DiscASP: A Graph-based ASP System for Finding Relevant Consistent Concepts with Applications to Conversational Socialbots

Fang Li, Huaduo Wang, Kinjal Basu, Elmer Salazar, Gopal Gupta
University of Texas at Dallas
Richardson, USA
{fang.li, huaduo.wang, kinjal.basu, elmer.salazar, gupta}@utdallas.edu

We consider the problem of finding *relevant consistent concepts* in a conversational AI system, particularly, for realizing a conversational socialbot. Commonsense knowledge about various topics can be represented as an answer set program. However, to advance the conversation, we need to solve the problem of finding *relevant consistent concepts*, i.e., find consistent knowledge in the "neighborhood" of the current topic being discussed that can be used to advance the conversation. Traditional ASP solvers will generate the whole answer set which is stripped of all the associations between the various atoms (concepts) and thus cannot be used to find relevant consistent concepts. Similarly, goal-directed implementations of ASP will only find concepts *directly* relevant to a query. We present the DiscASP system that will find the partial consistent model that is relevant to a given topic in a manner similar to how a human will find it. DiscASP is based on a novel graph-based algorithm for finding stable models of an answer set program. We present the DiscASP algorithm, its implementation, and its application to developing a conversational socialbot.

1 Introduction

Conversational AI has been an active area of research, starting from a rule-based system, such as ELIZA [22] and PARRY [7], to the recent open domain, data driven conversational agents like Amazon's Alexa [20]. Early rule-based bots were based on just syntax analysis and thus were limited, while the main challenge of modern machine learning (ML) based chatbots is the lack of "understanding" of the dialogs in the conversation. Current machine learning technology-based chat-bots [14] learn patterns in data from large corpora and compute a response without having any semantic understanding of the conversation utterances. Such an approach limits these chatbots, e.g., they may give an irrelevant response or they cannot provide an explanation for their response. We believe that a chatbot that can converse like a human *has to* employ semantic understanding and reasoning, aside from other AI technologies such as machine learning and natural language processing.

The goal of our research is to develop a *conversational socialbot* that can hold a conversation with a human stranger on topics of general interest such as movies, books, music, pets, family, etc. Our work is inspired by the Amazon Alexa Socialbot Challenge competition [1, 4] in which the authors' participated. A realistic socialbot should be able to understand and reason like a human. In human to human conversations, we do not always tell every detail, we expect the listener to fill gaps through their commonsense knowledge. Thus, modeling commonsense knowledge and commonsense reasoning is an important consideration in developing a socialbot.

As is well known, the human thought process is flexible and non-monotonic in nature, which means *what we believe now may become false in the future with new knowledge*. Given the need for non-monotonic reasoning, we use Answer Set Programming (ASP) [11] as the underlying formalism for commonsense knowledge representation and reasoning. With the use of (i) default rules, (ii) exceptions to defaults, and (iii) preferences over multiple defaults, one can model bulk of human-style reasoning [11], at least those parts that are needed for realizing a socialbot.

One of the problems that need to be addressed in developing a conversational socialbot is inferring the knowledge "relevant" to the topic at hand. That is, concepts that are consistent with the knowledge we possess about the topic. We call these concepts *relevant consistent concepts* for that topic. During a conversation about movies, for example, if the other person says that their favorite movie is Titanic, then we immediately will try to recall all the knowledge we possess about that topic. Thus, we may remember that Titanic's main actors were Leonardo Di Caprio and Kate Winslett, that Titanic got many Academy awards, and that the director of the movie, James Cameron, became very famous. We may then advance the conversation by saying, "Yes, Titanic was a wonderful movie, and it got many Academy Awards." Social conversations drift from a topic to another related topic. For example, after talking a little bit more about Titanic, the conversation may shift to another movie directed by Titanic's director James Cameron such as Avatar. Or, it may shift to another movie of one of the actors in Titanic, e.g., Wolf of Wall Street, in which Leonardo Di Caprio was also the lead actor. This knowledge that we recall about Titanic is the *relevant consistent concept* (RCC) for Titanic. Finding the RCC, given a topic, is an important problem in developing conversational socialbots, as it helps in moving the conversation along. In this paper, we focus on the problem of efficiently finding the RCC(τ) for a topic atom τ, given a commonsense knowledgebase coded in ASP. In this paper, we restrict ourselves to conversations about movies, though the approach applies to any domain.

We present an efficient graph-based algorithm for computing the RCC(τ) for a topic τ given relevant commonsense knowledge coded in ASP. The program can only have headless constraints, while constraints realized through *odd loop over negation* are not permitted. This is done for simplicity: such restricted programs are sufficient for modeling conversational socialbots. Our algorithm computes the set of atoms that represent the concepts that are related to the topic at hand in a given answer set. This set of atoms is computed incrementally, i.e., it is not based on computing the entire answer set. Given a query, we want to not only find out a justification for it, but also the related associated knowledge. E.g., given a query `?- flies(tweety)`, if we infer that Tweety flies because Tweety is a bird due to the rule `flies(X) :- bird(X), not ab_bird(X).`, then we also want to know the associated concept that Tweety has wings from the rule `haswings(X) :- bird(X)`. This is very similar to how humans will bring in relevant concepts in their current working memory through a combination of backward and forward chaining.

Our algorithm (called DiscASP) guarantees that the computed partial answer set is part of a complete answer set dictated by the Gelfond-Lifschitz transform [11]. Traditional tools for ASP such as CLINGO [10] are not suitable for this task, as they compute the entire answer set. The entire answer set can be quite large as the amount of commonsense knowledge can be enormous. Even if we restrict ourselves to conversations about movies, the movie database that a socialbot may have to work with will have information about at least 50 movies (assuming that a person can only keep information about that many movies in their head). In reality, this number can be much larger (e.g., the iMDB movie database has more than 7.5 Million entries). In addition, systems such as CLINGO lose all the information connecting two atoms during the grounding process. Thus, explaining the connections as we leap from topic to topic in a conversation is difficult.

