Handout on Basics behind Answer Sets

Yuliya Lierler
University of Nebraska Omaha

Introduction

Answer set programming (ASP) is a form of declarative programming oriented towards modeling

i intelligent agents in knowledge representation and reasoning and

ii difficult combinatorial search problems.

It belongs to the group of so called constraint programming languages. ASP has been applied to a variety of applications including plan generation and product configuration problems in artificial intelligence and graph-theoretic problems arising in VLSI design and in historical linguistics [1].

Syntactically, ASP programs look like logic programs in Prolog, but the computational mechanisms used in ASP are different: they are based on the ideas stemming from the development of satisfiability solvers for propositional logic.

This document presents the concept of an answer set for programs in their simplest form: *no variables, no classical negation symbol, no disjunction in the heads of rules.* The textbooks [**?**, 2] provide an account for general definition of this concept that assumes rules of the general form that includes all the features mentioned above. Yet, it is useful to first consider as simple programs as possible to build intuitions about answer sets.

In this note italics is primarily used to identify concepts that are being defined. Some definitions are identified by the word **Definition**.

Traditional Programs and their Answer Sets

1 Syntax

A *traditional rule* is an expression of the form

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n. \tag{1}$$

where $n \geq m \geq 0$; a_0 is a propositional atom or symbol $\bot$; and $a_1, \ldots, a_n$ are propositional atoms (propositional symbols). The expression a_0 is called the *head* of the rule, and the list

$$a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n$$

is its *body*. If the body is empty ($n = 0$) then the rule is called a *fact* and identified with its head a_0. We call rule (1) a *constraint* when its head is symbol $\bot$ (we then drop $\bot$ from a rule).

A *traditional (or propositional logic) program* is a finite set of traditional rules. For instance,

$$\begin{array}{l} p. \\ r \leftarrow p, q. \end{array} \tag{2}$$

and

$$\begin{array}{l} p \leftarrow \mathit{not}\ q. \\ q \leftarrow \mathit{not}\ r. \end{array} \tag{3}$$

are traditional programs.

A traditional rule (1) is *positive* if $m = n$, that is to say, if it has the form

$$a_0 \leftarrow a_1, \ldots, a_m. \tag{4}$$

A traditional program is *positive* if each of its rules is positive. For instance, program (2) is positive, and (3) is not.

2 The Answer Set of a Positive Program

We will first define the concept of an answer set for positive traditional programs. To begin, we introduce auxiliary definitions.

Definition 1. *A set X of atoms* satisfies *a positive traditional rule* (4) *when* $a_0 \in X$ *whenever* $\{a_1, \ldots, a_m\} \subseteq X$.

For instance, any positive traditional rule (4) is satisfied by a singleton set $\{a_0\}$.

To interpret Definition 1 recall the truth table of implication in propositional logic:

p	q	$p \rightarrow q$
$true$	$true$	$true$
$true$	$false$	$false$
$false$	$true$	$true$
$false$	$false$	$true$

One can intuitively read it in English as follows condition $p \rightarrow q$ holds if q holds whenever p holds. Expression $a_0 \in X$ plays a role of q whereas $\{a_1, \ldots, a_m\} \subseteq X$ plays a role of p in the definition of a set of atoms satisfying a rule.

Problem 1. *Given a set X of atoms and a positive traditional rule* (4)

	Does X satisfies rule (4)*?*
$\{a_1, \ldots, a_m\} \subseteq X$ *and* $a_0 \in X$	Yes
$\{a_1, \ldots, a_m\} \subseteq X$ *and* $a_0 \notin X$	
$\{a_1, \ldots, a_m\} \not\subseteq X$ *and* $a_0 \in X$	Yes
$\{a_1, \ldots, a_m\} \not\subseteq X$ *and* $a_0 \notin X$	

Definition 2. *A set X of atoms* satisfies *a positive traditional program* Π *if X satisfies every rule (4) in* Π*.*

For instance, any positive traditional program is satisfied by the set composed of the heads a_0 of all its rules (4).

Problem 2.

