Predictive Control of BDD Growth: Reinforcement Learning for Dynamic Variable Reordering

Luke Slater[1]

[1]University of Cape Town and CAIR, South Africa

Abstract

Knowledge compilation is a paradigm in knowledge representation and reasoning that transforms complex combinatorial objects into representations supporting tractable query evaluation. Among its most fundamental tools are Binary Decision Diagrams (BDDs), whose efficiency depends critically on variable ordering. In practice, this gives rise to the dynamic variable reordering problem, which is traditionally addressed via local search heuristics such as sifting, guided primarily by diagram size. This proposal develops a structure-aware and planning-based approach to dynamic reordering. We formulate the problem as a sequential decision process and introduce a compositional neural encoder that operates directly on BDD structure, enabling the use of deep reinforcement learning to guide reordering decisions. Furthermore, we exploit the fully observable and deterministic nature of BDD packages to incorporate model-based planning via lookahead search. The resulting framework aims to move beyond size-driven heuristics by enabling decisions informed by both the structure of the diagram and the anticipated long-term effects of reordering actions.

Keywords

Knowledge Compilation, Binary Decision Diagrams, Variable Ordering, Dynamic Variable Reordering, Reinforcement Learning, Model-Based Planning

1. Introduction

Binary Decision Diagrams (BDDs) are a central representation in knowledge compilation, enabling efficient reasoning tasks such as satisfiability checking, equivalence checking, and model counting. Their practical usefulness, however, depends heavily on variable ordering: a poor ordering can make a BDD exponentially larger, while a good ordering can keep it compact. Dynamic variable reordering addresses this problem during BDD-based computation, typically through local search heuristics such as sifting. These methods are effective, but they are guided mainly by diagram size and do not explicitly learn from the internal structure of the BDD or from the long-term effects of reordering decisions. This proposal studies dynamic variable reordering as a sequential decision-making problem. Adjacent variable swaps are treated as actions, the current BDD manager state as the environment state, and changes in diagram size as the reward signal. Building on this formulation, the proposed approach combines a compositional neural encoder for BDD structure with reinforcement learning and model-based lookahead search. The goal is to learn reordering policies that anticipate diagram growth rather than merely reacting to it after it occurs.

2. High Level Backround

2.1. Binary Decision Diagrams

A Binary Decision Diagram (BDD) G is a directed acyclic graph representation of a Boolean function constructed with respect to a fixed variable ordering [1]. Each node v in the diagram can be either a terminal node labelled 0 or 1, or an internal node labelled by a particular variable index $\mathrm{index}(v)$. Each internal node has two outgoing edges: a low edge $\mathrm{low}(v)$ (corresponding to assigning the variable to 0) and a high edge $\mathrm{high}(v)$ (assigning it to 1). The diagram is ordered, meaning variables appear in a fixed

Doctoral Consortium of the 23rd International Conference on Principles of Knowledge Representation and Reasoning (KR 2026 DC), July 20-23, 2026, Lisbon, Portugal
sltluk001@myuct.ac.za (L. Slater)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-06-30

CEUR-WS.org/Vol-4218/paper5.pdf

order along all paths, and reduced via two rules: merging isomorphic subgraphs and eliminating nodes whose low and high edges coincide. These constraints ensure that the representation is canonical for a given ordering. This structure yields strong algorithmic properties with respect to the diagram size $|G|$. Satisfiability, tautology, and equivalence checking are $O(1)$, equivalence reduces to pointer equality, model counting can be performed in $O(|G|)$ time, and the `apply` operation—used to combine two BDDs under a Boolean operator—runs in $O(|G_1| \cdot |G_2|)$ time [1]. As a result, many semantic queries and transformations can be performed efficiently once a compact representation is obtained.

2.1.1. BDD Packages

BDD-based computation is typically carried out within a BDD package, such as CUDD [2], as part of a larger program (e.g., written in C++). The program interacts with a persistent BDD manager, which maintains a shared, multi-rooted DAG representing a collection of Boolean functions as BDDs. These functions are treated as first-class objects: the program holds references to them, and operations are expressed as Boolean combinations of existing functions. Execution proceeds as a sequence of operations. At any time t, the manager maintains a collection B_t of BDDs. An operation ω_t takes one or more functions from B_t, applies a Boolean operator via procedures such as `apply`, and inserts the resulting diagram back into the shared structure, yielding an updated collection B_{t+1}. The manager also performs garbage collection when functions are no longer referenced by the program. In this way, a BDD-based computation can be viewed as an evolving system in which a sequence of operations $\Omega = (\omega_0, \omega_1, \dots)$ incrementally transforms a shared diagram state over time, while the program manipulates references to the underlying functions.