Our contribution is twofold: (i) we present an efficient graph-based algorithm for computing a subset of atoms of an answer set that are consistent with, and relevant to, a topic atom present in the answer set. To the best of our knowledge, the algorithm is novel. The algorithm also provides the logical relationship between the topic atom and any other atom in the computed partial answer set based on the commonsense

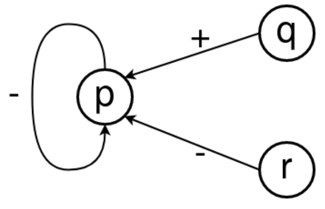
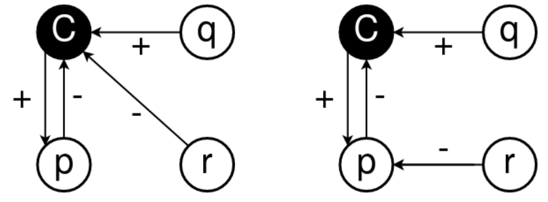


Figure 1: Dep. Graph for Programs 1 & 2



(a) CNR for Program 1 (b) CNR for Program 2

Figure 2: CNRs for Program 1 & 2

knowledge rules provided; (ii) we present a practical way of solving the important problem of computing the set of relevant consistent concepts for a topic atom while developing conversational socialbots.

2 Background

Answer set programming (ASP) [18] is a popular nonmonotonic-logic based paradigm for knowledge representation and reasoning and for solving combinatorial problems. Most ASP solvers employ SAT solver-like technology to find these answer sets. As a result, justification for why a literal is in the answer set is hard to produce. Compared to SAT solver based implementations, graph-based implementations of ASP have not been well studied. Graph-based approaches for ASP [2, 15, 16] are well designed, but their graph representations are complex as they all rely on extra information to map the ASP elements to nodes and edges of a graph. In contrast, our approach uses a much simpler graph representation, where nodes represent literals and an edge represents the relationship between the nodes it connects. Since this representation faithfully reflects the causal relationships, it is capable of producing causal justification for goals entailed by the program. In this section, we introduce the background concepts of our novel graph representation.

Call Dependency Graph: A call dependency graph or dependency graph [16] uses nodes and directed edges to represent call dependency relationships in an ASP rule.

Definition 1 *The dependency graph of a program is defined on its literals s.t. there is a positive (resp. negative) edge from q to p if q appears positively (resp. negatively) in the body of a rule with head p.*

Conventional dependency graphs are not able to represent ASP programs uniquely. This is due to the inability of dependency graphs to distinguish between non-determinism (multiple rules defining a proposition) and conjunctions (multiple conjunctive sub-goals in the body of a rule) in logic programs. For example, the following two programs have identical dependency graphs (Figure 1).

```
%% program 1                          %% program 2
p :- q, not r, not p.                 p :- q, not p. p :- not r.
```

To make conjunctive relationships representable by dependency graphs, we first transform them slightly to come up with a novel representation method. This new representation method, called conjunction node representation (CNR) graph, uses an artificial conjunction node to represent a conjunction of sub-goals in the body of a rule (Figure 2). The conjunction node (colored in black) refers to the conjunctive relation between the incoming edges from nodes representing sub-goals in the body of a rule. Note that a CNR graph is not a conventional dependency graph.

Converting CNR Graph to Dependency Graph: Since CNR graph does not follow the dependency graph convention, we need to convert it to a proper dependency graph in order to perform dependency

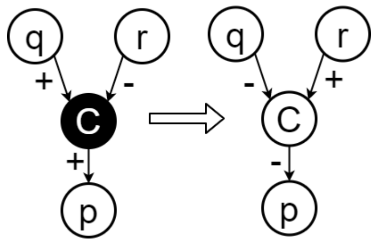
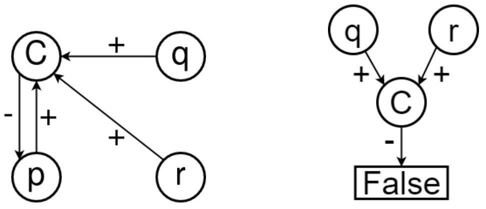


Figure 3: CNR-DG Transformation



(a) Program 4 (b) Program 5

Figure 4: Constraint DG

graph-based reasoning. We use a simple technique to convert a CNR graph to an equivalent conventional dependency graph. We negate all in-edges and out-edges of the conjunction node. This process essentially converts a conjunction into a disjunction. As an example, Figure 3 shows the CNR graph to dependency graph transformation for Program 3. We call such a dependency graph a CNR Dependency Graph. The transformation is a simple application of De Morgan's law. The rule in program `p :- q, not r.` can be represented as: `p :- conj.  conj :- q, not r.` The transformation produces the equivalent rules: `p :- not conj.  conj :- not q.  conj :- r.` Since conjunction nodes are just helper nodes that allow us to perform dependency graph reasoning, we don't report them in the final answer set. The transformation process is quite straightforward, so we do not give any more details.

ASP also allows for special types of rules called constraints. There are two ways to encode constraints: (i) headed constraint where the negated head is called directly or indirectly in the body (e.g., Program 3), and (ii) headless constraints (e.g., Program 4).

```
%% program 3                      %% program 4
p :- not q, not r, not p.         :- not q, not r.
```

Our algorithm models these constraint types separately. For the former one, we just need to apply the CNR-DG transformation directly. Note that the head node connects to the conjunction node both with an in-coming edge and an out-going edge (Figure 4a). For the headless constraint, we create a head node with the truth value as *False*. For simplicity, we currently only permit headless constraints, as these are sufficient to model conversational socialbots.