X	*Does X satisfies program* (2)*?*
$\emptyset$	No
$\{p\}$	Yes
$\{q\}$	
$\{r\}$	
$\{p\ q\}$	
$\{p\ r\}$	
$\{q\ r\}$	
$\{p\ q\ r\}$	

Proposition 1. *For any positive traditional program* Π *without constraints, there exists a set of atoms satisfying* Π*.*

Proof. Indeed. Consider the set X to be composed of all atoms occurring in Π. It follows that for every rule in Π its head atom is in X. By the definition of rule satisfaction it follows that X satisfies every rule of Π. (Note that we could have also considered other sets to construct a similar argument. Can you think of such a set?) □

Proposition 2. *For any positive traditional program* Π*, the intersection of all sets satisfying* Π *satisfies* Π *also.*

Proof. By contradiction. Suppose that this is not the case. Let X denote the intersection of all sets satisfying Π. By definition (of program's satisfiability), there exists a rule

$$a_0 \leftarrow a_1, \ldots, a_m.$$

in Π such that it is not satisfied by X, in other words

$$a_0 \notin X$$

and

$$\{a_1, \ldots, a_m\} \subseteq X.$$

Since X is an intersection of all sets satisfying Π then we conclude that (i) $\{a_1, \ldots, a_m\}$ belongs to each one of the sets satisfying Π and (ii) there is a set Y satisfying Π such that $a_0 \notin Y$. By definition, Y does not satisfy Π. We derive at contradiction.

□

Proposition 2 allows us to talk about the *smallest* set of atoms that satisfies Π.

Definition 3. *The smallest set of atoms that satisfies positive traditional program* Π *is called* the answer set *of* Π.

For instance, the sets of atoms satisfying program (2) are

$$\{p\},\ \{p, r\},\ \{p, q, r\},$$

and its answer set is $\{p\}$.

We now illustrate some interesting properties of answer sets. Let a program consist of facts only. The set of these facts form the only answer set of such a program. Intuitively, each fact states what is known and an answer set reflects this information by asserting that each atom mentioned as a fact is *true*, whereas anything else is *false*. Thus answer sets semantics follows closed world assumption (CWA) – presumption that what is not currently known to be *true* is *false*. From the definition of an answer set and Proposition 1, it immediately follows that any positive traditional program has a unique answer set. It is intuitive to argue that answer set semantics for logic programs generalizes CWA. Note that an atom, which does not occur in the head of some rule in a program, will not be a part of any answer set of this program:

Proposition 3. *If* X *is an answer set of a positive traditional program* Π*, then every element of* X *is the head of one of the rules of* Π.

Positive Rules Intuitively, we can think of (4) when its head is an atom as a rule for generating atoms: once you have generated $a_1, \ldots, a_m$, you are allowed to generate a_0. The answer set is the set of all atoms that can be generated by applying rules of the program in any order. For instance, the first rule of (2) allows us to include p in the answer set. The second rule says that we can add r to the answer set if we have already included p and q. Given these two rules only, we can generate no atoms besides p. If we extend program (2) by adding the rule

$$q \leftarrow p.$$

then the answer set will become $\{p, q, r\}$. We can easily implement such a process for generating the answer set for positive traditional program by an efficient procedure.

Positive rules may remind you *Horn clauses* or *definite clauses*. One can identify (4) with the following implication

$$a_1 \wedge \cdots \wedge a_m \Rightarrow a_0$$

that is equivalent to the Horn clause

$$\neg a_1 \vee \cdots \vee \neg a_m \vee a_0.$$

Rule (4) is satisfied by a set of atoms if and only if its respective Horn clause is satisfied by this set in propositional logic.

3 Answer Sets of a Program with Negation

To extend the definition of an answer set to arbitrary traditional programs, we will introduce one more auxiliary definition.

Definition 4. *The* reduct Π^X *of a traditional program* Π *relative to a set* X *of atoms is the set of rules (4) for all rules (1) in* Π *such that* $a_{m+1}, \ldots, a_n \notin X$.

In other words, Π^X is constructed from Π by (i) dropping all rules (1) such that at least one atom from $a_{m+1}, \ldots, a_n$ is in X, and (ii) eliminating *not* $a_{m+1}, \ldots,$ *not* a_n expression from the rest of the rules.

Thus Π^X is a positive traditional program.

