MORK-MOSES
Implementation Ideas

Ben Goertzel

July 21, 2025

Abstract

We outline a potential route to a scalable implementation of the MOSES (Meta-Optimizing Semantic Evolutionary Search) framework within the MORK implementation of the Hyperon Atomspace. We begin by leveraging gCoDD, a Combinatory Decision DAG representation of programs with grounded predicates at leaves, and applying Elegant Normal Form (ENF) with Correlation-Adapted ENF (CENF) for canonicalization. Variation is expressed via quantale operations, including mask-based crossover, and Estimation-of-Distribution (EDA) is performed by embedding quantale-valued factor graphs directly in Atomspace. We describe efficient local rewrites for crossover, mutation, ENF/CENF normalization, pattern mining for n-ary factor discovery, and belief-propagation sampling. Complexity analysis demonstrates that each MOSES step runs in time proportional to the size of affected subgraphs or variables, enabling parallel, distributed execution across Atomspace shards.

Contents

1 Introduction

Figure 1: Leading the way to efficient scalable automated program learning for AGI!

The MOSES framework combines evolutionary search with probabilistic model learning to automate program induction. In its original form, MOSES operates on program sketches and numeric or Boolean knobs for controlling program generation – iteratively sampling, evaluating, modeling, and varying programs to optimize fitness functions on program space. However, efficient scaling to large populations and complex program spaces in the MOSES approach involves many complexities, including representations and operations that support deduplication, local rewrites, and distributed inference. Achieving all this in a scalable fashion has proved challenging historically, but we believe this challenge can be met via implementing MOSES in a manner tightly integrated with the internal data structures and operations of the MORK realization of the Hyperon Atomspace.

The main technical ingredients of our proposed MORK-MOSES design are:

1. **gCoDD representation for programs.** We extend CoDD with atomic leaves for grounded predicates, preserving Elegant Normal Form (ENF) and enabling constant-time equality and maximal subgraph sharing.
2. **ENF and CENF normalization.** We describe local rewrite rules for ENF and its extension CENF, which clusters correlated guards into meta-guards to reduce dimensionality in EDA models.
3. **Quantale-based variation.** We formulate crossover and mutation as operations in a commutative quantale, including mask-based crossover for creative building-block preservation.
4. **Factor-graph EDA.** We embed probabilistic factor graphs in Atomspace using quantale-valued factors, discovered via Hyperon pattern mining of n-ary guard correlations, and perform efficient belief-propagation sampling.

By mapping MOSES primitives to Atomspace atoms and links, and leveraging these various techniques – implemented leveraging the low-level structures and processes provided by the MORK implementation of Hyperon Atomspace and combined together appropriately – this implementation design aims to support large-scale, parallel evolutionary search using the MOSES approach

Alongside qualitative analysis, we roughly analyze the cost of each MOSES phase in this implementation approach –sampling, construction, normalization, evaluation, selection, model update, inference, and variation–showing that all major steps run in time proportional to local subgraph or variable counts, and scale linearly across demes and Atomspace shards.

2 The MOSES Algorithm Framework

The MOSES (Meta-Optimizing Semantic Evolutionary Search) framework is a population-based, estimation-of-distribution style algorithm for automated program learning. It interleaves sampling, evaluation, model learning, and variation in a loop that promotes high-fitness, diverse programs.

2.1 Key Components

- **Programs and Knobs.** Each individual is a *program sketch* with a fixed structure and a small vector of numeric or Boolean parameters (“knobs”) that control details.
- **Demes.** A *deme* is a subpopulation of programs sharing one sketch. MOSES maintains multiple demes in parallel, each exploring a region of program space.
- **Fitness Oracle.** A user-supplied function evaluates each program on the target task and returns a fitness score.
- **Variation Operators.** Crossover and mutation are applied at the knob level (mixing or perturbing parameter vectors) or at the sketch level (splicing subtrees).

- **Distribution Model.** An estimation of distribution algorithm (EDA) builds a generative model over knob vectors in each deme, capturing correlations that lead to high fitness.
- **Normalization.** Programs are canonicalized (ENF or CENF) after variation to ensure unique representation and efficient comparison.
- **Deme Management.** Demes are spawned, retired, merged or migrated based on exemplar performance and resource constraints.

2.2 Algorithmic Loop

1. *Initialization.* Create one or more demes by instantiating base sketches with random knob settings.
2. *Sampling.* For each deme, sample a batch of knob vectors from its current generative model (initially uniform).
3. *Program Construction.* Convert each knob vector into a full program by filling in the sketch and normalizing to ENF.
4. *Evaluation.* Run the fitness oracle on each program and record scores.
5. *Selection.* Choose the top-performing knob vectors or programs as exemplars.
6. *Model Learning.* Update the deme's EDA model (e.g. Chow-Liu tree or block model) using the selected exemplars, capturing statistical dependencies.
7. *Variation.* Generate new knob vectors via model sampling, mask-based crossover, and mutation.
8. *Normalization.* Apply ENF (and optionally CENF) to each new program to enforce canonicity.
9. *Deme Update.* Replace low-fitness individuals, spawn new demes for novel sketches, retire redundant demes, and migrate exemplars between demes as needed.
10. *Termination.* Repeat until a stopping criterion is met (e.g. fitness threshold or resource limit).

2.3 Desirable Properties

- **Scalability.** Demes run independently and in parallel; variation and normalization are local graph rewrites.
- **Duplication Control.** Canonical forms guarantee that identical programs collapse to one representation, saving evaluation time.

- **Adaptive Exploration.** The EDA model focuses sampling on promising regions while preserving diversity via probabilistic masks.
- **Modularity.** The separation of sketch, knobs, model, and evaluation allows plugging in new primitives or oracles without changing the core loop.
- **Incremental Update.** Model learning and normalization operate incrementally on changed nodes, enabling efficient online updates.

MOSES has had two major implementations in the past – a standalone version loosely integrated with OpenCog Classic, and a more recent version called "Atomspace-MOSES", fully implemented in the OpenCog Atomspace. Implementation of MOSES in the MeTTa language from Hyperon has been envisioned for some time, and has potential to create a very flexible and configurable MOSES version suitable for learning a wide variety of program types, including recursive meta-learning of cognitive control programs for Hyperon systems. However, a straightforward implementation of MOSES in MeTTa has risk of not possessing sufficient scalability for the large-scale applications needed for AGI or sizeable commercial applications. This motivates the current ideas, which aim to design an implementation of MOSES mapping at the low level onto the MORK implementation of Atomspace. Program sketches and knob vectors map naturally to Atomspace representations, variation becomes local Atom rewrites, normalization is ENF/CENF, and EDA model learning leverages MORK's pattern-mining and factor-graph primitives.