The reason why we don't treat a headless constraint the same way as a headed constraint is because in the latter case, if head node (`p` in Program 3) is provable through another rule, then the headed constraint is inapplicable. Therefore, we cannot simply assign a false value to its head.

3 A Graph Algorithm for Computing Partial Answer Sets

We have developed DiscASP, a graph-based algorithm, for finding partial answer sets. The philosophy of DiscASP is to translate an ASP program into a CNR dependency graph, which is always constrained by some constraint rules, then try to satisfy the constraints by assigning presumed truth values to the related nodes, until all constraints have been satisfied. At the same time, DiscASP will propagate truth values of the nodes whose truth value has already been determined. Our graph-based approach performs reasoning in an incremental manner. It starts from the constraints in the answer set program and traces along with causal nodes until it finds support through facts (well-founded case) or it detects a cycle through negation (cyclic case). Our algorithm can be thought of as a more general form of the Galliwasp algorithm for query-driven execution of answer set programs [19]. The DiscASP approach is constraint-driven and thus significantly reduces the search space by avoiding the exploration of worlds that are inconsistent with

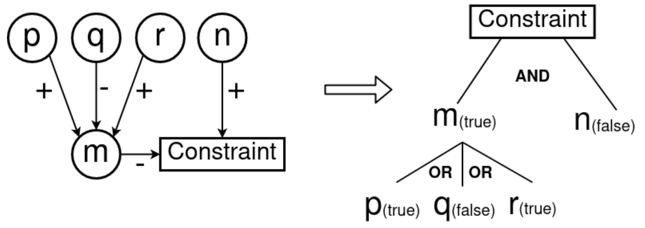


Figure 5: Satisfiability Example

the constraints. Furthermore, the incremental reasoning from constraints allows DiscASP to perform query-driven execution.

Input: At present, the DiscASP algorithm takes only pure grounded propositional ASP programs as input. A valid rule should be in the form of *head* :- *body*., :- *body*., or *head*. For example, if we want to represent 3 balls, the form $ball(1..3)$ is invalid. Instead, we have to declare them separately as $ball(1)$. $ball(2)$. $ball(3)$. The input ASP program will be converted and saved into a directed-graph data structure representing the CNR dependency graph. The conversion process is based on the concepts that were introduced in Section 2. When given a query `Q`, the additional constraint `:- not Q` will be appended to the original ASP program.

The DiscASP Algorithm: The DiscASP algorithm is a recursive algorithm. Since a CNR dependency graph represents the causal relationships among nodes, a topological order would indicate the truth values flow along edges from one node to another, starting from the leaves. By their nature, the constraint nodes (labeled *False*, discussed in Section 2) will be at the end of such flows in the CNR dependency graph. Therefore, we can incrementally establish the satisfiability relationships across all the nodes starting from the constraint nodes. This incremental establishment of satisfiability starting from the constraint nodes amounts to developing a proof tree.

```
%% program 5
m :- p.        m :- not q.        m :- r.        :- not m.        :- n.
```

An example (Program 5) is shown in Figure 5 where the graph is to the left and the proof tree to the right. To falsify the constraint node, i.e., to ensure it is *False*, node *m* must be *True* and *n* must be *False*. For node *m* to be *True*, at least one of the three must hold: *p* is *True*, *q* is *False*, or *r* is *True*. When every node's presumed truth value has been found to be consistent with all the dependencies, the algorithm will return the answers. Algorithm 1 shows an abstraction of the core algorithm. Here we explain some key concepts as follows:

Effective Edge: An effective edge in a CNR dependency graph refers to any edge that propagates *True* value to the node it is incident on. There are two types of *effective* edges: (i) positive edge emanating from a *True* node; (ii) negative edge emanating from a *False* node. An effective edge only points to a *True* node.

Satisfying Conjunction vs. Disjunction: There are two kinds of dependencies that may arise for a node in the CNR dependency graph: conjunctive and disjunctive. A conjunctive dependency refers to the situation where a node is presumed to be *False*. In this case, none of the edges incident into the node should be effective edges. A disjunctive dependency indicates that when a node is presumed to be *True*, at least one of the in-coming edges should be effective. Since DiscASP works in a reverse manner (from constraints to facts), we may get multiple partial models before we can validate the *True*/*False* label of

Algorithm 1 DiscASP core algorithm (abstraction)

```
 1: procedure REASONINGREC(state, node, value)
 2:     if (value == false) then                    ▷ case 1: presume the current node to be false
 3:         if the current node's presumed value can be proved then
 4:             add the current node to facts according to its proved value
 5:             propagate truth values
 6:             return facts
 7:         else
 8:             for each predecessor do
 9:                 assign truth value which doesn't make the current node true
10:                 recursively find the sub-answer set accordingly
11:             end for
12:             run conjunctive merge
13:             return the results
14:         end if
15:     else                                        ▷ case 2: presume the current node to be true
16:         if the current node's presumed value can be proved then
17:             add the current node to facts according to its proved value
18:             propagate truth values
19:             return facts
20:         else
21:             for each predecessor do
22:                 assign truth value which doesn't make the current node false
23:                 recursively find the sub-answer set accordingly
24:             end for
25:             run disjunctive merge
26:             return the results
27:         end if
28:     end if
29: end procedure
```

the current node. For both conjunction and disjunction, these partial models need to be merged for the sake of integrity as well as efficiency. The merging process is discussed later.