Problem 3. *Let* Π *be* (3)*,*

X	*What is* Π^X*?*	*Explanation*
$\emptyset$	$p.$ $q.$	$p \leftarrow$ ~~*not q.*~~ $q \leftarrow$ ~~*not r.*~~
$\{p\}$	$p.$ $q.$	$p \leftarrow$ ~~*not q.*~~ $q \leftarrow$ ~~*not r.*~~
$\{q\}$	$q.$	~~$p \leftarrow$ *not q.*~~ $q \leftarrow$ ~~*not r.*~~
$\{r\}$		
$\{p\ q\}$		
$\{p\ r\}$		
$\{q\ r\}$		
$\{p\ q\ r\}$		

Definition 5. *We say that* X *is an* answer set *of* Π *if* X *is the answer set of* Π^X *(that is, the smallest set of atoms satisfying* Π^X*).*

Problem 4.

X	*Is* X *an answer set of program* (3)*?*
$\emptyset$	No
$\{p\}$	No
$\{q\}$	Yes
$\{r\}$	
$\{p\ q\}$	
$\{p\ r\}$	
$\{q\ r\}$	
$\{p\ q\ r\}$	

If Π is positive then, for any X, $\Pi^X = \Pi$. It follows that the new definition of an answer set is a generalization of the definition from Section 2: for any positive traditional program Π, X is the smallest set of atoms satisfying Π^X iff X is the smallest set of atoms satisfying Π.

Intuitively, rule (1) allows us to generate a_0 as soon as we generated the atoms $a_1, \ldots, a_m$ *provided that none of the atoms* $a_{m+1}, \ldots, a_n$ *can be*

generated using the rules of the program. There is a vicious circle in this sentence: to decide whether a rule of Π can be used to generate a new atom, we need to know which atoms can be generated using the rules of Π. The definition of an answer set overcomes this difficulty by employing a "fixpoint construction." Take a set X that you suspect may be exactly the set of atoms that can be generated using the rules of Π. Under this assumption, Π has the same meaning as the positive program Π^X. Consider the answer set of Π^X, as defined in Section 2. If this set is exactly identical to the set X that you started with then X was a "good guess"; it is indeed an answer set of Π.

In Problem 4, to find all answer sets of program (3) we constructed its reduct for each subset of $\{p, q, r\}$ to establish whether these sets are answer sets of (3). The following general properties of answer sets of traditional programs allow us to sometime establish that a set is not an answer set in a trivial way by inspecting its elements rather than constructing the reduct of a given program.

Proposition 4. *If X is an answer set of a traditional program Π then every element of X is the head of one of the rules of Π.*

Proposition 5. *If X is an answer set for a traditional program Π then no proper subset of X can be an answer set of Π.*

In application to program (3), Proposition 4 tells us that its answer sets do not contain r, so that we only need to check

$$\emptyset, \{p\}, \{q\}, \text{ and } \{p, q\}.$$

Once we established that $\{q\}$ is an answer set, by Proposition 5

- $\emptyset$ cannot be an answer set because it is a proper subset of the answer set $\{q\}$, and
- $\{p, q\}$ cannot be an answer set because the answer set $\{q\}$ is its proper subset.

Set $\{p\}$ is not an answer set as set $\{q\}$ is the answer set of program's reduct wrt $\{p\}$. Consequently, $\{q\}$ is the only answer set of (3).

Program (3) has a unique answer set. On the other hand, the program

$$\begin{array}{l} p \leftarrow not\ q. \\ q \leftarrow not\ p. \end{array} \tag{5}$$

has two answer sets: $\{p\}$ and $\{q\}$. The one-rule program

$$r \leftarrow not\ r. \tag{6}$$

has no answer sets.

Problem 5. *Prove that if X is an answer set of a traditional program Π so that for some rule (1), it holds that $\{a_1, \ldots, a_m\} \subseteq X$ and $\{a_{m+1}, \ldots, a_n\} \cap X = \emptyset$, then $a_0 \in X$.*

Problem 6. *Find all answer sets of the following program, which extends (5) by two additional rules:*

$$p \leftarrow not\ q.$$
$$q \leftarrow not\ p.$$
$$r \leftarrow p.$$
$$r \leftarrow q.$$

Problem 7. *Find all answer sets of the following combination of programs (5) and (6):*

$$p \leftarrow not\ q.$$
$$q \leftarrow not\ p.$$
$$r \leftarrow not\ r.$$
$$r \leftarrow p.$$