3 gCoDD: CoDD with Grounded Predicates for MOSES

What we call here the *gCoDD* representation augments the standard Combinatory Decision DAG (CoDD) formalism (summarized in the Appendix to the "Weakness is All You Need" paper) by treating each call to a fixed library of grounded predicates as an *atomic leaf* in the DAG. This simple extension preserves all of the core advantages of CoDD for the MOSES framework while accommodating real-world primitive operations. In particular:

- **Canonical, Duplicate-Free Programs.** By collapsing every grounded predicate call to a single labeled leaf node, gCoDD retains the Elegant Normal Form (ENF) property: behaviorally equivalent programs map to the same DAG. This yields pointer-level sharing and constant-time program equality checks without inspecting predicate internals.
- **Local and Efficient Variation.** Variation operators (crossover, mutation) act only on the DAG structure above the leaves. Each grounded call counts as exactly one node, so splicing or grafting remains $O(k)$ in the number of affected nodes. No bespoke tree surgery inside predicates is required.

- **Occam-Style Program Cost.** The total node count still serves as a direct measure of program complexity (weakness): each grounded call contributes unit cost (or a cost with a certain weight determined based on the specifics of the grounding function). This enforces a clear length-penalty bias that encourages simpler, more general programs.

- **Clean Knob Vector for EDA.** Decision edges and combinator choices above the leaves yield a fixed-length binary "knob vector" for each program. Grounded predicates simply appear as categorical leaf types, which can be treated as additional knobs in factor-graph or tree-based distribution estimation.

- **Seamless Integration with Grounded Libraries.** Since leaves are opaque atoms, adding or modifying grounded predicates requires no change to the ENF rules or core MOSES pipeline. New primitives are supported immediately as new leaf labels.

- **Incremental Compilation.** Translating between MeTTa-IL and gCoDD is a local, linear-time pass. Grounded leaves survive the round-trip unchanged; only the surrounding combinator DAG is re-normalized.

In practice, gCoDD lets MOSES exploit the performance and deduplication benefits of CoDD while retaining full flexibility to call out to domain-specific predicates. Grounded operations remain atomic, preserving search locality and analytic clarity, and can be elaborated with optional metadata or parameter subgraphs only when deeper EDA insight is required.

3.1 Grounded Functions in gCoDD Programs

The gCoDD representation treats calls to external, "grounded" functions as atomic leaves. These primitives must be provided by the MORK runtime or host language to implement program semantics, pattern mining, factor-graph inference, and quantale operations. Typical categories include:

- **Basic combinator primitives:**
  - Identity, composition, K and S combinators, branching combinators.
- **Arithmetic and numeric operations:**
  - addition, subtraction, multiplication, division, comparisons (`<`, `<=`, `=`, `>=`, `>`).
- **Boolean logic:**
  - `and`, `or`, `not`, implication.
- **Data structure constructors:**
  - list `cons`, `nil`, `append`, `map`, `filter`, `fold`.

- **Quantale operations:**
  - `q-join` ($\oplus$), `q-times` ($\otimes$), `q-residuum`, `q-unit`.
- **Pattern mining primitives:**
  - `countPair(guard1,guard2)`, `updateCountPair`, `findFrequentPatterns`.
- **Clustering primitives:**
  - `singleLinkageMerge`, `thresholdCluster`, `defineMetaGuard`.
- **Factor graph API:**
  - `fg-add-variable`, `fg-add-factor`, `fg-infer`, `fg-sample`.
- **Randomness and sampling:**
  - `randomBernoulli(p)`, `randomUniform`, `randomChoice`.
- **System and I/O primitives (optional):**
  - `sensorRead`, `actuatorWrite`, `fileRead`, `networkRequest`.

These grounded functions supply the leaf-level semantics and system interfaces, allowing the gCoDD layer to orchestrate their composition, variation, and inference through DAG rewrites and quantale algebra.

3.2 Conversion between MeTTa and gCoDD

The following outlines the two transformations required to interoperate between MeTTa-IL programs and the gCoDD representation used by MOSES. It's a bit obvious, but worth being on clear regarding what sort of processes are involved here, if for no other reason than to ward off the impression there might be more complexity involved.

3.3 From MeTTa-IL to gCoDD

1. **Parse MeTTa-IL.** Read the MeTTa-IL source into an abstract syntax tree (AST).
2. **Emit raw DAG.** Walk the AST and for each node:
   - If it is a combinator or operator, create an internal DAG node.
   - If it is a grounded predicate call, create an *atomic leaf* labeled by the predicate identifier.
3. **Hash-consing.** Merge identical subtrees by pointer-sharing to form a DAG with maximal reuse.
4. **ENF normalization.**

- Apply local rewrite rules (promote, distribute, eliminate duplicates, etc.) until a fixpoint.
- Sort the children lists of commutative nodes to enforce a unique ordering.

5. **Resulting gCoDD.** The final, canonical DAG is the gCoDD representation of the original program.

3.4 From gCoDD to human-readable MeTTa

1. **Topological traversal.** Perform a topological sort of the DAG nodes so that dependencies are emitted before use.
2. **Emit MeTTa-IL expressions.** For each node in order:
   - Internal node: emit the corresponding combinator or operator application, inserting child subexpressions in the canonical order.
   - Leaf node: emit the grounded predicate call using its original MeTTa syntax.
3. **Pretty-printing.** Optionally apply:
   - Inlining of trivial subexpressions.
   - Re-introduction of MeTTa surface syntax (macros, sugar) where it preserves semantics.
4. **Final MeTTa program.** The output is a human-comprehensible MeTTa (or MeTTa-IL) program that is semantically equivalent to the gCoDD.

4 Efficient Representation of gCoDD Programs in MORK

A gCoDD program is a rooted directed acyclic graph (DAG) whose internal nodes are combinators or guards and whose leaves are grounded predicate calls. To store and manipulate these efficiently in MORK, we use the following conventions:

1. **Node atoms.** Each DAG node is represented by an atom of the form

   ```
   ProgramNode(id : SYMBOL)
   ```

   where `id` is a unique identifier (hash of its contents) and `SYMBOL` is the combinator name or predicate label.
2. **Child links.** For each ordered child list $(c_1, \ldots, c_n)$ of a node `p`, we add link atoms

```
Edge(p, c_i, i)
```

for $i = 1 \ldots n$. The integer slot preserves ENF's canonical child ordering for commutative nodes.

3. **Hash-consing for maximal sharing.** When inserting a new `ProgramNode` we first compute a content hash of its symbol and child list. If an identical atom already exists, we reuse its pointer. This yields $O(1)$ structural equality tests and shared sub-DAGs at no extra cost.

4. **Path indexing for fast lookup.** MORK's built-in path index records all outgoing edges from a given atom. To match or rewrite a sub-DAG rooted at `p`, we query

   ```
   find ProgramNode(p : _), Edge(p, c, _)
   ```

   which returns only the relevant atoms in $O(\log N)$ time.