2.1.2. The Variable Ordering Problem

A central limitation of BDDs is that, once a variable ordering is fixed, the size of the resulting reduced diagram is completely determined by that ordering. Since all major operations scale with diagram size, the efficiency of the entire BDD paradigm reduces to an optimization problem over permutations of the variables. This problem has been studied in two distinct forms. In the *static* setting, one is given a Boolean formula (typically CNF) and seeks to predict a variable ordering prior to construction, after which the BDD is built from scratch using `apply` under that fixed ordering. Primary examples of this approach include the FORCE [3] and MINCE [4] algorithms.

In the *dynamic* setting, originating from Rudell's work [5], the key enabling primitive is the *adjacent swap* algorithm: given an already constructed BDD (or collection of BDDs), one can swap two neighboring variables in the ordering by traversing the diagram and applying local graph transformations that produce the BDD for the new ordering, without reconstructing it from the original formula. This primitive induces a local search space over permutations, where each move corresponds to a cheap transition between adjacent orderings. All dynamic reordering methods are built around this primitive, with the dominant approach being *sifting* [5] and its variants. The basic sifting algorithm selects a variable and repeatedly swaps it with its neighbors, moving it through all possible positions in the ordering while evaluating the resulting diagram size at each step, and finally placing it in the position that minimizes size; this procedure is then repeated for multiple variables, yielding a greedy local search over orderings. Variants such as *symmetrical sifting* [6], which exploits symmetries between variables, and *group sifting* [7], which moves blocks of variables together, refine this process but retain the same underlying mechanism of exploring adjacent orderings using Rudell's swap primitive guided by size.

This distinction is especially important in practical BDD packages. Static methods are typically used, when possible, to select an initial ordering before program execution. During execution, however, the diagram state may grow unpredictably, and dynamic reordering is used to adapt the ordering in place. In systems such as CUDD, this may be invoked manually through manager routines or triggered automatically once memory usage exceeds a threshold. As a result, dynamic reordering has become the dominant approach in practice: rather than restarting the computation under a new ordering, the system attempts to repair the existing diagram state and continue execution. This proposal focuses on

the dynamic setting.

2.2. Reinforcement Learning

Reinforcement learning (RL) is a paradigm for sequential decision making in which an agent interacts with an environment over time [8]. The environment is modeled as a set of states $\mathcal{S}$, where at each step t the agent observes a state $s_t \in \mathcal{S}$ and selects an action $a_t \in \mathcal{A}(s_t)$ from the set of available actions. This induces a transition to a new state s_{t+1} and produces a scalar reward $r_t \in \mathbb{R}$. The agent's behaviour is defined by a policy $\pi(a \mid s)$, which specifies a distribution over actions given a state. The objective is to learn a policy that maximizes the *return*, defined as the cumulative reward over a trajectory, typically $R = \sum_{t=0}^{T-1} r_t$. Learning proceeds through repeated interaction with the environment: by exploring different action sequences and observing their outcomes, the agent gradually improves its policy to favor actions that lead to higher long-term return. In many problems, the state space is large and highly structured, making it impractical to represent states explicitly. *Deep* reinforcement learning addresses this by using neural networks to encode states and approximate policies or value functions, enabling the agent to process complex structured inputs and generalize to previously unseen states.

3. Problem Formulation

The proposed research is based on the observation that dynamic variable reordering in BDDs can be naturally formulated as a sequential decision-making problem. The current BDD manager state—i.e., the live diagrams under a given variable ordering—serves as the environment state, while adjacent variable swaps define the available actions that induce local transitions in this structure. The effect of each action is directly observable through changes in diagram size, providing a natural step-wise reward signal, while the cumulative reduction (or growth) in size over a sequence of swaps defines the overall return. This correspondence aligns closely with the reinforcement learning paradigm, suggesting that effective reordering strategies can be learned as policies over diagram states.