Problem 8. *Prove the claim of Proposition 4*

Constraints Consider a constraint

$$\leftarrow p.$$

Extending program (2) by this rule will result in a program that has no answer sets. In other words, constraint $\leftarrow p$ eliminates the only answer of (2). It is convenient to view any constraint

$$\leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$$

as a clause (a disjunction of literals)

$$\neg a_1 \vee \cdots \vee \neg a_m \vee a_{m+1} \vee \cdots \vee a_n.$$

Then, we can state the general property about constraints: answer sets of a program satisfy the propositional logic formula composed of its constraints (here (i) the notion of satisfaction is as understood classically in

propositional logic and (ii) an answer set is associated with an interpretation in an intuitive manner). Furthermore, for a program Π and a set Γ of constraints the answer sets of $\Pi \cup \Gamma$ coincide with the answer sets of Π that satisfy Γ. Consequently, constraints can be seen as elements of classical logic in logic programs.

4 Classical Negation in Programs

The textbook [2] immediately introduces programs that are of more complex form even in propositional case. Indeed, it allows additional connective $\neg$ (classical negation). So that basic entities of a program are literals (a *literal* is either an atom a or an atom proceeded with classical negation $\neg a$) In turn, the concept of an answer set is defined over the consistent sets of literals whereas here we defined an answer set over the sets of atoms. Yet, it is easy to *simulate* classical negation in the simpler form of programs that we consider here.

Classical negation can always be eliminated from a program by means of auxiliary atoms and additional constraints. Indeed, given a program composed of rules of the form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \tag{7}$$

so that l_0 is a literal or $\bot$ and $l_1 \ldots l_n$ are literals, we can replace an occurrence of any literal of the form $\neg a$ with a fresh atom a' and add a constraint to this program in the form

$$\leftarrow a, a'.$$

The answer sets of this new program as defined in this handout are in one to one correspondence with the answer sets as defined in the textbook. In particular, by replacing atoms of the form a' by $\neg a$ we obtain the textbook answer sets.

Acknowledgments

Parts of these notes follow the lecture notes on *Answer Sets; and Methodology of Answer Set Programming; course Answer set programming: CS395T, Spring 2005*[1] by Vladimir Lifschitz. Zachary Hansen helped to correct some typos.

[1] `http://www.cs.utexas.edu/~vl/teaching/asp.html`

References

[1] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[2] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press, 2014.

Handout on Answer Set Programming Paradigm

Yuliya Lierler
University of Nebraska Omaha

Introduction

Answer set programming paradigm (ASP) is a form of declarative programming oriented towards difficult combinatorial search problems. It belongs to the group of so called constraint programming languages. The idea of ASP is to represent a given computational problem by a program whose answer sets correspond to solutions, and then use an answer set solver to generate answer sets for this program. In this note we discuss the methodology of answer set programming as well as the use of software systems for computing answer sets. First, a graph coloring problem is utilized to illustrate the use of answer set programming in practice. Then, solutions to Hamiltonian cycle and to n-queens problems are presented. Across the handout you are given problems to solve. This handout is self-contained: you are given all the definitions and links that are required in constructing solutions.

In the text italics is primarily used to identify concepts that are being defined. Some definitions are identified by the word **Definition**.

In this course we will use the answer set system CLINGO[1] that incorporates answer set solver CLASP[1] with its front-end grounder GRINGO[1] (user guide is available online at `https://sourceforge.net/projects/potassco/files/guide/2.0/guide-2.0.pdf/download`). You may access system CLINGO via web interface available at `https://potassco.org/clingo/run/` or download an executable for CLINGO version 5 from the url listed at footnote 1.

Answer set programming practitioners develop applications that rely on ASP languages, which allow variables. Yet, common ASP solvers, including CLASP (a subsystem of CLINGO) process propositional logic programs only. We now re-introduce such programs and restate the definition of an answer set in the form convenient for this part of the course.

[1] `https://potassco.org/clingo/`.

1 Propositional Logic Programs Extended with Choice Rules

Recall traditional rules from the lecture notes on *Basics behind Answer Sets*[2] of the form

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n, \tag{1}$$

where a_0 is a propositional atom or symbol $\bot$; $a_1, \ldots, a_p$ are propositional atoms. In practice of answer set programming, rules of more complex structure are common. Here we provide intuitions behind so called choice rules. Later in the notes we talk about rules that go beyond propositional atoms. We will also mention constructs called aggregates that are convenient in modeling problems requiring, for example, counting.