Proof Branch: In the DiscASP algorithm, we start from the constraints that have to be shown to be false and incrementally construct a proof tree obeying the constraints imposed by the CNR dependency graph. In this incremental reasoning process, we will pursue various paths in the CNR dependency graph. Our proof will have multiple branches, corresponding to various paths in the CNR dependency graph. Traversal of a branch stops when we reach a fact node whose value has already been given by the ASP program (i.e., known to be true due to being a fact, or known to be false because the atom does not have a rule with that atom as head), or sense that the branch contains a **loop** (discussed next).

Loop Detection and Handling: In an ASP program, loops among literals may exist. There are three kinds of loops that can be found in the program: even loops, odd loops, and positive loops. Even loops and odd loops refer to loops that have an even or odd number of negative edges in the corresponding dependency graph. Positive loops are loops with no negative edge. It is well known that even loops generate multiple worlds, while odd loops kill worlds. For example, in program `p :- not q.  q :- not p.`, *p* and *q* form an even loop, which generates two mutually exclusive worlds: {p/True, q/False} and {q/True, p/False}. For program `p :- not q.  q :- not r.  r :- not p.`, nodes *p*, *q* and *r* form an odd loop, which makes the program unsatisfiable. Odd loops are currently disallowed in DiscASP, as constraints are expressed through the headless constraint construct.

For positive loops, we need to ensure that any models computed are consistent with ASP semantics. Suppose we have a program `p :- q.  q :- p.`, under ASP semantics, it will have only one answer set: {p/False, q/False}. The other model ({p/True, q/True}) has to be rejected, as it is not well-founded per ASP semantics. Thus, positive loops have to be handled properly so that only correct answer sets are reported. To detect a loop, DiscASP keeps track of the presumed nodes along the branch, when the current node has been seen previously, we will check whether there exists any *False* node between these two nodes. If so, it is an even loop, otherwise, it is a positive loop and only the falsifying assignment should be computed.

Model Merging: As mentioned previously, for each presumed node *n* (i.e., a node assigned a truth value), its dependencies will be either conjunctive (if *n* is presumed *False*) or disjunctive. (if *n* is presumed *True*). For both conditions, we need to merge the partial models that have been computed so far while assigning a truth value to the dependent nodes. For the conjunctive condition, the merging process only takes successfully merged models, each of which is the union of two non-conflicting sub-models. For example, consider a node *n* that is presumed to be *False*. Suppose it has two predecessors *p* and *q*, both *p* and *q* connect to *n* via negative edges. So *n* will only be *False* when both *p* and *q* will be *True*. We need a conjunctive merge here. Suppose we have sub-models {p_1:{a/True, d/True, b/False}, p_2:{a/False, b/True}} that hold for *p* to be *True*, and sub-models {q1:{a/True, c/True, b/False}} for *q* to be *True*. The conjunctive merging of sub-models between *p* and *q* will only accept the union of p_1 and q_1, because p_2 conflicts with q_1. Therefore, there will only be one model to satisfy for *n* being *False*, that is {a/True, c/True, d/True, b/False}.

For disjunctive merging, we will keep the conflicted sub-models along with successfully merged ones. Let's modify the above example a little bit by presuming the value of node *n* to be *True*, and keep everything else unchanged. Now the merging condition becomes disjunctive because one of *p* or *q* being *False* will still make *n True*. Since p_1 and q_1 can be merged without conflict, we replace them with their union {a/True, c/True, d/True, b/False}. But this time we don't discard p_2, because p_2 is also a valid model that makes *n True*. Therefore, after this merging, we will have two sub-models for *n* being *True*: {{a/True, c/True, d/True, b/False}, {a/False, b/True}}.

Forward Propagation: Since nodes are assigned values is in a backward chaining manner, where we compute the truth assignment of the predecessors before that of the current node, the sub-models need to cover as much information as possible. If some nodes' values can be inferred from the proven nodes, they must also be added to the sub-model. For example, suppose we have a sub-model {a/True, b/False} for making node *n True*. Suppose there are two additional rules related to nodes *a* and *b*: `(i) c :- a. (ii) d :- not b.` In this case, we know that *c* and *d* must also be *True*.

DiscASP propagates truth values every time a presumed node value has been established, by using a causal map that covers all of the causal relationships for each node/value. When a presumed node/value is established, DiscASP will check whether there is any other node whose value can be inferred from current node assignments. If there are any, the inferred value is assigned to that node and propagation continues until the model does not change.

Query Handling: A query w.r.t. an ASP program amounts to checking whether a literal is in one of the models of the program. For instance, ASP program `p :- not q.  q :- not p.  :- p, q.` has two models {{p/True, q/False}, {p/False, q/True}}. If we query *p*, we should get the model {*p*}.

For query handling, DiscASP negates the query literal and appends it to the ASP program as an additional constraint. So for the above example, the query $p/True$ will be converted to a constraint rule

`:- not p.` and added to the original program. So the program will now be `p :- not q. q :- not p. :- p, q. :- not p.`

DiscASP begins its reasoning from a constraint node (typically, the query represented as a constraint), then searches for a partial answer set to satisfy the constraint.

Theorem: Let *P* be a program, and *A* be an answer generated by DiscASP for *P*. Assuming *P* is consistent, *A* is a subset of some stable model of *P*.
Proof Sketch: All constraint (headless rules) bodies are connected via incoming edges to a special constraint node. If the body of any constraint rule is *True*, then the constraint will be *True*. We want the constraint to be *False*, and so DiscASP begins constructing answers by trying to prove that the constraint is *False*. We can show by induction on the depth of the proof tree that if an answer is produced it is a subset of some stable model. (A detailed proof can be found in http://utdallas.edu/~gupta/discasp.pdf).

4 Application of DiscASP to Conversational Socialbots

In some application scenarios, finding the whole answer sets may be overkill or impractical. Especially for commonsense reasoning, where the knowledgebases may be large and guaranteeing consistency may be hard as parts of the knowledgebase were constructed separately. If a subset of the knoweldgebase that contains the answer we are seeking is consistent, we may not care about other inconsistent part of the knowledgebase.