5. **Grounded leaves as atomic nodes.** Each grounded predicate call is stored as

   ```
   ProgramNode(id : PREDICATE_NAME)
   ```

   with zero outgoing edges. This preserves ENF cost semantics and requires no additional structure.

6. **Local rewrites and normalization.** ENF and CENF normalization rules are expressed as rewrite patterns over `ProgramNode` and `Edge` atoms. Because each rule touches only a constant number of atoms, each rewrite is $O(1)$ or $O(k)$ in the size of the affected region.

7. **Deme storage.** Each deme's population is a set of root-node atoms `ProgramNode(root_id)`. Traversal, variation, serialization and garbage collection all operate via MORK's atom and link indices, ensuring locality and parallelism.

By representing gCoDD programs as hashed, pointer-shared atoms plus indexed links, MORK achieves:

- Constant-time equality and deduplication via hash-consing.
- Local, $O(k)$ DAG rewrites for crossover, mutation and ENF/CENF.
- Efficient pattern mining and factor-graph inference by matching only the necessary atoms.
- Seamless scaling across distributed Atomspaces using Hyperon's routing of sub-graph queries.

5 Translating gCoDD Programs into Logical Form

To support both crisp (symbolic) and probabilistic reasoning over gCoDD programs in MORK, we will need to transform the DAG representation into a set of logic facts and weighted relations. This need not be done explicitly in a batch process (no need to duplicate gCoDD programs in logical forms in the Atomspace!); rather PLN or other logic engines can perform these transformations automatically and ad hoc in the course of doing reasoning on gCoDD programs. We pedantically outline the relationships involved here just to give a sense of what sort of predicates PLN would be dealing with when explicitly logically reasoning about gCoDD programs.

5.1 Extracting Crisp Logic Facts

1. For each DAG node `ProgramNode(id : SYMBOL)`, emit a fact

   ```
   node(id, SYMBOL).
   ```

2. For each edge atom `Edge(parent, child, Pos)`, emit

   ```
   edge(parent, child, Pos).
   ```

3. For each guard or meta-guard, emit

   ```
   guard(id, GuardName).
   ```

4. Store these facts in MORK's logic index. Queries such as

   ```
   find node(N,_), edge(N,C,_)
   ```

   run in $O(\log N)$ time per match.

5.2 Generating Probabilistic Annotations

1. Assign each node and edge a weight from the chosen quantale Q:

   ```
   weight(node, id, W).
   weight(edge, parent, child, W2).
   ```

2. These weights serve as factor potentials in a factor graph: each `node/3` and `edge/4` fact becomes a factor of arity 1 or 2.

3. Use MeTTa's (proposed) Quantales.FactorGraphs API to add variables and factors:

```
fg-add-variable G id Q.
fg-add-factor   G (node-factor id)   [id].
fg-add-factor   G (edge-factor P C) [P,C].
```

4. Inference (crisp or probabilistic) is then `fg-infer G` in $O(E)$ time.

5.3 Efficient Transformation in MORK

- **Single pass indexing:** Use a MORK `find` query over all `ProgramNode` and `Edge` atoms to generate logic facts in one pass.
- **Hash-cons sharing:** Because identical sub-DAGs share node atoms, duplicate facts are automatically collapsed.
- **Incremental updates:** When the gCoDD changes locally (via crossover or mutation), only the affected node and edge atoms are re-indexed and re-emitted, in $O(k)$ time.
- **Parallel export:** Multiple demes can export their logic facts concurrently to separate logic contexts or distributed factor graphs, leveraging Hyperon's Atomspace routing.

This transformation yields a unified logical view of gCoDD programs, enabling fast symbolic queries and efficient probabilistic inference without leaving the MORK environment.

6 Elegant Normalization of gCoDD Programs in MORK

6.1 Summary of ENF and CENF

Let us briefly review the core ideas of *Elegant Normal Form* (ENF) and its extension *Correlation-Adapted ENF* (CENF), highlights their desirable properties, and overviews the algorithms.

6.1.1 Elegant Normal Form (ENF)

ENF is a canonical form for program trees – created by Craig Holman with a focus on Boolean programs, and here applied to Combinatory Decision DAGs – that ensures:

- **Uniqueness.** Every behaviorally equivalent program has exactly one ENF DAG. This yields constant-time equality checks.

- **Dead-code elimination.** Redundant subexpressions are removed by local rewrite rules.
- **Maximal sharing.** Identical subgraphs are merged via hash-consing, reducing memory and avoiding repeated evaluation.
- **Locality.** All rewrite rules (promote, distribute, sort children, etc.) apply to a small neighborhood of the DAG, in time proportional to the size of that region.
- **Occam bias.** Program complexity corresponds directly to node count, supporting a simple length penalty in fitness.

ENF Algorithm Outline

1. *Hash-consing.* Build a raw DAG from the input program, merging identical subtrees.
2. *Local rewrites.* Repeatedly apply the 8 ENF rewrite rules (promote, distribute, collapse, etc.) until no changes occur.
3. *Child ordering.* Sort children of commutative nodes by their hash identifiers to enforce a unique order.

6.1.2 Correlation-Adapted ENF (CENF)

CENF is a new proposal I have made, which extends ENF by grouping strongly co-occurring guard bits into *meta-guards*, with an aim of improving probabilistic modelling:

- **Meta-guards.** Clusters of guards that frequently appear together are collapsed into single guard nodes.
- **Reduced dimensionality.** Fewer guard variables in the EDA model leads to faster learning and sampling.
- **Maintained canonicity.** CENF remains a unique normal form: meta-guard definitions are deterministic given the clustering threshold.
- **Incremental updates.** Only clusters whose co-occurrence counts exceed a threshold trigger new meta-guards, enabling efficient online adaptation.

CENF Algorithm Outline

1. *Co-occurrence mining.* Measure pairwise frequencies of guard edges in the population, weighted by fitness.
2. *Clustering.* Group guards into clusters whenever their co-occurrence exceeds a threshold.

3. *Meta-guard rewrite.* For each cluster, replace occurrences of its guards by a single `MetaGuard` node in the DAG.

4. *ENF re-normalization.* Apply ENF rewrites to restore canonicity after meta-guard insertion.

6.2 Efficient CENF Implementation in MORK for gCoDD

Below is an outline of a MORK-efficient pipeline for gCoDD DAGs:

6.2.1 Guard Co-occurrence Mining

1. Each gCoDD DAG stores guard edges as atoms of the form `Guard(id)`.

2. Use MORK pattern-mining to collect count atoms `CountPair(id1,id2,n)` for each co-occurring guard pair $(id1, id2)$, weighted by program fitness.