Answer sets semantics by default follows closed world assumption (CWA) – presumption that what is not currently known to be *true* is *false*. For instance, program

$$\begin{array}{l} p \\ q \leftarrow r \end{array} \tag{2}$$

has a single answer set $\{p\}$. Intuitively, the ASP semantics states that there are "no grounds" for deriving r or q, hence per CWA they are not the case.

The so called choice rules are used to "locally eliminate" CWA. *Choice rules* of the form

$$\{a_0\} \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n, \tag{3}$$

are important constructs of common ASP dialects. Consider a program resulting from appending choice rule

$$\{r\}. \tag{4}$$

to program (2). This program has two answer sets $\{p\}$ and $\{p, q, r\}$. The choice rule (4) tells that, intuitively, that an atom r *may* be a part of an answer set.

2 Formalization of Graph coloring problem by means of a Propositional logic program

Consider a *graph coloring* problem *GC*:

> *A* 3*-coloring of a directed graph* (V, E) *is a labeling of its vertexes with at most* 3 *colors (named,* 1*,* 2 *and* 3*) such that no two vertexes sharing an edge have the same color.*

[2] `https://works.bepress.com/yuliya_lierler/71/`

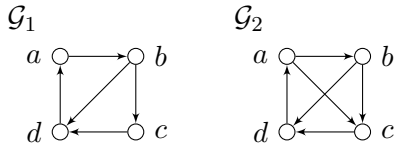

Figure 1: Sample graphs $\mathcal{G}_1$ and $\mathcal{G}_2$.

For instance, see two specific graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ in Figure 1. It is easy to see that there are six distinct 3-colorings for graph $\mathcal{G}_1$ including the following:

> assigning color 1 to vertexes a and c, color 2 to vertex b, and color 3 to vertex d forms a 3-coloring of $\mathcal{G}_1$.

We denote this 3-coloring of graph $\mathcal{G}_1$ by $\mathcal{S}_{\mathcal{G}_1}$. Graph $\mathcal{G}_2$ has no 3-colorings.

A solution to this problem can be described by a set of atoms of the form c_{vi} ($v \in V$ in a given graph (V, E) and $i \in \{1, 2, 3\}$); including c_{vi} in the set indicates that an assertion that vertex v is assigned color i. A solution is a set X satisfying the following conditions:

1. Each vertex must be assigned a color
2. A vertex may not be assigned more that one color
3. Vertexes sharing an edge must be assigned a distinct color.

A following propositional logic program under answer set semantics encodes these specifications so that its answer sets are in one to one correspondence with the 3-colorings of a given graph (V, E):

$$\begin{array}{ll} \{c_{vi}\} & (v \in V, 1 \leq i \leq 3) \\ \leftarrow \mathit{not}\ c_{v1}, \mathit{not}\ c_{v2}, \mathit{not}\ c_{v3}. & (v \in V). \\ \leftarrow c_{vi}, c_{vj} & (v \in V,\ 1 \leq i < j \leq 3), \\ \leftarrow c_{vi}, c_{wi} & (\{v, w\} \in E,\ 1 \leq i \leq 3), \end{array} \tag{5}$$

We now provide an intuitive reading of this program.

- A collection of choice rules for each vertex v captured by the first line in (5) intuitively says that vertex v may be assigned some colors. *(Condition 1)*

