable hash function that maps vertices in U to a real vector in $\mathbb{R}^d$. To better visualize this process, assume that we are working with just two abstract states, $d = 2$. For simplicity, assume that we use a single layer GNN similar to the RandCB example in the previous example. In the first step of a GNN computation, the aggregation step computes $Agg(n_j)$ for each $u_{n_j} \in U$, we can visualize this as the computation of a 2-dimensional hash similar to how a one dimensional hash is computed in RandCB.

The initial positions of the vectors $Agg(n_i)$ are mapped using the $Comb$ function which essentially acts as a kernel

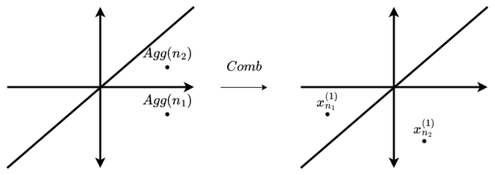

Figure 4: Two dimensional hash space

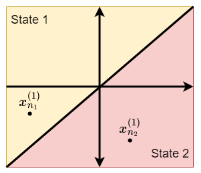

Figure 5: $H = argmax$ spatial partitioning function in two dimensional space

function as in Fig. 4. We then use a spatial partitioning function H to map the real space into d abstract states. An example is shown in Fig. 5, where H is chosen to be the argmax function.

The computations of a graph neural network abstraction function can be thought of as three functions: a hash function, a kernel function, and a spatial partitioning function. As shown in previous work on GNNs, additional layers in the graph neural network increase the expressivity of the model and essentially allow for the approximation of a wider class of kernel functions.

Graph Neural Network Initialization

Another important aspect of designing a graph neural network abstraction function is determining the process for the generation of the initial vectors $x^0_{y_{j,a}}$. It can be useful to generate $x^0_{y_{j,a}}$ such that the initial vectors for variable nodes specifying the same variable with different values are related.

For example, consider an input graph where the domains of each variable in $C(v_i)$ is limited to two values. Our initial construction of the GNN abstraction function assumes that $x^0_{y_{j,1}}$ and $x^0_{y_{j,2}}$ are drawn independently from $\mathcal{N}(0, 1)^d$. However, this means that we lose some information in the form that the two nodes in Y represent the same variable v_j. One possible way to limit the effect of this loss of information is to define a function J such that:

$$x^0_{y_{j,2}} = J(x^0_{y_{j,1}}) \quad (16)$$

and we only need to sample $x^0_{y_{j,1}}$. For this paper, we use $J(x) = -x$ as the initialization mapping as the problems we deal with have boolean-valued variables.

Empirical Results

For evaluation we run Dynamic Abstraction Sampling using a Graph Neural Network with the evolution strategies reinforcement learning approach described in the section (with

Table 1: Graph Neural Network parameters for empirical tests

Parameter	Value
Comb	$f(Cx_v + A[Agg(v)] + R[Readout] + b)$
Agg	$\frac{1}{\lvert N(v)\rvert}\sum_{j\in N(v)} x_j$
H	$\sim Softmax(x)$
Initialization	$x^0_{y_{j,1}} \sim Uniform[-1,1], x^0_{y_{j,2}} = -x^0_{y_{j,1}}$

Table 2: Empirical results on a selection of problems for 1 hr of computation. Estimates closer to ground truth values are bolded.

Problem Instance	Truth	RandCB	GNN
grid20x20.f2	291.732	291.310	**291.537**
grid20x20.f5	665.116	661.149	**661.859**
grid20x20.f10	1311.983	1302.819	**1307.786**
grid20x20.f15	1962.977	1944.677	**1949.414**
grid20x20.f15.wrap	1979.962	1966.380	**1968.297**
rbm20	58.530	**57.354**	57.058
rbm21	63.112	57.592	**62.829**
rbm22	66.553	61.219	**65.454**
or_chain_10.fg	-8.330	-8.810	**-8.391**

population size = 40, 2 evaluations per generation). As a fitness function, we use

$$fitness = -\frac{1}{m}\sum_i (max(Z_{est} - \hat{Z}_i, 0))^2 \qquad (17)$$

where Z_{est} is the current running estimate over all population members of the Z-value. The graph neural network is specified with the parameters listed in Table 1, with trainable parameters (C, A, R, b). We compare the running estimate Z_{est} of the GNN after one hour of computation with one hour of abstraction samplign using RandCB in Table 2 on a preliminary selection of problem instances taken from existing benchmarks: Grids, DBN, and Promedas (Kask et al. 2020).

Dynamic Abstraction Sampling is the most effective when the problem is somewhat difficult but not too large as in the problems tested in Table 2. Compared to Abstraction Sampling with RandCB, each full probe generation using Dynamic Abstraction Sampling takes roughly 3-4 x more time. As a result, if the problem is small and Abstraction Sampling already performs very well, the benefits provided from learning better abstraction functions may not be able to overcome the difference in computational efficiency.