3. Maintain these counts incrementally as new individuals enter the deme.

6.2.2 Cluster Guards into Meta-guards

1. Extract the adjacency list of `CountPair` atoms in MORK.

2. Run a lightweight clustering (e.g. single-linkage or threshold pruning) purely in MORK:

   - Define atoms `Cluster(c,[id1,id2,...])`.
   - Merge any two clusters if any count exceeds a threshold T.

3. The result is a set of `Cluster` atoms, each naming a new meta-guard identifier.

6.2.3 Local CENF Rewrite Rules

1. For each `Cluster(c,Gs)` atom, define a rewrite pattern:

   > match any sub-DAG with guard atoms in set `Gs`, and replace all occurrences by a single atom `MetaGuard(c)` whose children are the shared sub-DAG context.

2. Implement this as a MORK local rewrite:

   - `find` all occurrences of the pattern in the Atomspace.
   - For each match, `remove` the individual `Guard(id)` nodes and `add` the single `MetaGuard(c)` node.

3. These rewrites preserve ENF confluence and remain $O(k)$ in the size of each matched region.

6.2.4 Incremental and Parallel Execution

- Only clusters whose counts changed since the last CENF pass are reprocessed.
- MORK's internal index structure routes pattern queries directly to affected Atomspaces.
- Multiple CENF passes can run in parallel on disjoint subgraphs, exploiting MORK's lock-free local rewriting.

6.2.5 Complexity and Scaling

- Guard mining: $O(E)$ per generation where E is total guard edges.
- Clustering: near $O(G \log G)$ where G is number of distinct guards (typically small).
- Rewrite passes: $O(k)$ per cluster match, with k the subgraph size.

This CENF implementation leverages MORK's pattern-mining, atom-based clustering, and local rewrite primitives to merge correlated guards efficiently, keeping gCoDD programs canonical and amenable to fast variation and EDA.

7 Representing Crossover and Mutation as Quantale Operations

Let $(Q, \otimes, \oplus, e)$ be a commutative quantale. A program is viewed as a function

$$m : X \to Q,$$

mapping contexts or knob assignments $x \in X$ to a quantale element $m(x)$. Variation operators combine parent models pointwise in Q.

7.1 Crossover as Join and Product

Given two parent models $m_1, m_2 : X \to Q$, define:

- **Join-crossover:**

  $$m_{\text{child}}(x) \;=\; m_1(x) \;\oplus\; m_2(x).$$

  This selects the weaker or more permissive prediction of the two parents.

- **Product-crossover:**

  $$m_{\text{child}}(x) \;=\; m_1(x) \;\otimes\; m_2(x).$$

  This selects the intersection or strongest common prediction.

7.2 Mask-based Crossover for Creative Mixing

To achieve classical GA creativity, we introduce a mask function

$$p : X \to \{0, e\} \subseteq Q,$$

where $p(x) = e$ means "inherit from parent 1" at x, and its complement is

$$\bar{p}(x) \;=\; e \multimap p(x),$$

the residuum of p relative to the unit e. Then the masked crossover child is

$$m_{\text{child}}(x) \;=\; \big(p(x) \;\otimes\; m_1(x)\big) \;\oplus\; \big(\bar{p}(x) \;\otimes\; m_2(x)\big).$$

For each x, this picks exactly one parent's value, realizing the building-block mixing power of genetic crossover within the elegant quantale context.

7.3 Mutation as Additive and Multiplicative Perturbation

Let $\delta : X \to Q$ be a small random perturbation. Define two basic mutation operators:

- **Additive (weakening) mutation:**

  $$m^{+}(x) \;=\; m(x) \;\oplus\; \delta(x).$$

  This relaxes predictions where δ is large.

- **Multiplicative (sharpening) mutation:**

  $$m^{\times}(x) \;=\; m(x) \;\otimes\; \delta(x).$$

  This sharpens predictions where δ has high support.

7.4 Implementation Sketch in MORK/MeTTa

In MeTTa or MORK, these operators become simple rewrites on the DAG:

```
(defn crossover-join (Q m1 m2)
  (fn (x) ((Q :join)  (m1 x) (m2 x))))

(defn masked-crossover (Q mask m1 m2)
  (let ((join  (Q :join))
        (times (Q :times))
        (resid  (Q :resid))
        (unit   (Q :unit)))
    (fn (x)
      (join
        (times (mask x)              (m1 x))
        (times (resid unit (mask x)) (m2 x))))))
```

```
(defn mutate-plus  (Q m delta)
  (fn (x) ((Q :join)  (m x) (delta x))))

(defn mutate-times (Q m delta)
  (fn (x) ((Q :times) (m x) (delta x))))
```

Each operator inserts at most two quantale nodes per affected subgraph, yielding $O(k)$ local rewrites.

8 Factor Graph-based Estimation of Distribution in MOSES

To model and sample high-fitness programs from a population of gCoDD/CENF individuals, we embed a factor graph in MORK using the (proposed) `Quantales.FactorGraphs` API. Unlike chained PLN control which applies inference rules sequentially and scales poorly with population size (though this can be improved, potentially, with sophisticated history-guided statistical inference control!), factor graph message passing exploits sparse, local dependencies and parallelism to support efficient EDA.

8.1 Variable and Factor Definitions

Each guard or meta-guard bit and each numeric knob becomes a random variable in the graph. In MORK:

```
fg-add-variable G var-id Q_prob
```

where `G` is the graph, `var-id` is the node identifier, and `Q_prob` is the probabilistic quantale (e.g. [0,1], times=*, join=max).

Dependencies among variables are encoded as factors. For each detected dependency (binary or n-ary):

```
fg-add-factor G factor-func [var1,var2,...,varN]
```

Here `factor-func` is a potential function mapping assignments of the listed variables to a quantale weight.

8.2 Pattern Mining for N-ary Factors

Hyperon's pattern-mining primitives let us discover frequent multi-way linkages among guard bits:

1. *Mine patterns*: Use a query such as `find GuardPattern(ids=[g1,g2,...,gN])` weighted by program fitness counts.
2. *Define potentials*: Convert pattern frequencies into a potential function `phi(g1,...,gN)` in `Q_prob`.

3. *Insert n-ary factor*: `fg-add-factor G phi [g1,...,gN]`

This captures high-order correlations that binary-dependency-based reasoning would miss, improving generative accuracy.

8.3 Inference and Sampling

After all variables and factors are added, perform belief propagation:

```
fg-infer G
```

This runs in time linear or near-linear in the number of factors and variables. To draw new programs, sample assignments:

```
fg-sample G    % returns a vector of var-id -> value
```

8.4 Generating New Programs

Each sampled assignment is mapped back to a gCoDD program by:

1. Filling the base sketch's knobs with the sampled values.
2. Applying ENF/CENF normalization to obtain a canonical DAG.

These programs form the next generation for evaluation.