$\{c_{a1}\}$	$\{c_{a2}\}$	$\{c_{a3}\}$	$\{c_{b1}\}$	$\{c_{b2}\}$	$\{c_{b3}\}$	$\{c_{c1}\}$	$\{c_{c2}\}$	$\{c_{c3}\}$	$\{c_{d1}\}$	$\{c_{d2}\}$	$\{c_{d3}\}$
$\leftarrow not\ c_{a1}, not\ c_{a2}, not\ c_{a3}.$											
$\leftarrow not\ c_{b1}, not\ c_{b2}, not\ c_{b3}.$											
$\leftarrow not\ c_{c1}, not\ c_{c2}, not\ c_{c3}.$											
$\leftarrow not\ c_{d1}, not\ c_{d2}, not\ c_{d3}.$											
$\leftarrow c_{a1},\ c_{a2}$			$\leftarrow c_{a1},\ c_{a3}$			$\leftarrow c_{a2},\ c_{a3}$					
$\leftarrow c_{b1},\ c_{b2}$			$\leftarrow c_{b1},\ c_{b3}$			$\leftarrow c_{b2},\ c_{b3}$					
$\leftarrow c_{c1},\ c_{c2}$			$\leftarrow c_{c1},\ c_{c3}$			$\leftarrow c_{c2},\ c_{c3}$					
$\leftarrow c_{d1},\ c_{d2}$			$\leftarrow c_{d1},\ c_{d3}$			$\leftarrow c_{d2},\ c_{d3}$					
$\leftarrow c_{a1},\ c_{b1}$			$\leftarrow c_{a2},\ c_{b2}$			$\leftarrow c_{a3},\ c_{b3}$					
$\leftarrow c_{d1},\ c_{a1}$			$\leftarrow c_{d2},\ c_{a2}$			$\leftarrow c_{d3},\ c_{a3}$					
$\leftarrow c_{b1},\ c_{d1}$			$\leftarrow c_{b2},\ c_{d2}$			$\leftarrow c_{b3},\ c_{d3}$					
$\leftarrow c_{c1},\ c_{d1}$			$\leftarrow c_{c2},\ c_{d2}$			$\leftarrow c_{c3},\ c_{d3}$					
$\leftarrow c_{b1},\ c_{c1}$			$\leftarrow c_{b2},\ c_{c2}$			$\leftarrow c_{b3},\ c_{c3}$					

Figure 2: Logic program for 3-coloring for graph $\mathcal{G}_1$.

- The second line states that it is impossible that a vertex is not assigned a color. *(Condition 1)*
- The third line says that it is impossible that a vertex is assigned two colors. *(Condition 2)*
- The fourth line states that it is impossible that any two adjacent vertexes are assigned the same color. *(Condition 3)*

Recall graph $\mathcal{G}_1$ in Figure 1. Figure 2 presents a logic program in spirit of (5) for $\mathcal{G}_1$. Horizontal lines separate the clauses that come from distinct "schematic rules" in (5). This program has six answer sets including

$$\{c_{a1},\ c_{b2},\ c_{c1},\ c_{d3}\},$$

which captures solution $\mathcal{S}_{\mathcal{G}_1}$. To encode the 3-coloring problem for graph $\mathcal{G}_2$ one has to extend the set of rules in Figure 2 with rules

$$\leftarrow c_{a1}, c_{c1} \qquad \leftarrow c_{a2}, c_{c2} \qquad \leftarrow c_{a3}, c_{c3}.$$

This program has no answer sets, which captures the fact that this graph has no 3-colorings.

The ASP specification (5) illustrates the use of the so-called GENERATE and TEST methodology within answer set programming paradigm. The

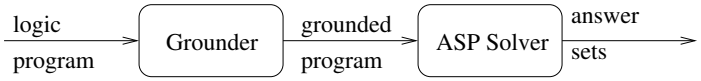

Figure 3: Common Architecture of ASP Systems

GENERATE part of the specification "defines" a collection of "perspective" answer sets that can be seen as potential solutions. The TEST part consists of conditions that eliminate the "perspective" answer sets of the GENERATE part that do not correspond to solutions. The first line in (5) corresponds to GENERATE: saying that any subset of the atoms of the form c_{vi} ($v \in V$ in a given graph (V, E) and $i \in \{1, 2, 3\}$) forms a potential solution. The remaining lines correspond to TEST. Observe, how choice rules provide a convenient tool in ASP for formulating GENERATE, whereas constraints are used to formulate TEST.

3 Programs with Variables and Grounding

ASP practitioners develop applications that rely on languages, which go beyond propositional/ground atoms. Figure 3 presents a typical architecture of an answer set programming tool that encompasses two parts: a system called *grounder* and a system called *solver*. The former is responsible for eliminating variables in a program. The latter is responsible for finding answer sets of a respective propositional (ground) program. For instance, system GRINGO[3] is a well known grounder that serve as front-ends for many solvers including CLASP. A combination of GRINGO and CLASP is known as system CLINGO.