Relevant Consistent Concepts: When we hold a social conversation, such as when we meet another person at a social event, the conversation will start from a topic such as one person asking the other "Have you seen any movies lately?" Once the second person's response is known (e.g., the person answers Titanic), a reasonable conversation strategy is to engage the other person in a conversation around this topic. Based on our commonsense knowledge, we will immediately attempt to recall all the information we know about the movie Titanic that is consistent with and relevant to our knowledge that the other persons like it. This information may include the movie's actors, the director, trivia about the movie, awards the movie earned, the movie plot, etc. This information is recalled based on our commonsense knowledge that a person interested in a movie is also interested in its actors, director, related trivia, movie plot, awards it won, etc. and so recalling the information will help us prepare a response. We will then pick one of the aspects and respond, in order to continue the conversation. For example, we may say, that "Oh, yes, Leonardo Di Caprio, did a great job as the lead actor." The conversation may revolve around Titanic, but may change to another movie that Leonardo Di Caprio has also acted in, such as Wolf of Wall Street. Conversations will continue to evolve from the topic of Titanic movie in this manner, until the topic changes completely, say, to pet ownership. Conversations will then ensue on this topic in a manner similar to the topic of movies.

For a socialbot to hold such casual conversations with a human, it needs to figure out the knowledge related to a topic (represented as a propositional atom, e.g., like(john, titanic)). We term the conceptual knowledge related to a topic τ as *relevant consistent concepts* (RCC) of τ. These concepts are inferred by humans from commonsense rules that they learn over time, for example, we might learn that *normally, a person who likes a movie, will also like the actors in the movie*. Thus, if we know that John likes Titanic, we infer that very likely, he likes Leonardo Di Caprio, the lead actor, too.

Why A Partial Answer Set Solver?: Finding the complete answer set of a program requires considering the entire dependency graph, which may include sub-graphs that are disconnected from each other. For example, knowledgebase about books will be largely independent of the knowledgebase about movies.

A traditional answer set solver will generate answer sets where a truth assignment is generated for every atom. However, for applications such as conversational socialbot, given a topic τ, we only want to generate the set of *relevant consistent concepts* of τ. Since the knowledge base contains more knowledge than necessary related to a topic, for example, when we hold a conversation about movies, finding the whole answer set is overkill. In our work, we use DiscASP (for **Disc**ussions with **ASP**) for computing a partial answer set representing the `RCC(`τ`)`, given a topic atom τ.

In DiscASP, if a dependency sub-graph is disconnected to a query related to a topic τ, then for obvious reasons, the atoms in this sub-graph do not have to be considered at all to compute the `RCC(`τ`)` set. Given a query for a topic τ, we only need to verify its validity and find out its related consistent concepts. As discussed (Section 3), DiscASP guarantees the following property: given an answer set program `P`, if `P` has an answer set, then any partial answer set computed by DiscASP will be a subset of a complete answer set of `P`.

DiscASP is designed as a system that finds partial stable models for a given topic with respect to a specific knowledgebase coded in ASP. The knowledgebase will be encoded as propositional logic facts and will contain both generic data that is collected from data sources, e.g., iMDB movie database, user profiles, etc., as well as commonsense knowledge rules that capture the knowledge of the domain. These facts and rules are expressed in ASP. The commonsense rules are grounded using the GRINGO grounding system [10]. Since DiscASP is a graph based ASP solver, we need to transform the grounded program into a CNR dependency graph as discussed earlier. While building this graph, the topic query, `Q`, will also be added as the constraint (`:- not Q.`). Then the dependency graph will be passed to the partial answer set solver, which computes partial stable models. The DiscASP algorithm is quite simple: it starts with the constraint introduced by the query (`:- not Q.`) and finds the smallest model, `M`, that ensures that the constraint holds. If there are atoms in `M` that also involve other constraints, then the DiscASP algorithm ensures that these constraints are also satisfied. The partial model found is the relevant consistent concept (RCC) set of `Q`.

DiscASP Illustration: Let's use a simple propositional ASP programs to demonstrate how DiscASP works. Consider *Program 6* with query `?- p`, the full answer sets are {`r, q, p, s, t, j, m, k, n, o, w, a`} and {`r, q, p, s, t, j, m, k, n, o, w, b`}. There are two answer sets because nodes `a, b` constitute an even loop, which creates two mutually exclusive worlds.

```
%% program 6
p :- q, r.  q :- s, not x.  t :- s.  j :- r.  m :- t.  k :- j.  n :- p.  o :- n.
r :- not u, not v.  w :- not v.  a :- not b.  b :- not a.  s.
```

DiscASP returns one single partial answer set {`r, q, p, s, t, j, m, k, n, o, w, not x, not u, not v`}. Because nodes `a, b` have no connection to the sub-graph that contains the query node `p`, DiscASP only returns the most general answer containing the query node. If our query is extended to `?- p, a`, then `a, not b` will be added to this answer set.

Narrowing the RCC: The DiscASP algorithm provides the facility to narrow the RCC set even further. This is because even a partial answer set may still contain too many atoms. Suppose our socialbot is chatting with a user John about movies and comes to know that the user likes the movie *Titanic*. DiscASP will take `like_movie(john, titanic)` as a query, compute a partial stable model from the knowledge base consisting of the iMDB movie database. The answer set may contain atoms that are at quite some distance to the query in the CNR dependency graph. DiscASP thus also takes a radius parameter (a positive number), `r`, and finds all atoms in the RCC that are at a distance `r` or less from the query constraint in the CNR dependency graph. The DiscASP algorithm will also output the path between the query constraint and an arbitrary atom in the RCC. The path describes the connection between the two atoms and can be used to construct the socialbot's natural language response.