8.5 Efficiency in MORK

- **Incremental updates**: When individuals change, only affected variables and factors are re-added or updated in $O(k)$ time.
- **Locality**: Pattern mining and factor insertion operate on local subgraphs via indexed `find` queries.
- **Parallelism**: Independent demes build and infer their factor graphs concurrently in separate Atomspaces.
- **Scalability**: N-ary factors reduce model error with fewer variables, and message passing scales to large populations where chained PLN would struggle.

9 Quantales in the MOSES Framework

Let us now briefly indulge ourselves in a more abstract view of all this.

Within MOSES, according to the proposed implementation design, we leverage several distinct commutative quantales to structure different algorithmic phases. Below we summarize each quantale, its operations, and how they interrelate.

9.1 Logic and Truth-Value Quantales for PLN

PLN uses quantales to represent uncertain truth values. A common choice is

$$Q_{\text{PLN}} = \big([0,1],\ \otimes = \times,\ \oplus = \max,\ e = 1\big),$$

where:

- $\otimes$ (product) models conjunction of evidence,
- $\oplus$ (join) models disjunction or aggregation of support,
- $e = 1$ is the unit (certain truth).

The residuum $a \multimap b = \min(1, b/a)$ encodes implication. These operations support both crisp and probabilistic inference via factor graphs and message passing.

9.2 Variation Quantale for Genetic Operators

For crossover and mutation we treat program variation as algebra in a separate quantale

$$Q_{\text{var}} = \big(Q,\ \otimes,\ \oplus,\ e\big),$$

where Q may be any domain supporting:

- *Join-crossover:* $m_{\text{child}} = m_1 \oplus m_2$ selects a permissive mix,
- *Product-crossover:* $m_{\text{child}} = m_1 \otimes m_2$ selects common structure,
- *Mask-based crossover:* uses a mask p with complement $\bar{p} = e \multimap p$ to define
$$m_{\text{child}} = (p \otimes m_1) \oplus (\bar{p} \otimes m_2),$$
- *Mutation:* additive $(m \oplus \delta)$ or multiplicative $(m \otimes \delta)$ perturbations.

These operations act pointwise on models $m : X \to Q$ and are implemented as local DAG rewrites in MORK.

9.3 Program-Space Quantale for gCoDD

The space of gCoDD programs itself forms a quantale

$$Q_{\text{prog}} = \big(\mathcal{P}(\mathit{Prog}),\ \oplus_{\text{prog}},\ \otimes_{\text{prog}},\ e_{\text{prog}}\big),$$

where:

- $\oplus_{\text{prog}}$ is nondeterministic choice (union of program sets),
- $\otimes_{\text{prog}}$ is sequential composition of programs,
- e_{prog} is the identity program (no-op),
- arbitrary joins model generalised choice among many programs.

This algebraic structure supports reasoning about sets of candidate programs and implements program combination operations.

9.4 Weakness Measure as a Quantale Morphism

When learning weak models we equip programs with a complexity cost. Define

$$Q_{\text{cost}} = \big(\mathbb{R}_{\geq 0},\ \oplus_{\text{cost}} = \min,\ \otimes_{\text{cost}} = +,\ e_{\text{cost}} = 0\big).$$

A *weakness measure*

$$w : Q_{\text{prog}} \to Q_{\text{cost}}$$

is a quantale morphism satisfying

$$w(p \otimes_{\text{prog}} q) = w(p) \oplus_{\text{cost}} w(q), \quad w(p \oplus_{\text{prog}} q) = w(p) \wedge w(q).$$

This enforces an Occam-style bias: program composition adds cost, choice takes the simpler alternative.

9.5 Interplay of Quantales

- **PLN inference** uses Q_{PLN} for propagating truth-values.
- **GA operators** use Q_{var} to mix and perturb program-models.
- **Program algebra** in Q_{prog} defines how programs are combined and enumerated.
- **Weakness bias** via $w : Q_{\text{prog}} \to Q_{\text{cost}}$ integrates model simplicity into selection.

Each quantale is implemented in MORK as a set of atoms and local rewrite rules, enabling scalable, uniform reasoning across inference, variation, and program-space exploration.

9.6 CDiagram of Quantale Relationships

The following simple diagram summarizes how the various quantales in MOSES interact:

$$\begin{tikzcd}
Q_{\text{PLN}} & & Q_{\text{var}} \\
& Q_{\text{prog}} \arrow[ul, dashed, "inference"] \arrow[ur, dashed, "variation"] \arrow[r, "w"] & Q_{\text{cost}}
\end{tikzcd}$$

Figure 2: Quantales and morphisms in the MOSES framework. Dashed arrows denote usage of the logic and variation quantales in program construction; the solid arrow w is the weakness-measure morphism from program space to cost.

9.7 Functorial Adaptation via Quantale Morphisms

In MOSES each quantale is an object in a category of quantales and quantale morphisms. By defining functorial mappings between quantales, we obtain a uniform mechanism to adapt variation, inference, and hyperparameters across the entire framework.

9.7.1 Cost-to-Variation Morphism

Let

$$w : Q_{\mathrm{prog}} \to Q_{\mathrm{cost}}$$

be the weakness-measure morphism, and let

$$f : Q_{\mathrm{cost}} \to Q_{\mathrm{var}}$$

be a quantale morphism that maps program cost to variation strength. Composing these,

$$Q_{\mathrm{prog}} \xrightarrow{w} Q_{\mathrm{cost}} \xrightarrow{f} Q_{\mathrm{var}}$$

yields a setting where simpler programs receive milder mutations and complex programs receive stronger perturbations automatically.

9.7.2 Inference-to-Variation Morphism

Let

$$Q_{\mathrm{PLN}} \xrightarrow{g} Q_{\mathrm{var}}$$

be a morphism that maps truth-value confidence in learned dependencies to mask-based crossover parameters. High confidence links produce masks that preserve those bits more often, while low confidence leads to more exploratory mixing.

9.7.3 Program-to-Inference Morphism

A mapping

$$h : Q_{\mathrm{prog}} \to Q_{\mathrm{PLN}}$$

interprets program performance or complexity statistics as initial potentials in the factor-graph model. This functor seeds the EDA inference phase with a prior adapted to the current population distribution.

9.7.4 Hyperparameter Quantale and Natural Transformations

Introduce a small quantale of hyperparameters

$$Q_{\mathrm{hyper}}$$

whose elements index settings such as mutation rate, selection pressure, and clustering threshold. Define monoidal actions (natural transformations)

$$Q_{\mathrm{hyper}} \times Q_{\mathrm{cost}} \;\to\; Q_{\mathrm{var}}, \quad Q_{\mathrm{hyper}} \times Q_{\mathrm{PLN}} \;\to\; Q_{\mathrm{PLN}}$$

so that increasing an "exploration" hyperparameter uniformly inflates mutation strength and softens inference joins. By adjusting a single element of Q_{hyper}, the entire MOSES pipeline adapts coherently.