We recall that given a signature σ consisting of object constants, variables, predicate symbols, and function symbols (where predicate and function symbols are associated with a nonnegative integer n called arity, so that we identify function symbols of arity 0 with object constants),

1. Any object constant or variable in σ is a *term*, and

   an expression of the form $f(t_1, \ldots, t_n)$ is a *term* where f is a function symbol in σ of arity $n > 0$ and $t_1, \ldots t_n$ are terms.

2. Any predicate symbol in σ of arity 0 is an atom, and

[3] `http://potassco.sourceforge.net/` .

an expression of the form $p(t_1, \ldots, t_n)$ is an *atom* where p is a predicate symbol in σ of arity n and $t_1, \ldots t_n$ are terms.

3. Any term that contains no variables is called *ground*. Similarly, any atom that contains no variables is *ground*. Otherwise, we refer to these entities as non-ground.

A *logic program with variables* is a finite set of rules of the form (1), where a_0 is symbol $\bot$ or a non-ground atom; and $a_1, \ldots, a_p$ are non-ground atoms. Grounding a logic program replaces each rule with all its instances obtained by substituting ground terms, formed from the object constants and function symbols occurring in the program, for all variables. For a program Π, by $ground(\Pi)$ we denote the result of its grounding. (We use the convention common in logic programming: variables are represented by capitalized identifiers.) We illustrate this concept on an example. Let Π be a program

$$\begin{array}{l} \{a(1)\} \ \ \{a(2)\} \ \ \{b(1)\} \\ c(X) \leftarrow a(X), b(X), \end{array} \tag{6}$$

$ground(\Pi)$ follows

$$\begin{array}{l} \{a(1)\} \ \ \{a(2)\} \ \ \{b(1)\} \\ c(1) \leftarrow a(1), b(1) \\ c(2) \leftarrow a(2), b(2). \end{array}$$

The answer sets of a program Π with variables are answer sets of $ground(\Pi)$. For instance, there are eight answer sets of program (6) including $\emptyset$ and set $\{a(1)\ b(1)\ c(1)\}$.

Problem 1. *(a) Consider the following program with variables*

$$\begin{array}{l} \{a(1)\}. \ \ \{a(2)\}. \ \ \{b(1)\}. \\ d(X, Y) \leftarrow a(X), b(Y). \end{array} \tag{7}$$

Construct the result of grounding for this program.

(b) Follow the link `https://potassco.org/clingo/run/` *. Replace symbol "$\leftarrow$" by ":-" in the ground program you constructed in (a) and let* CLINGO *run on your program using* reasoning mode *"enumerate all".*

(c) Using the same procedure as in (b), find all answer sets for the program listed in (a). Do answer sets found in (b) coincide with the ones enumerated in this step?

Given a program Π with variables, grounders often produce a variable-free program that is smaller than $ground(\Pi)$, but still has the same answer

sets as *ground*(Π); we call any such program an *image* of Π. For example, program

$$\{a(1)\}\ \ \{a(2)\}\ \ \{b(1)\}$$
$$c(1) \leftarrow a(1), b(1)$$

is an image of (6).

Passing parameter $-t$ in command line when calling CLINGO on a program will force the system to produce ground program in human readable form. Running CLINGO in online interface with checkbox "statistics" allows one to obtain valuable information about the execution of the system. For example, lines titled

- "Rules" provides number of rules in a grounding of the input and suggests the relative size of a ground program,
- "Choices" corresponds to the number of backtracks done by the system in search for the solution (we will learn of this feature in a little bit),
- "Time" reports the execution time of the system.

In command line to obtain statistics while running CLINGO use flag "--stats"

Problem 2. *Let* CLINGO *run on the program* (7) *using* reasoning mode *"enumerate all" and marking* checkbox *"statistics". What is the number of rules that* CLINGO *reports?*

This number corresponds to the size of the image (measured in number of rules) produced by GRINGO *after grounding the input program.*

Think of an image for program (7) *that is of the same size as* GRINGO *computed. List this image.*

When a program Π with variables has at least one function symbol and at least one object constant, grounding results in infinite *ground*(Π). Yet, even for an input program of this kind, grounders often find an image that is a finite propositional program (finite image). For instance, for program

$$\begin{array}{l} p(0) \\ q(f(X)) \leftarrow p(X) \end{array} \tag{8}$$

grounding results in infinite program outlined below

$$\begin{array}{l} p(0) \\ q(f(0)) \leftarrow p(0) \\ q(f(f(0))) \leftarrow p(f(0)) \\ q(f(f(f(0)))) \leftarrow p(f(f(0))) \\ \ldots \end{array}$$

A finite image of program (8) follows

$$p(0)$$
$$q(f(0)) \leftarrow p(0).$$

Program

$$p(0)$$
$$q(f(0))$$

is another image of (8). In fact, given program (8) as an input grounder GRINGO will generate the latter image.

