On LLM-generated Logic Programs and their Inference Execution Methods

Paul Tarau

University of North Texas

ICLP'2024

code at `https://github.com/ptarau/recursors/`
demo at `https://deepllm.streamlit.app/`
demo at `https://deep-auto-quests.streamlit.app/`

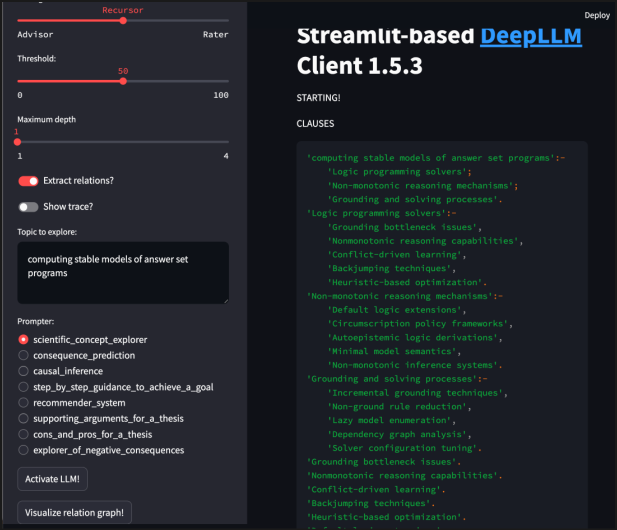
LLMs as Knowledge Stores - ready to be mined

- LLMs are highly compressed knowledge repositories
- the usual *anthropocentric* approach: chat, get knowledge snippet, trust it
- our approach: automate the process, extract knowledge in the form of a Logic Program!
- a focussed, goal-driven approach
- recursive exploration to unwrap knowledge, centered on a topic chosen by the user

Our framework: DeepLLM

- DeepLLM is a system that automates deep step-by step reasoning in an LLM dialog thread by recursively exploring alternatives (OR-nodes) and expanding details (AND-nodes) up to a given depth
- starting from a single succinct task-specific initiator we steer the automated dialog thread to stay focussed on the task by synthesizing a prompt that summarizes the depth-first steps taken so far
- semantic similarity to ground-truth facts or oracle advice from another LLM instance is used to restrict the search space and validate the traces of justification steps returned as focussed and trustable answers
- Use cases:
  - consequence predictions
  - causal explanations
  - step by step guidance to achieve a goal
  - topic-focussed exploration of scientific literature
  - supporting arguments for a thesis
  - recommendation systems

Exploring a topic with DeepLLM



Threshold: 50 (0–100)

Maximum depth: 1 (1–4)

Extract relations? [✓]

Show trace?

Topic to explore:

computing stable models of answer set programs

Prompter:

[✓ scientific_concept_explorer]
consequence_prediction
causal_inference
step_by_step_guidance_to_achieve_a_goal
recommender_system
supporting_arguments_for_a_thesis
cons_and_pros_for_a_thesis
explorer_of_negative_consequences

Activate LLM!

Visualize relation graph!

Streamlit-based DeepLLM Client 1.5.3

STARTING!

CLAUSES

```
'computing stable models of answer set programs':-
    'Logic programming solvers';
    'Non-monotonic reasoning mechanisms';
    'Grounding and solving processes'.
'Logic programming solvers':-
    'Grounding bottleneck issues',
    'Nonmonotonic reasoning capabilities',
    'Conflict-driven learning',
    'Backjumping techniques',
    'Heuristic-based optimization'.
'Non-monotonic reasoning mechanisms':-
    'Default logic extensions',
    'Circumscription policy frameworks',
    'Autoepistemic logic derivations',
    'Minimal model semantics',
    'Non-monotonic inference systems'.
'Grounding and solving processes':-
    'Incremental grounding techniques',
    'Non-ground rule reduction',
    'Lazy model enumeration',
    'Dependency graph analysis',
    'Solver configuration tuning'.
'Grounding bottleneck issues'.
'Nonmonotonic reasoning capabilities'.
'Conflict-driven learning'.
'Backjumping techniques'.
'Heuristic-based optimization'.
```

Figure: DeepLLM app

DeepLLM: abducibles at the limit

- SLD-resolution's clause selection via unification is replaced by LLM-driven dynamic clause head creation with an option of focusing by proximity of embeddings to ground truth facts
- as dialog units are sentences, the underlying logic is propositional
- client-side management (via the API) of the LLM's memory is based on the equivalent of a *goal stack* and a *goal trace* recording our steps on the current search path
- instead of variable bindings, answers are traces of justification steps clearly explaining where they are derived from
- when their depth-limit is reached, the items on the goal stack are interpreted as "abducibles", statements that can be hypothetically assumed and then checked against "integrity constraints"