On the other hand, because we need multiple rollouts to update our abstraction functions, it is difficult to apply our method directly to large problem instances. In the Grids benchmark, it can take several minutes for just one rollout on an 80 x 80 sized problem instance. This limits the direct applicability of our method. However, given certain constraints, we can attempt to use transfer learning to bridge this gap in applicability. For example, it is possible to learn abstraction functions on a small problem instance such as grid20x20.f5 and then evaluate the function on similar but large instances such as grid40x40.f5 or grid 80x80.f5. We demonstrate the capability of our method to generalize from smaller instances in Table 3.

Table 3: Transfer learning results for networks trained for 1 hr on given small instance and then evaluated for 1 hr on larger instance in Grid benchmark. Values are compared with results from 1 hr of Abstraction Sampling using RandCB.

Base	Truth	RandCB	GNN
Trained on grid20x20.f15			
grid40x40.f5	2792.203	2743.674	**2747.990**
grid40x40.f10	5491.076	**5402.128**	5392.410
grid40x40.f15	8198.61	8022.986	**8037**
grid80x80.f5	11163	10901	**10902**
grid80x80.f10	21785.5	21225.005	**21226.268**
grid80x80.f15	32550	31655	**31750**
Trained on grid20x20.f5			
grid40x40.f5	2792.203	2743.674	**2746.933**
grid40x40.f15	2792.203	8022.986	**8031.676**
Trained on grid20x20.f10			
grid40x40.f5	2792.203	2743.674	**2746.589**
grid40x40.f10	5491.076	5402.128	**5405.262**
grid40x40.f15	8198.61	8022.986	**8033.929**
grid80x80.f10	21785.5	21225.005	**21256.046**

Conclusion

We have proposed Dynamic Abstraction Sampling, an online method for learning better abstraction functions. Our method uses evolutionary methods as a backbone to optimize towards a running estimate. We propose a bipartite graph structure for capturing the information present in the context of frontier nodes and a graph neural network architecture to exploit the information encoded in this structure. We provide preliminary empirical evaluation over several problem instances from different benchmarks and also demonstrate the potential generalizability that our method can have by training on smaller instances and evaluating on larger ones in a given benchmark.

While it seems that evolutionary strategies are an effective method for learning abstraction functions, it would be interesting to explore other reinforcement learning approaches such as policy gradient methods. However, any potential gradient based method will be slower than the current evolutionary approach and it will be necessary to evaluate whether the benefit of using gradient based algorithms can outweigh the increasing computational cost.

Furthermore there are still some details that could be expanded upon in the graph neural network architecture. In particular, it is not clear what initialization functions should be used if the variables in a problem instance take more than two values. In this case, it may be interesting to learn the initialization function along with the abstraction function.

Acknowledgements This work supported in part by NSF grant IIS-2008516 and AFOSR grant 1010GWA357. Special thanks to Alexander Ihler and Bobak Pezeshki for their help in discussions about the work.

References

Baluja, S.; and Caruana, R. 1995. Removing the genetics from the standard genetic algorithm. In *Machine Learning Proceedings 1995*, 38–46. Elsevier.

Barceló, P.; Kostylev, E.; Monet, M.; Pérez, J.; Reutter, J.; and Silva, J.-P. 2020. The logical expressiveness of graph neural networks. In *8th International Conference on Learning Representations (ICLR 2020)*.

Beyer, H.-G.; and Schwefel, H.-P. 2002. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1): 3–52.

Broka, F.; Dechter, R.; Kask, K.; and Ihler, A. 2018. Abstraction sampling in graphical models. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Chen, P. C. 1992. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, 21(2): 295–315.

Darwiche, A. 2009. *Modeling and reasoning with Bayesian networks*. Cambridge university press.

Dechter, R. 2013. Reasoning with probabilistic and deterministic graphical models: Exact algorithms. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 7(3): 1–191.

Dechter, R.; and Mateescu, R. 2007. AND/OR search spaces for graphical models. *Artificial intelligence*, 171(2-3): 73–106.

Gogate, V.; and Dechter, R. 2011. SampleSearch: Importance sampling in presence of determinism. *Artificial Intelligence*, 175(2): 694–729.

Hamilton, W.; Ying, Z.; and Leskovec, J. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30.

Kask, K.; Pezeshki, B.; Broka, F.; Ihler, A.; and Dechter, R. 2020. Scaling up AND/OR abstraction sampling. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence,{IJCAI} 2020*.

Knuth, D. E. 1975. Estimating the efficiency of backtrack programs. *Mathematics of computation*, 29(129): 122–136.

Salimans, T.; Ho, J.; Chen, X.; Sidor, S.; and Sutskever, I. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, 629–637.