3.1. Dynamic Variable Reordering as a Markov Decision Process

Our first proposal is to develop a controller for dynamic variable reordering, i.e., a decision-making mechanism that actively selects reordering operations based on the current state of a fixed collection of BDDs (corresponding to a snapshot of the program state) within a manager, with the objective of minimizing their overall representation size. We model this as a finite-horizon Markov decision process initialized by a BDD manager containing a collection of diagrams B_0 under an initial variable ordering σ_0. At each step t, the state $s_t \in \mathcal{S}$ is given by

$$s_t = (B_t, k_t),$$

where B_t denotes the current collection of diagrams (implicitly defining the ordering σ_t), and k_t is the remaining swap budget. At each state s_t, the controller conditions its next decision on the current diagram state itself—rather than solely on scalar feedback such as diagram size as with sifting—and selects an action from the available action set $\mathcal{A}(s_t)$. The action space consists of adjacent variable swaps together with a termination action:

$$\mathcal{A}(s_t) = \{0, 1, \ldots, n-2\} \cup \{\text{STOP}\}.$$

Selecting an index $i \in \{0, 1, \ldots, n-2\}$ applies a swap between the variables at positions i and $i+1$ using Rudel's adjacent swap algorithm, yielding a new state $s_{t+1} = (B_{t+1}, k_t - 1)$. The STOP action terminates the episode. Because BDDs are canonical to variable ordering, transition dynamics are deterministic. Let $|B_t|$ denote the total number of nodes in the current set of diagrams. A natural step-wise reward is given by the change in representation size,

$$r_t = |B_t| - |B_{t+1}|,$$

so that positive reward corresponds to a reduction in diagram size. Over a sequence of T actions, the return is

$$R = \sum_{t=0}^{T-1} r_t = |B_0| - |B_T|,$$

which directly captures the total reduction (or increase) in representation size achieved by the reordering sequence. In summary, the BDD manager state defines the environment state, adjacent swaps define the action space, and changes in diagram size define the reward signal.

3.2. Predictive Control

While the formulation above treats reordering as a bounded optimization problem over a fixed collection of diagrams, practical BDD usage occurs within an evolving computational process in which diagrams are continuously constructed and transformed. In this setting, the objective is no longer simply to reduce the size of a static collection, but to control the growth of the diagram state over time. This motivates a predictive control perspective, in which reordering decisions are interleaved with program execution and are made in anticipation of future operations, rather than applied only after undesirable growth has already occurred.

We model this as a sequential decision process over the execution of a BDD-based program. Let $\Omega_0 = (\omega_0, \omega_1, \dots, \omega_{T-1})$ denote a sequence of operations to be applied by the BDD manager. At each step t, the state is given by

$$s_t = (B_t, k_t, \Omega_t),$$

where B_t is the current collection of diagrams under ordering σ_t, k_t is a local swap budget available before the next operation, and $\Omega_t = (\omega_t, \omega_{t+1}, \dots)$ is the remaining sequence of operations (assumed known or partially observable depending on the setting). At each state, the controller observes the current diagram state together with the upcoming computational context and selects an action from

$$\mathcal{A}(s_t) = \{0, 1, \dots, n-2\} \cup \{\text{STEP}\}.$$

Selecting an index i applies an adjacent swap between variables at positions i and $i+1$, updating the diagrams and reducing the budget k_t. Selecting STEP executes the next operation ω_t, advancing the program and resetting the local swap budget.

Let Δ_t denote the number of BDD nodes created by the manager between successive STEP actions (including any intermediate swaps). A natural reward is

$$r_t = -\Delta_t,$$

so that the return

$$R = \sum_{t=0}^{T-1} r_t$$

captures the total allocation cost over the execution. This objective penalizes transient blow-ups as well as final diagram size, encouraging the controller to proactively regulate the growth of the diagram state throughout the computation.

4. Proposed Approach and Early Results

Building on the formulations above, the proposed approach aims to move beyond existing dynamic reordering methods, which perform local search guided primarily by diagram size, by introducing a structure-aware and planning-based framework. At a high level, the approach combines three components: (i) a compositional neural encoder that extracts representations of the BDD state itself, (ii) a reinforcement learning controller that selects reordering actions based on these representations, and (iii) model-based search that leverages the exact transition dynamics provided by the BDD manager. Together, these components enable decision making that is informed by both the structural properties of the diagram and the anticipated long-term effects of reordering actions.

4.1. Compositional BDD Encoder

Existing dynamic reordering methods are effectively blind to the internal structure of the diagram, relying almost entirely on scalar signals such as node count to guide decisions. In contrast, our approach conditions directly on the BDD itself, using a neural encoder to extract representations of its functional structure. BDDs admit a highly constrained and canonical form that generic graph encoders fail to exploit. In particular, they exhibit invariance to logical equivalence, equivariance under variable relabeling, and symmetry under complementation, while each node corresponds uniquely to a Boolean subfunction. This structure allows the diagram to be viewed as a canonical circuit, suggesting an encoder that mirrors its recursive semantics rather than treating it as an arbitrary graph.