Generating Dual Horn Clauses: exploring the Known to be False

- A *Dual Horn clause* is a disjunction of literals with at most one negative literal (or exactly one if it is a definite Dual Horn clause).
- A *Dual Horn clause Program* is a conjunction of Dual Horn clauses. $\neg p_0 \vee p_1 \vee \ldots \vee p_n$ in an equivalent implicational form $p_0 \rightarrow p_1 \vee \ldots \vee p_n$
- Prolog-like syntax, with $\rightarrow$ represented as "`=>`" and $\vee$ as "`;`"
- "`s => false`" represents a negated fact the same way as "`s :- true`" would represent a positively stated fact
- small embedded language, SymLP: a compilation algorithm[1] will transform a Dual Horn Program into a definite program placed in module `false`

[1] https://github.com/ptarau/TypesAndProofs/blob/master/symlp/compile_clauses.pro

Dual Horn Programs: Why?

- The objective of Dual Horn programs is to describe (constructively) why something is not true
- $\Rightarrow$ falsify the initiator goal by back-propagating from its negative (undesirable, unwanted, harmful, impossible, etc.,) consequences
- an instance of goal-directed forward reasoning!

Unwanted Consequences

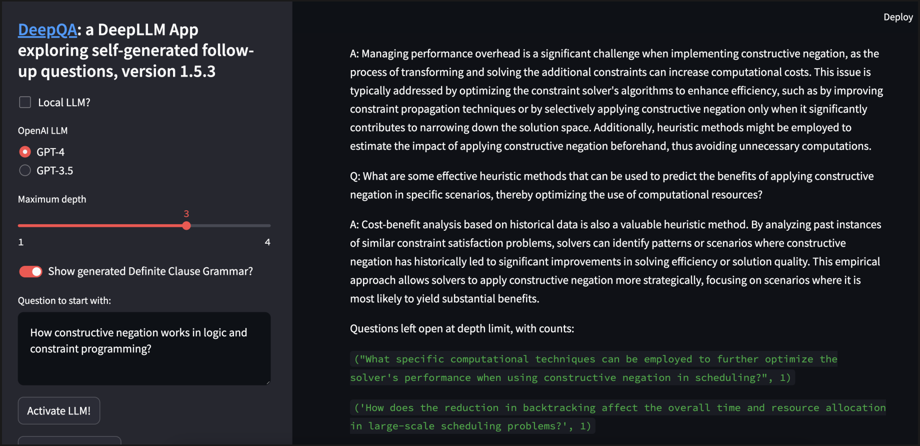
Snippet from a descent level 2 on '*loosing the FED_s independence*':

```
'loosing the FED_s independence' =>
    'Increased political influence on monetary policy'. % level 0
...
'Increased political influence on monetary policy' =>
    'Politicized interest rates';
    'Short-term economic manipulation';
    'Eroded investor confidence'; % level 1
    'Heightened market volatility';
    'Policy-driven inflation risks'.
 ...
 'Eroded investor confidence' => % level 2
    'Market volatility';
    'Capital flight';
    'Reduced foreign investment'.
```

DeepQA: Exploring the Tree of Follow-up Questions

- after started from an initiator question on a topic of the user's choice, the app explores its tree of *follow-up questions* up to a given depth
- as output, it generates a Definite Clause Grammar that can be imported as part of a Prolog program
- the DCG, in generation mode, will replicate symbolically the equivalent of the "stream of thoughts" extracted from the LLM interaction
- the synthesized grammar is designed to generate a finite language (by carefully detecting follow-up questions that would induce loops)
- paths in the question-answer tree are free of repeated answers, which get collected as well, together with questions left open as a result of reaching the user-set depth limit

Recursing of LLM-initiated Follow-up Questions



DeepQA: a DeepLLM App exploring self-generated follow-up questions, version 1.5.3

Local LLM? [ ]

OpenAI LLM
[✓ GPT-4]
GPT-3.5

Maximum depth: 3 (1–4)

Show generated Definite Clause Grammar? [✓]

Question to start with:

How constructive negation works in logic and constraint programming?