Definition 2 *The set of relevant consistent concepts for a given topic τ and radius r, denoted* `RCC(`τ`,`r`)`*, is a subset of the partial answer set containing τ, where for each atom a in this partial answer set, the distance between a and the atom τ in the CNR dependency-graph is less than or equal to r.*

For example, from the atom `like_movie(john, titanic)`, we know that John may also like to talk about Matt Damon because he and Leonardo Di Caprio, the hero of Titanic, were together in the movie *The Departed*. Then John may also like to talk about Tom Hanks who was a co-actor with Matt Damon in the movie *Saving Private Ryan*. Here if we set the propagation radius to 2, DiscASP will only give us `like_actor(john, matt_damon)`, omitting Tom Hanks. Then, our socialbot will start talking about Matt Damon. If we switch from *Titanic* directly to Tom Hanks, the user may feel confused. Thus, it is important to keep the radius small to keep the topics relevant. For program 6 above, if we set the radius to 2, for the query `q`, we will compute the partial answer set {`r, q, p, s, t, j, not x`}.

An Example: Socialbot Conversation about Movies: We have chosen the real-world example of the movie domain to demonstrate our DiscASP algorithm. First, we need to define the ASP rule-set that captures our commonsense knowledge for holding conversations about movies. These rules are written in a way that portrays the human thinking process. In other words, these rules mimic the reasoning process of how a human will choose what to talk next about a particular movie in a conversation. For example, humans will assume that *normally, if someone likes an actor, he/she likes the movies by the actor and vice-versa*. This condition can be depicted using the following two default rules with exceptions, where person *P* likes a movie *M* or an actor *A*. The rules are self-explanatory, so to save space, details are omitted.

```
like_movie(P, M) :- movie(M), actor(A), like_actor(P, A), movie_actor(M, A),
        not ab_like_movie(P, M).
like_actor(P, A) :- movie(M), actor(A), like_movie(P, M), movie_actor(M, A),
        not ab_like_actor(P, A).
```

We use the `talk_preference/3` predicate to capture the attribute A, that the socialbot can talk about next, regarding a movie M, given a user P. For example, *people prefer to talk about an actor of a movie if he/she is the main actor, a famous actor or an Oscar winning actor*. Also, *normally, people like to talk about a movie's awards, awards to actors in the movie, or trivia about a movie*. Note that these default rules include exceptions, for example, to capture the situation that whenever the socialbot is done talking about an attribute it adds it to its exception list so as to not talk about it again in the same conversation. The `ab_talk_preference/3` predicate models these exceptional situations (in the code below, the user John has already talked about the movie Avatar).

```
talk_preference(P, M, A) :- actor(A), movie(M), like_movie(P, M),
        main_actor(A, M), not ab_talk_preference(P, M, A).
talk_preference(P, M, A) :- actor(A),like_movie(P, M),movie_actor(M, A),
        famous_actor(A), not ab_talk_preference(P, M, A).
talk_preference(P, M, A) :- actor(A),like_movie(P, M),movie_actor(M, A),
        award_won(A,oscar), not ab_talk_preference(P,M,A).
talk_preference(P, M, awards) :- movie(M), like_movie(P, M),
        not ab_talk_preference(P, M, awards).
talk_preference(P, M, actor_award) :- movie(M), movie_actor(M, A),
        like_actor(P, A), not ab_talk_preference(P, M, actor_award).
talk_preference(P, M, trivia) :- movie(M), like_movie(P, M),
        not ab_talk_preference(P, M, trivia).
```

```
ab_talk_preference(P, M, A) :- already_talked(P,M, A).
already_talked(john, avatar, actor).
```

Also, we will have default rules that model knowledge about people's preferences such as: *Normally, young people like action and sci-fi movies*, or *children like kid's movies*, or *women like romance and drama movies*. One example rule is given below.

```
like_movie(P, M) :- movie(M), age_category(P, young), genre(M, scifi),
        not ab_like_movie(P, M).
```

Human reasoning process utilizes *constraints* on a regular basis. In this movie domain, we can have constraints that say - *anyone who is a child should not be involved with discussions about R-rated movies* and this can be written as follows:

```
:- like_movie(P, M), is_adult_movie(M), age_category(P, children).
```

Similar to *constraints*, *preferences over defaults* are also an integral part of human reasoning process. An example rule will be: *normally, people are more interested in talking about the actor than the director of a movie, however, people are more interested in talking about the director if he/she won an Oscar.*

```
movie_cast_crew_type(actor).        movie_cast_crew_type(director).
like_director(P, D) :- movie(M), director(D), like_movie(P, M), movie_director
        (M, D), not ab_like_director(P, A).
talk_preference(P, M, actor) :- actor(A), like_actor(P, A),
        not ab_talk_preference(P, M, actor), not neg_talk_preference(P,M, actor).
talk_preference(P, M, director) :- director(D), like_director(P, D),
        not ab_talk_preference(P, M, director),
        not neg_talk_preference(P, M, director).
neg_talk_preference(P, M, X1) :- movie_cast_crew_type(X1),
        movie_cast_crew_type(X2), talk_preference(P, M, X2), X1 \= X2.
neg_talk_preference(P, M, director) :- actor(A), like_actor(P, A),
        not ab_neg_talk_preference(P, M, director).
ab_neg_talk_preference(P, M, director) :- director(D), like_director(P, D),
        director_award(D, oscar).
ab_talk_preference(P, M, actor) :- director(D), like_director(P, D),
        director_award(D, oscar).
```