We therefore propose a compositional encoder that operates bottom-up over the diagram. Each node v is assigned a vector representation $\mathrm{Enc}(v) \in \mathbb{R}^d$. The terminal nodes 0 and 1 are associated with learned embedding vectors $e_0, e_1 \in \mathbb{R}^d$, while each non-terminal node is computed by applying a shared node block—a parametric neural transformation with shared weights across all nodes—to its variable index and the representations of its children:

$$\mathrm{Enc}(v) = \begin{cases} e_0 & v = 0, \\ e_1 & v = 1, \\ \mathrm{NodeBlock}([\mathrm{level}(v) \,\|\, \mathrm{Enc}(\mathrm{low}(v)) \,\|\, \mathrm{Enc}(\mathrm{high}(v))]) & \text{otherwise.} \end{cases}$$

Operationally, this defines a neural computation that mirrors the structure of the BDD: a forward pass evaluates the encoder bottom-up from the terminal nodes to the roots, producing representations for all subfunctions, while a backward pass propagates gradients through the same structure during training. In this sense, the BDD itself induces a neural circuit whose parameters are shared across nodes. This construction can be viewed as a learnable analogue of the standard bottom-up hashing procedure used in BDD packages, replacing fixed structural fingerprints with trainable representations. By aligning the architecture with the canonical structure of BDDs, the encoder respects the underlying invariances and captures the semantics of subfunctions, enabling the controller to reason directly over the diagram rather than relying only on indirect signals such as size.

To support decision making, the representation at the root of the diagram (or an aggregation over multiple roots in the case of a collection B_t) is used as a global embedding of the current BDD state. This embedding is then passed to task-specific output heads, such as a policy head that produces a distribution over actions or a value head that estimates the expected return of the current state. The entire architecture—comprising the encoder and the output heads—is trained end-to-end using standard reinforcement learning objectives, with gradients propagated through the induced computation graph of the BDD.

4.2. Model Based Reinforcement Learning with Search

A key observation is that BDD reordering, as implemented in packages like CUDD, provides a fully observable, deterministic, and efficiently simulatable environment. Given a current diagram state, the effect of local reordering operations—such as adjacent swaps—can be evaluated exactly through the manager's transition procedures, effectively yielding a high-fidelity model of the underlying dynamics. This makes the setting particularly well suited to model-based reinforcement learning, where the agent has access to (or learns) a model of the environment and can use it to simulate future trajectories. In this context, augmenting learning with search refers to performing lookahead planning—e.g., exploring sequences of swaps using the simulator—guided by learned policy and value functions. The combination is especially natural here: the availability of exact transition dynamics enables efficient planning, while learning provides a mechanism for generalizing across diagram states and directing search toward promising regions of the reordering space. The key point here is that existing methods perform local search guided solely by diagram size, whereas our approach augments this process with learned representations of diagram structure and planning.

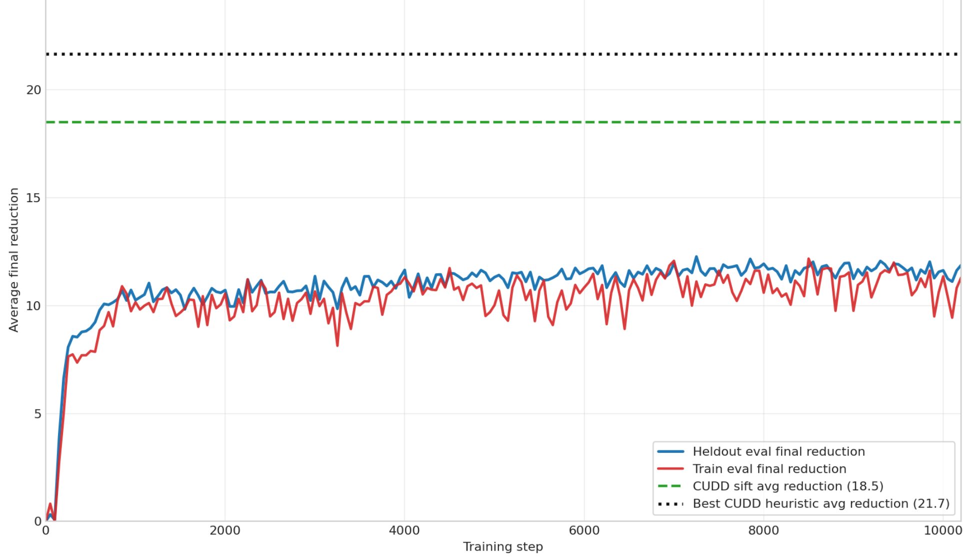

Figure 1: Average final BDD size reduction during training.