9.7.5 Task-Specific Quantale Functors

For different problem domains, one can select alternate instantiations of each quantale (for example, crisp vs. fuzzy truth values or min-plus vs. max-plus variation). Functors between these quantales allow swapping an entire column of the quantale diagram without altering the core MOSES implementation.

9.7.6 Summary

By treating the mappings between program, cost, variation, and inference quantales as first-class morphisms, MOSES gains:

- A principled method to propagate complexity and confidence measures through all phases.
- A uniform mechanism to tune hyperparameters system-wide via Q_{hyper}.
- The ability to swap in domain-specific quantales via functors, reusing the same infrastructure.

10 Decentralized Agents for MOSES Demes in Mettacycle

We have not yet addressed the top level of the MOSES framework, the deme management system. Implementation-wise, this does not have so much to do with MORK, but rather can leverage the higher-level MettaCycle and Hyperon frameworks into which MORK is embedded.

In a scalable MOSES implementation in Hyperon, each deme should be managed by a lightweight, autonomous agent running in its own Atomspace. Mettacycle's integration of MORK and Rholang provides a natural substrate for these agents to coordinate, migrate, and evolve gCoDD populations without centralized control.

10.1 Agent Responsibilities

Each MOSES agent ("deme agent") carries out the following tasks:

- **Local Population Storage.** The agent's Atomspace holds the current deme's gCoDD roots, ENF/CENF state, factor-graph model, and metadata (exemplar fitness, resource usage).
- **Evolutionary Loop.** Within its Atomspace the agent runs the MOSES loop: sample knob vectors, construct and normalize programs, evaluate

fitness (via MeTTa $\rightarrow$ Rholang execution), update EDA model, and apply quantale-based variation.

- **Peer Discovery.** Agents publish a small summary tuple `(demeId, exemplarScore, load)` on a shared Rholang pub-sub channel.
- **Load Balancing and Migration.** Under-loaded agents subscribe to peer summaries and volunteer to host migrations of high-potential exemplars or entire subpopulations; over-loaded agents may offload individuals to peers.
- **Retirement and Spawning.** Agents retire when their exemplar falls below a global performance threshold or when a superior exemplar's clone has stabilized; new agents spawn upon detection of novel sketches or resource availability.

10.2 Embedding in Mettacycle

Mettacycle couples MORK's Atomspace with Rholang processes to realize this architecture:

- **Atomspace as State Store.** Each Rholang process embeds a MORK instance whose atoms and links persist the deme's graphs and factor models.
- **Rholang Pub-Sub.** Deme agents communicate summaries and migration requests over Rholang channels, leveraging built-in concurrency and fault tolerance.
- **Transactional Updates.** Mettacycle's transactional semantics ensure that local evolution steps (e.g. CENF normalization, factor-graph inference) occur atomically, avoiding inconsistent deme views.
- **Dynamic Scaling.** New agents can be instantiated via Rholang's `deploy` primitive; resource monitoring routines can trigger scale-out or scale-in of deme agents.
- **Security and Sandboxing.** Grounded predicate calls and fitness oracles run in sandboxed Rholang contexts, ensuring deme isolation and safe execution of user-provided code.

This decentralized, peer-to-peer deme management leverages Mettacycle's strengths–local Atomspace rewrites, Rholang concurrency, and on-chain coordination–to implement MOSES demes that scale naturally across distributed clusters.

11 Complexity Analysis of MOSES Steps

We now give a hand-wavy analysis of the complexity involved in this proposed MOSES implementation.

Let N be the batch size per generation, n the number of knob/guard variables, k the size of a local subgraph, and V, E the number of variables and factors in the factor graph.

- **Sampling from EDA model.**
  - Ancestral sampling on a Chow-Liu tree: $O(n)$ per sample, total $O(N\,n)$.
  - Loopy belief propagation: $O(E)$ per inference, plus $O(n)$ per sample.
- **Program Construction and Canonicalization.**
  - Build raw DAG from knob vector: $O(L)$, where L is sketch size.
  - ENF normalization (hash-consing + local rewrites): $O(L \log L)$ worst-case, empirically near $O(L)$.
  - CENF clustering: co-occurrence mining $O(G)$, clustering $O(G \log G)$ for G distinct guards, and local rewrites $O(k)$ per cluster.
- **Evaluation.**
  - Expanding to MeTTa-IL: $O(L)$.
  - Fitness oracle cost: application dependent, denote T_{oracle}.
- **Selection.**
  - Sorting or partial selection of N scores: $O(N \log N)$ or $O(N)$ with quickselect.
- **Model Update.**
  - Mutual information computation: $O(n^2)$ for pairwise counts, or $O(N\,n)$ incrementally.
  - Chow-Liu tree construction: $O(n \log n)$.
  - Block model clustering: $O(n \log n)$ or spectral cost if used.
- **Factor Graph Inference.**
  - Belief propagation: $O(E)$ per generation.
  - Sampling assignments: $O(n)$ per sample.
- **Variation Operators.**
  - Quantale-based crossover/mutation: $O(k)$ per offspring for local DAG rewrites.
  - Mask generation and mutation: $O(k)$ for mask nodes.
- **Deme Management.**
  - Pub-sub advertise/discover: $O(1)$ network messages per deme.

– Migration and retirement decisions: $O(D)$ for D demes.

Overall per-generation cost (excluding oracle) is dominated by

$$O\big(N\,n + L\log L + n^2 + E + N\,k + D\big).$$

Since $n, k \ll N$ and $E = O(n)$ for tree models, MOSES in MORK scales roughly as $O(N\,n + N\,k + L\log L + n^2)$, with all major steps parallelizable across demes and Atomspace shards. (Note that for program trees of the size pertinent in practice, n will never be more than a couple hundred, so the quadratic dependence on n is not a practical problem.)

12 When Will EDA's Accelerate Evolutionary Convergence?

12.1 Accelerated Convergence in MOSES via Factor Graph EDA

This section examines how the use of factor graph-based estimation of distribution (EDA) in MOSES can improve convergence rates beyond those predicted by the Weakness Schema Theorem (from the "Weakness Is All You Need" paper), by capturing and exploiting higher-order dependencies among program components.

12.1.1 Weakness Schema Theorem

The Weakness Schema Theorem generalizes Holland's schema theorem to the setting of weakness-based genetic search. It states that, under proportional selection, crossover and mutation, the expected frequency of any schema S after one generation satisfies

$$E\big[m(S, t+1)\big] \;\geq\; \frac{\bar{w}(S)}{\bar{w}}\, m(S, t)\,(1-\epsilon),$$

where $m(S, t)$ is the schema count at generation t, $\bar{w}(S)$ its average fitness, $\bar{w}$ the population mean fitness, and ϵ a disruption term bounded by crossover and mutation rates. Iterating this bound yields a geometric growth rate for above-average schemas.