The above rules represent part of the knowledge that a socialbot must have to hold a conversation with a person (in addition to knowledge about movies from a movie-database such as iMDB). The socialbot will then compute the RCC(τ, r) set given a topic τ that interests the user (like_movie(john,titanic)) for example, then select one of the talk_preference/3 atoms to figure out what to say next. Given the topic like_movie(john, titanic), the above rules with the iMDB database (restricted to data for 55 movies) computed the following RCC with a distance of 3 (we only show the talk_preference/3 predicates as those are most relevant and used for advancing the conversation:

{talk_preference(john,titanic,trivia), talk_preference(john,titanic,awards),
talk_preference(john,titanic,leonardo_dicaprio)}

From this RCC set, we may pick talk_preference(john, leonardo_di_caprio). Our algorithm will give us the "path" in the dependency graph from like_movie(john,titanic) that leads to this inference. We will use the path to craft the response: "Leonardo Di Caprio was the actor in the movie and did a great job." We may have also picked talk_preference(john, wolf_of_wall_street, leonardo_di_caprio) and the path computed will tell us that the connection is through the lead actor De Caprio. So the socialbot will craft the response: "Leonardo Di Caprio did a great job in Titanic. He also acted in the movie Wolf of Wall Street. Did you see that movie?". Phrases such a "did a great job" and “Did you see that movie?” are based on having further commonsense knowledge that associates such phrases with talking points (`talk_preference/3`) about an actor or a new movie mentioned.

Note that if we pick, for example, `talk_preference(john,leonardo_di_caprio)` then the fact `already_discussed(john, titanic,leonardo_di_caprio)` will be added to our database. When the RCC is computed for the next utterance, we will not pick the topic `talk_preference(john, titanic, leonardo_di_caprio)` for advancing the conversation.

Table 1: Performance Table for DiscASP

Query	Distance	#Result	Execution Time (ms)
like_movie(john,titanic)	3	3	9.90
like_movie(john,titanic)	5	8	11.01
like_movie(john,forrest_gump)	3	2	4.02
like_movie(john,forrest_gump)	5	2	7.61
like_actor(john,tom_hanks)	3	2	5.84
like_actor(john,tom_hanks)	5	6	5.04

Performance: We use the rules from the movie example in Section 4 with a data set of 55 movies from iMDB movie database for performance testing of DiscASP system. As shown in Table 1, the queries are passed into the system, and each query has two different RCC distances (3 and 5), the outputs of the system are shown as the number of `talk_preference/2` atoms computed. The knowledge base has 740 grounded rules per user. As can be observed, query execution is quite fast. This is important in a conversational socialbot as the latency between the user utterance and the bot response cannot be too much (at most a few hundred milliseconds).

5 Related Work

The CNR Dependency Graph and our DiscASP algorithm have a number of advantages: (i) partial answer set can be incrementally computed to find the RCC set; the chain of reasoning that connects one concept atom in the answer set to another can be found with no additional costs; this is in contrast to other methods such as CLINGO, where the *entire* answer set has to be computed, and Galliwasp, where atoms related to proving a query only can be found. (ii) Declarative Knowledge: The DiscASP algorithm allows knowledge in a conversational socialbot to be represented declaratively as an answer set program, considerably reducing the complexity of the project. (iii) A better representation for Knowledge Graph: The CNR dependency graph provides a better representation for a knowledge graph, as it can account for exceptions and preferences as well. Traditional knowledge graphs such as ConceptNet [17] and Microsoft Knowledge Graph [23] represent knowledge by connecting words through relations and providing an ontology; they cannot represent exceptions and preferences. Thus, knowledge is inconsistent and wrong conclusions (such as “penguins can fly”) can be drawn.

Logic programming has been used in the past for developing chatbots. Tarau and Figa [9] represent domain knowledge with propositional clauses in Prolog and find answers via using goal driven queries. Also, Tarau et al. [21] explored the use of dependency graphs with lexical knowledge and TextRank algorithm to give the most relevant answers given a query. In recent times, most chatbots are developed using machine-learning technology [14]. Researchers have developed many dialog datasets to train and test the language models, such as SwitchBoard corpus [12], bAbI dialog dataset [5], etc.

These systems that are built with deep learning techniques learn the patterns of the training text remarkably well and show promising results on test data. With the recent advancements in the language model research, the pre-trained models such as BERT [8] and GPT-3 [6] show outstanding capability in generating natural language responses. Using these language models people have also developed techniques to build conversational chatbots, such as PD-NRG (policy driven neural response generator) [13]. While these tools and techniques based on machine learning and deep learning are very impressive, they are based on syntactic and contextual pattern matching. They produce absolutely no understanding of the knowledge contained in the text. They utterly fail when any type of reasoning is required. Their black-box nature also precludes them from explaining their responses.

There are also systems, such as StaCACK [3], based on ASP that not only outperform the neural-models in accuracy on the bAbI-dialog dataset but also show natural language justification for each of its answers. Our conversational socialbot application using DiscASP is another step forward toward mimicking the human reasoning process in intelligent systems. We believe that, to obtain truly intelligent behavior, machine learning and commonsense reasoning should work in tandem.

6 Conclusion and Future Work

We proposed a dependency graph based algorithm called DiscASP to compute the atoms of an answer set that are related to a query atom within a fixed "causal distance". We used a novel transformation to ensure that each program has a unique dependency graph, as otherwise multiple programs can have the same dependency graph. A major advantage of our algorithm is that it can produce a justification for any proposition that is entailed by a program. Currently, DiscASP only works for propositional answer set programs. Our goal is to extend it so that answer sets of datalog programs can also be computed without having to ground them first. This will be achieved by dynamically propagating bindings along the edges connecting the nodes in our algorithm's propagation phase. Thus, DiscASP will help us pave the way for finding answer sets efficiently without grounding the program first. The DiscASP algorithm is quite efficient and has been applied to practical problems such as developing conversational socialbots.