4.3. Early Results

The proposed approach is still at an early stage of development, and as such, extensive experimental evaluation has not yet been conducted. Nevertheless, we have performed preliminary experiments on small-scale instances using deep Q-learning to assess the viability of the reinforcement learning formulation for dynamic variable reordering. In this setting, deep Q-learning, as introduced by Mnih et al. [9], is used to approximate an action-value function $Q(s, a)$ over BDD manager states $s_t = (B_t, k_t)$ and reordering actions. A neural network takes as input a representation of the current diagram state produced by the compositional BDD encoder described above, and outputs estimates of Q-values for each possible action. The value $Q(s, a)$ represents the expected discounted return obtained by applying a swap (or termination) action in a given state and continuing the reordering process thereafter:

$$Q(s, a) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t \,\middle|\, s_0 = s, a_0 = a\right],$$

where rewards are defined in terms of changes in diagram size, $r_t = |B_t| - |B_{t+1}|$, and $\gamma \in [0, 1)$ is a discount factor. The network is trained from observed transitions to minimize the discrepancy between predicted and target Q-values, enabling the agent to learn action preferences that reflect long-term reductions in diagram size.

In this experiment, the compositional BDD encoder uses an embedding dimension of 64, with both the encoder node block and the Q-value head implemented using two hidden layers of 128 units. The model is trained using Q-learning targets generated from adjacent-swap search. On held-out functions, the learned greedy policy reaches an average final reduction of roughly 12 nodes after about 10k training steps, using about 41 adjacent swaps on average. By comparison, CUDD sifting achieves an average reduction of 18.5 nodes using about 127 swaps, while the best result across all CUDD reordering heuristics achieves an average reduction of 21.7 nodes using about 4610 swaps. The corresponding training-set evaluation closely tracks the held-out curve, indicating that the learned policy is not simply memorizing the training functions. This is notable because deployment uses purely greedy action selection from the learned Q-values: each adjacent swap is chosen directly from the network's predicted action values, without lookahead, rollout search, or access to CUDD's internal reordering procedure.

Acknowledgments

This work is based on the research supported in part by the National Research Foundation of South Africa (REFERENCE NO: SAI240823262612, PMDS250603318790).

Declaration on Generative AI

During the preparation of this work, the author used ChatGPT to assist with grammar, spelling, phrasing, LaTeX formatting, and the revision of explanatory text. The author reviewed and edited the generated suggestions and takes full responsibility for the content of the publication.

References

[1] Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Transactions on Computers C-35 (1986) 677–691.

[2] F. Somenzi, Cudd: Cu decision diagram package, release 3.0.0, 2015.

[3] F. A. Aloul, I. L. Markov, K. A. Sakallah, Force: a fast and easy-to-implement variable-ordering heuristic, GLSVLSI '03, Association for Computing Machinery, New York, NY, USA, 2003, p. 116–119. URL: https://doi.org/10.1145/764808.764839. doi:10.1145/764808.764839.

[4] F. A. Aloul, I. L. Markov, K. A. Sakallah, MINCE: A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation, Journal of Universal Computer Science 10 (2004) 1562–1596.

[5] R. Rudell, Dynamic variable ordering for ordered binary decision diagrams, in: Proceedings of 1993 International Conference on Computer Aided Design (ICCAD), 1993, pp. 42–47.

[6] S. Panda, F. Somenzi, B. F. Plessier, Symmetry detection and dynamic variable ordering of decision diagrams, in: Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design, ICCAD '94, IEEE Computer Society Press, Washington, DC, USA, 1994, p. 628–631.

[7] S. Panda, F. Somenzi, Who are the variables in your neighbourhood, in: Proceedings of IEEE International Conference on Computer Aided Design (ICCAD), 1995, pp. 74–77. doi:10.1109/ICCAD.1995.479994.

[8] R. S. Sutton, A. G. Barto, Reinforcement Learning: An Introduction, second ed., The MIT Press, 2018.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep reinforcement learning, Nat. 518 (2015) 529–533.