12.1.2 Limitations of Classical GA Convergence

The theorem predicts schema growth only in proportion to individual fitness differences and survival probabilities. However:

- High-order schemas (involving many bits or program fragments) suffer exponential disruption under blind crossover.

- Epistatic interactions may prevent simple schemata from correlating with fitness.
- Deceptive landscapes can mislead proportional selection into suboptimal basins.

12.1.3 EDA-Driven Acceleration in MOSES

By learning a generative model over CENF-normalized gCoDD programs, MOSES overcomes these limitations in cases where program structure exhibits exploitable patterns.

Block Decomposability When the fitness function factors into largely independent subcomponents ("building blocks"), a factor graph captures these as disjoint factors. Sampling from the model preserves entire blocks, avoiding schema disruption and yielding exponential speedups over bitwise crossover.

Conditional Independence and Low Treewidth If program variables (guards, knobs) form a graph of low treewidth, the Chow-Liu tree or n-ary factor model approximates the true joint distribution closely. Belief propagation then samples high-fitness assignments in $O(n)$ time, accelerating convergence relative to $O(2^k)$ search in k-bit schema space.

Handling Epistasis and Deception Factor graphs learn direct potentials for interacting groups of variables, so that epistatic effects and deceptive traps are modeled explicitly. This avoids reliance on short schemata and reduces the disruption term ϵ in the schema theorem by preserving learned interactions.

Scalability of Factor Graph PLN Inference Chained PLN inference on multivariate dependencies grows combinatorially with rule chaining. In contrast, factor graph message passing in a quantale-valued model runs in time proportional to the number of factors and variable cardinalities. This enables scalable EDA even in large demes.

12.1.4 Summary of Acceleration Conditions

The EDA aspect of MOSES accelerates convergence beyond the Weakness Schema Theorem when:

- The problem admits modular or block-structured programs.
- Variable interactions obey low-order dependencies.
- Epistatic or deceptive interactions can be captured as factors.
- Factor graph inference remains tractable (low treewidth, limited factor arity).

Under these conditions, MOSES's factor graph EDA drives schema growth at rates exceeding those guaranteed by classical GA analysis.

12.2 Accelerating Convergence via Adaptive Mask-Based Crossover

This section examines how adapting the mask distribution in mask-based crossover–using an EDA model such as factor-graph inference–can reduce schema disruption and accelerate convergence beyond the guarantees of the Weakness Schema Theorem (Weakness-Theory).

12.2.1 Weakness Schema Theorem Recap

The Weakness Schema Theorem states that for any schema S, its expected count $E\big[m(S,t+1)\big]$ in the next generation satisfies

$$E\big[m(S,t+1)\big] \;\geq\; \frac{\bar{w}(S)}{\bar{w}}\, m(S,t)\;(1-\epsilon),$$

where $\bar{w}(S)$ is the average fitness of programs matching S, $\bar{w}$ is the population mean fitness, and ϵ is a disruption term depending on crossover and mutation rates. Schemas of above-average fitness thus grow geometrically, but high-order schemata suffer from large ϵ and may be disrupted rapidly.

12.2.2 Mask-Based Crossover Disruption

In classical n-point or uniform crossover, ϵ scales with the defining length $\delta(S)$ of S. With a random mask $p(x) \sim \text{Bernoulli}(1/2)$, the probability that a schema spanning k decision bits is preserved is $(1/2)^{k-1}$, yielding

$$\epsilon \approx 1 - 2^{-(k-1)}.$$

High-order schemas ($k \gg 1$) are thus almost always broken.

12.2.3 Adaptive Masks via EDA

An EDA model over guard and knob variables can learn which subsets of decision bits (schema building blocks) co-occur in high-fitness programs. By sampling masks that respect these learned linkages, we obtain:

- **Reduced disruption.** Masks align with schema boundaries, so the probability of breaking a good schema of size k becomes

$$\Pr(\text{mask splits } S) \;\ll\; (1/2)^{k-1}.$$

- **Enhanced schema growth.** The effective disruption term ϵ' in

$$E\big[m(S,t+1)\big] \;\geq\; \frac{\bar{w}(S)}{\bar{w}}\, m(S,t)\;(1-\epsilon')$$

  is significantly smaller, yielding faster exponential growth of $m(S,t)$.

12.2.4 When EDA Masks Help

Adaptive mask-based crossover accelerates convergence especially when:

1. **Block structure exists.** The fitness function decomposes into largely independent subprograms (building blocks). EDA learns these blocks and masks preserve them intact.
2. **Low-order dependencies.** Variables exhibit strong low-arity correlations that a factor graph or Chow-Liu tree can capture accurately.
3. **Epistatic interactions.** Important high-order interactions among bits are modeled as factors, so the mask sampler avoids breaking these epistatic schemas.
4. **Deceptive landscapes.** Learned masks steer search away from deceptive recombinations by preserving known good substructures.

Under these conditions, adaptive masks transform mask-based crossover into a schema-preserving operator, reducing the disruption term below the bound assumed in the Weakness Schema Theorem and yielding convergence rates that surpass classical GA predictions.

12.3 Applicability of EDA and Adaptive Masks to Learning Goals

To make the above abstract points more concrete, we now speculatively consider four example program learning tasks and attempt to qualitatively assess how factor-graph EDA and adaptive mask-based crossover can accelerate convergence.

1. Learning a new inference chaining heuristic
2. Learning tweaks to the CENF algorithm
3. Learning a navigation routine for dynamic environments
4. Learning a dialogue control routine for a novel social context

12.3.1 Learning a New Inference Chaining Heuristic

- **Knobs and guards.** Represent the heuristic as an ordered sequence or decision tree over inference rules, with guard bits testing local proof-state features.
- **EDA suitability.** Recurring subchains of rules form ?building blocks?. A Chow?Liu tree or factor graph captures dependencies between rule choices at different positions, preserving high-fitness subsequences when sampling.

- **Adaptive masks.** Masks that select entire contiguous subsequences or subtrees preserve effective rule chunks. Learning mask distributions from fitness data reduces the probability of breaking useful chains.

- **Conclusion.** Inference chaining with clear recurring subchains is highly amenable to both EDA and mask-based crossover.

12.3.2 Learning Tweaks to the CENF Algorithm

- **Knobs and guards.** Meta-parameters such as clustering thresholds, minimum support, and rule ordering are numeric or categorical knobs.

- **EDA suitability.** The parameter space is modest in size. Factor graphs learn low-order dependencies between parameters, enabling focused sampling of high-fitness configurations.

- **Adaptive masks.** Masks over groups of related parameters (e.g. all clustering parameters) swap successful parameter subsets intact, avoiding disruptive single-parameter changes.

- **Conclusion.** CENF tuning is strongly amenable to EDA and adaptive masks, yielding faster convergence than blind search.