We described a real-world application of DiscASP to developing conversational socialbots. We showed how commonsense knowledge about holding a conversation about movies can be coded as an answer set program. We showed how atoms that are relevant to advancing a conversation given a topic can be found using the DiscASP algorithm. Future work includes incorporating knowledge about other social topics such as books, movies, sports, family, pets, etc., to realize a full-fledged socialbot using ASP technology. Current technology for developing socialbots is primarily based on machine learning. An effective socialbot can only be built if the socialbot "understands" the utterances of the human user and can reason about them. Thus, knowledge-based approaches such as those based on ASP are crucial for developing conversational socialbots.

Acknowledgment: Authors are supported by grants from NSF (IIS 1718945, IIS 1910131, IIP 1916206), Amazon and DoD.

References

[1] Amazon (2021): *Amazon Alexa Socialbot Challenge 4.0*. Available at https://utdallas.edu/~gupta/alexa.html.

[2] Christian Anger, Kathrin Konczak & Thomas Linke (2001): *NoMoRe: A System for Non-Monotonie Reasoning under Answer Set Semantics*. *PSC 802 BOX 14 FPO 09499-0014*, p. 406, doi:10.1007/3-540-45744-5_24.

[3] Kinjal Basu, Sarat Varanasi, Farhad Shakerin, Joaquin Arias & Gopal Gupta (2021): *Knowledge-driven Natural Language Understanding of English Text and its Applications*. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 12554–12563.

[4] Kinjal Basu, Huaduo Wang, Nancy Dominguez, Xiangci Li, Fang Li, Sarat Varanasi & Gopal Gupta (2021): *CASPR: A Commonsense Reasoning-based Conversational Socialbot*. Forthcoming.

[5] Antoine Bordes & Jason Weston (2016): *Learning End-to-End Goal-Oriented Dialog*. *CoRR* abs/1605.07683. Available at http://arxiv.org/abs/1605.07683.

[6] Tom B. Brown, Benjamin Mann et al. (2020): *Language models are few-shot learners*. *arXiv preprint arXiv:2005.14165*. Available at https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[7] Kenneth Mark Colby, Sylvia Weber & Franklin Dennis Hilf (1971): *Artificial paranoia*. *Artificial Intelligence* 2(1), pp. 1–25, doi:10.1016/0004-3702(71)90002-6.

[8] Jacob Devlin, Ming-Wei Chang et al. (2018): *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. *CoRR* abs/1810.04805. Available at http://arxiv.org/abs/1810.04805.

[9] Elizabeth Figa & Paul Tarau (2004): *Knowledge Assimilation and Web Deployment Techniques for Conversational Agents*. In: *Proceedings of the AAMAS 2004 Workshop On Embodied Conversational Agents: Balanced Perception and Action*, Citeseer, pp. 108–114.

[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2014): *Clingo = ASP + Control: Preliminary Report*. *CoRR* abs/1405.3694.

[11] Michael Gelfond & Yulia Kahl (2014): *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press, doi:10.1017/CBO9781139342124.

[12] John J Godfrey et al. (1992): *SWITCHBOARD: Telephone speech corpus for research and development*. In: *Proc. IEEE Conf. on Acoustics, Speech, and Signal Processing*, pp. 517–520, doi:10.1109/ICASSP.1992.225858.

[13] Behnam Hedayatnia et al. (2020): *Policy-Driven Neural Response Generation for Knowledge-Grounded Dialogue Systems*. *CoRR* abs/2005.12529. Available at https://arxiv.org/abs/2005.12529.

[14] Daniel Jurafsky & James H. Martin (2008): *Speech and Language Processing (2nd Edition)*. Prentice Hall.

[15] Kathrin Konczak, Thomas Linke & Torsten Schaub (2006): *Graphs and colorings for answer set programming*. *Theory Pract. Log. Program.* 6(1-2), pp. 61–106, doi:10.1017/S1471068405002528.

[16] Thomas Linke & Vladimir Sarsakov (2005): *Suitable graphs for answer set programming*. In: *Proc. LPAR*, Springer, pp. 154–168, doi:10.1007/978-3-540-24609-1_26.

[17] H. Liu & P. Singh (2004): *ConceptNet — A Practical Commonsense Reasoning Tool-Kit*. *BT Technology Journal* 22(4), pp. 211–226, doi:10.1023/B:BTTJ.0000047600.45421.6d.

[18] Victor W Marek & Miroslaw Truszczyński (1999): *Stable models and an alternative logic programming paradigm*. In: *The Logic Programming Paradigm*, Springer, pp. 375–398, doi:10.1006/jcss.1995.1053.

[19] Kyle Marple, Ajay Bansal, Richard Min & Gopal Gupta (2012): *Goal-directed execution of answer set programs*. In: *Proc. PPDP'12*, ACM, pp. 35–44, doi:10.1145/2370776.2370782.

[20] Ashwin Ram, Rohit Prasad, Chandra Khatri et al. (2018): *Conversational AI: The Science Behind the Alexa Prize*. *CoRR* abs/1801.03604. Available at http://arxiv.org/abs/1801.03604.

[21] Paul Tarau & Eduardo Blanco (2021): *Interactive Text Graph Mining with a Prolog-Based Dialog Engine*. *Theory and Practice of Logic Programming* 21(2), p. 244–263, doi:10.14569/IJACSA.2017.081052.

[22] Joseph Weizenbaum (1966): *ELIZA—a computer program for the study of natural language communication between man and machine*. *CACM* 9(1), pp. 36–45, doi:10.1145/367593.367617.

[23] Zheng Yu, Haixun Wang, Xuemin Lin & Min Wang (2016): *Understanding Short Texts Through Semantic Enrichment and Hashing*. *IEEE TKDE* 28(2), pp. 566–579, doi:10.1109/TKDE.2015.2485224.