A: Managing performance overhead is a significant challenge when implementing constructive negation, as the process of transforming and solving the additional constraints can increase computational costs. This issue is typically addressed by optimizing the constraint solver's algorithms to enhance efficiency, such as by improving constraint propagation techniques or by selectively applying constructive negation only when it significantly contributes to narrowing down the solution space. Additionally, heuristic methods might be employed to estimate the impact of applying constructive negation beforehand, thus avoiding unnecessary computations.

Q: What are some effective heuristic methods that can be used to predict the benefits of applying constructive negation in specific scenarios, thereby optimizing the use of computational resources?

A: Cost-benefit analysis based on historical data is also a valuable heuristic method. By analyzing past instances of similar constraint satisfaction problems, solvers can identify patterns or scenarios where constructive negation has historically led to significant improvements in solving efficiency or solution quality. This empirical approach allows solvers to apply constructive negation more strategically, focusing on scenarios where it is most likely to yield substantial benefits.

Questions left open at depth limit, with counts:

```
("What specific computational techniques can be employed to further optimize the
solver's performance when using constructive negation in scheduling?", 1)
```

```
('How does the reduction in backtracking affect the overall time and resource allocation
in large-scale scheduling problems?', 1)
```

Figure: DeepQA with "*How constructive negation works in logic and constraint programming?*"

Generating Relations

- DeepLLM app has an option to generate from the minimal model of the program a relation graph
- edges: implication links and generalization links
- implication links are extracted directly from the logic program
- generalization links are generated by the LLM via an additional request
- possible other relations: extraction of <subject, verb, object> (SVO) triplets obtained by prompting the LLM to split a complex sentence in simpler ones and extract from each simple sentence an SVO triplet

Reasoning with soft unification on noisy facts

- the minimal models of LLM-generated Horn clause programs encapsulate facts and their consequences elicited from DeepLLM's initiator queries in the form of natural language sentences
- given a logic program that performs symbolic reasoning relying on a ground fact database of such sentences, an interesting form of *abductive reasoning* emerges
- when hitting an undefined ground sentence, intended as a query to match database facts, we can *rely on vector embeddings* of the sentences of the query and the facts in the vector space as a "good enough" match, provided that the semantic distance between them is below a given threshold

Softlog: a Natlog extension with soft unification

```
    def unify_with_fact(self, goal, trail):

        # q = query goal to be matched
        # k = number of knns to be returned
        # d = minimum knn distance
        # v = variable to be unified with the matches

        q, k, d, v = goal
        d = float(d) / 100
        _, answers = self.emb.knn_query(q, k)
        for sent, dist in answers:
            if dist <= d:
                self.abduced_clauses[(q, sent)] = dist
                yield unify(v, sent, trail)
```

```
knn 3.
threshold 70.

quest  Quest Answer:
  knn K, % K is the number passed to the K Nearest Neighbors query
  threshold D,
  ~ Quest K D Answer.
```

more

- by contrast to the usual exact unification based answers, `Softlog` works quite well when the query is *close enough* to a matching entry in the sentence store

```
?- quest 'What did Wilde say about temptation' X?
ANSWER: {'X': 'I can resist anything except temptation said Oscar Wilde.'}

?- quest 'What did Alice say about following advice' X?
ANSWER: {'X': 'I give myself very good advice but I very seldom follow it
               said Lewis Carroll.'}
```

- given the nature of semantic search, surname is enough to find Oscar Wilde and as `Alice` associates with the author Lewis Carroll, soft unification will fetch it from the sentence store

Execution Mechanisms

- fast symbolic fixpoint computation - low polynomial algorithm, scalable to very large LLM generated programs (at deeper recursion levels)
- linear algebraic encodings: possibility to use torch (and GPUs): - see Katsumi Inoue's ICLP24 invited talk on the best algorithms

```
def tp(M, v):
    """
    one step fixpoint operator
    """
    r = M @ v
    return (r >= 1.0).to(torch.float32)
```

- Prolog system (ideally with tabling enabled, e.g., XSB, SWI)

Conclusion

- Generative AI is a major disruptor not just of industrial fields but also a disruptor of research fields, including symbolic AI as we know it and machine Learning itself.
- results produced by dominant ML or NLP techniques as well as work on integration of neural and symbolic systems have become replaceable by much simpler applications centered around LLM queries and RAG systems
- this motivates our effort to “join the disruption” and explore several new ways to elicit the knowledge encapsulated in the LLMs' parametric memory as logic programs, together with an investigation of their optimal inference execution methods
- we hope that this effort has revealed some natural synergies between Generative AI systems and logic programming tools!