12.3.3 Learning a Navigation Routine for Dynamic Environments

- **Knobs and guards.** Policy is a decision DAG with guards on sensor thresholds and leaf actions. Knobs encode threshold values and action choices.

- **EDA suitability.** Reusable behavior modules (e.g. wall-follow) form subgraphs whose presence correlates with fitness. Factor graphs capture these modules and sample new policies that combine them.

- **Adaptive masks.** Masks that select subtrees corresponding to context-action modules preserve entire routines. Learning mask patterns from performance data maintains robust modules under recombination.

- **Conclusion.** Reactive navigation with modular routines benefits from EDA and adaptive masks, especially when modules are canonically labelled.

12.3.4 Learning a Dialogue Control Routine for a Novel Social Context

- **Knobs and guards.** A control DAG tests dialogue-act types, user sentiment, and context flags. Knobs choose templates or turn-taking strategies.

- **EDA suitability.** Common conversational modules (e.g. agenda-setting) can be fingerprinted and modelled as factors. Sampling preserves effective multi-utterance strategies.

- **Adaptive masks.** Masks that select entire dialogue modules prevent breaking coherent subdialogues. Adaptive masks learned from dialogue success rates maintain high-level strategy structure.
- **Conclusion.** Dialogue control is more challenging due to noise and sparsity, but can still leverage EDA and masks if conversational modules are identified and fitness evaluation is robust.

Tweaking EDA Inference for Dialogue Control . This leads to the question of what modifications to the standard factor-graph EDA might improve performance in difficult, "messy" contexts like this. Some ideas are:

1. **Contextual Variable Design.**
    - Introduce *sliding-window context* variables that capture the last k dialogue acts or semantic frames.
    - Factor graph variables become tuples $(d_{t-k+1}, \ldots, d_t)$ rather than single-step guards.
2. **Hierarchical Factor Graphs.**
    - Build a two-level graph: low-level factors over utterance-level decisions, and high-level factors over dialogue-phase decisions (e.g. *greeting*, *agenda*, *negotiation*).
    - Perform inference first at the high level to select a phase, then condition the low-level inference on that phase.
3. **Adaptive Potential Scaling.**
    - Use reinforcement signals (success/failure rewards) to re-weight factor potentials ϕ_i after each dialogue episode.
    - Apply an exponential moving average to smooth updates:

      $$\phi_i^{(new)} = \alpha\, \phi_i^{(old)} + (1-\alpha)\, \hat{\phi}_i,$$

      where $\hat{\phi}_i$ is the empirical potential from the latest data.
4. **N-Ary Conversational Motifs.**
    - Mine frequent subdialogue patterns of length m (e.g. *greet*→*ask*→*confirm*) as n-ary factors.
    - Add these factors to the graph to capture higher-order dependencies without chaining many binary factors.
5. **Uncertainty-Aware Quantale.**
    - Replace the crisp probabilistic quantale with a fuzzy or evidence-weighted quantale (e.g. $([0,1], \otimes = \min, \oplus = \max)$) to tolerate noisy observations.

    - This yields softer inference that maintains diversity in sampled dialogue strategies.

6. **Online Structure Learning.**
    - Periodically re-mine the graph structure (add or remove factors) based on cumulative dialogue data.
    - Use a threshold on mutual information change to trigger structure updates, keeping the model aligned with evolving dialogue patterns.

These extensions–contextual variables, hierarchical factors, adaptive potential scaling, n-ary motif factors, uncertainty-aware quantales, and online structure learning–could potentially tailor the EDA inference to the complexities of dialogue control, preserving robust conversational modules while adapting to dynamic social contexts. How far we need to go in this direction remains to be determined via experiment!

13 Estimated gCoDD Program Size for PLN Chainer and CENF

In this section we give rough, hand-wavy estimates of the size of gCoDD DAGs (node count) required to implement (1) a PLN chaining inference engine and (2) the CENF algorithm, based on a breakdown of core components and combinator requirements. The goal is just to get a rough sense of how big are the gCoDD program dags we need to worry about evolving, to get a system that is capable of evolving its own "cognitive modules" (which then would allow it to replace and upgrade its own cognitive infrastructure, bit by bit).

13.1 PLN Chainer

A minimal PLN chainer must support:

- **Quantale operations:** product, join, residuum, unit (4 nodes plus associated wiring).
- **Logical connectives:** conjunction, disjunction, implication as combinators (3 nodes).
- **Pattern matching and rule application:** for each inference rule (deduction, induction, abduction, etc.), a combinator tree of size 10–20 nodes. Typical PLN includes 6-8 rules, total ≈ 120 nodes.
- **Control flow:** loop, recursion or fixpoint combinators (map, filter, fold) to apply rules until convergence, ≈ 30 nodes.
- **Data structures:** handling lists or multisets of statements via combinators (cons, nil, append), ≈ 20 nodes.

- **Glue logic:** dispatch, termination checks, confidence updates, ≈ 30 nodes.

Total estimate: $4 + 3 + 120 + 30 + 20 + 30 \approx 207$ nodes. Allowing for shared subgraphs and optimisations, a practical gCoDD may use $\mathbf{15 - 250}$ nodes.

13.2 Correlation-Adapted ENF (CENF)

The CENF pipeline includes:

- **Guard extraction:** atoms for each guard edge, ≈ 1 node per guard.
- **Co-occurrence mining:** counting and sketch combinators, ≈ 20 nodes.
- **Clustering logic:** single-linkage or threshold merge combinators, ≈ 15 nodes.
- **Meta-guard rewrite:** pattern-match and replace rules for each cluster, ≈ 10 nodes per cluster; with 5-10 clusters, ≈ 75 nodes.
- **ENF re-normalization:** the standard 8 ENF rules, ≈ 8 nodes.
- **Control flow:** loop to trigger until fixpoint, ≈ 10 nodes.

Total estimate: $1 * G + 20 + 15 + 75 + 8 + 10$, where G is the number of guards (typically $50 - 100$). For $G = 75$, total $\approx 1 * 75 + 20 + 15 + 75 + 8 + 10 \approx 203$ nodes. In practice, **1500250** nodes suffice.

13.3 Combined System Footprint

A unified MOSES engine with both PLN chaining and CENF normalization can share quantale and ENF rule nodes, reducing duplication. A conservative combined estimate is

$$207\text{ (PLN)} + 203\text{ (CENF)} - 50\text{ (shared)} \approx 360\text{ nodes.}$$

Thus a complete gCoDD implementation of both components typically requires on the order of **300-400** DAG nodes.

13.4 Implications for Scalability

- **Local rewrites:** each variation or normalization step touches $O(k)$ nodes where $k \ll 400$.
- **Shared subgraphs:** common quantale and combinator structures are stored once.
- **Factor graph EDA:** knob and guard variables scale linearly with node count.

This compact size ensures that MOSES operations on PLN chaining and CENF remain efficient and scalable in MORK.