To produce images for input programs, grounders follow techniques exemplified by intelligent grounding. Different grounders implement distinct procedures so that they may generate different images for the same input program. One can intuitively measure the quality of a produced image by its size so that the smaller the image is the better. A common syntactic restriction that grounders pose on input programs is "safety". A program Π is *safe* if every variable occurring in a rule of Π also occurs in positive body of that rule. For instance, programs (6) and (8) are safe. The safety requirement suggests that positive body of a rule must contain information on the values that should be substituted for a variable in the process of grounding. Safety is instrumental in designing grounding techniques that utilize knowledge about the structure of a program for constructing smaller images. The GRINGO grounder and the grounder of the DLV system expect programs to be safe. For programs with function symbols, to guarantee that the grounding process terminates, grounders pose additional syntactic restrictions (in other words, to guarantee that a grounder is able to construct a finite image).

4 Formalization of Graph Coloring Problem by Means of a Logic Program with Variables

We now revisit our graph coloring example and illustrate how often a set of propositional rules that follow a simple pattern can be represented concisely by means of logic programs with variables. Recall program (5). We now capture atoms of the form c_{vi} by expressions $c(v, i)$, where c is a predicate symbol and v, i are object constants denoting a vertex v and color i respectively. Atom of the form $vtx(v)$, intuitively, states that an object constant v is a vertex, while atom $e(v, w)$ states that there is an edge from vertex v to vertex w in a given graph. Atom $color(i)$ states that an object constant i represents a color. Recall graph coloring problem GC for an input graph (V, E). We now present a program with variables that encodes a

solution to this problem. First, this program consists of facts that encode graph (V, E):

$$\begin{array}{ll} vtx(v) & (v \in V) \\ e(v, w) & (\{v, w\} \in E) \end{array} \tag{9}$$

Second, facts

$$color(c) \qquad (c \in \{1, 2, 3\}) \tag{10}$$

enumerate three colors of the problem. The following rules conclude the description of the program:

$$\{c(V, I)\} \leftarrow vtx(V),\ color(I) \tag{11}$$

$$\leftarrow not\ c(V, 1), not\ c(V, 2), not\ c(V, 3),\ vtx(V) \tag{12}$$

$$\leftarrow c(V, I),\ c(V, J),\ I < J,\ vtx(V),\ color(I),\ color(J) \tag{13}$$

$$\leftarrow c(V, I),\ c(W, I),\ vtx(V),\ vtx(W),\ color(I),\ e(V, W) \tag{14}$$

These rules are the counterparts of groups of rules in propositional program (5). Indeed,

- rule (11) states that every vertex may be assigned some colors;
- the second rule (12) states that it is impossible that a vertex is not assigned a color;
- rule (13) says that it is impossible that a vertex is assigned two colors; and
- rule (14) says that it is impossible that any two adjacent vertexes are assigned the same color.

Programs with variables permit for a concise encoding of an instance of a search problem. Indeed, size of a program composed of rules (9-12) is almost identical to the size of a given graph (V, E). There are $|V| + |E| + 7$ rules in this program. On the other hand, the line

$$\leftarrow c_{vi}, c_{wi} \qquad (\{v, w\} \in E,\ 1 \leq i \leq 3)$$

of program (5) alone encapsulates $3|E|$ rules.

5 Modeling of Search Problems in ASP

Answer set programming provides a general purpose modeling language that supports elaboration tolerant solutions for search problems. We now define a

search problem abstractly. A *search problem* P consists of a set of instances with each *instance* I assigned a finite set $S_P(I)$ of solutions. In answer set programming to solve a search problem P, we construct a program Π_P that captures problem specifications so that when extended with facts D_I representing an instance I of the problem, the answer sets of $\Pi_P \cup D_I$ are in one to one correspondence with members in $S_P(I)$. In other words, answer sets describe all solutions of problem P for the instance I. Thus solving of a search problem is reduced to finding a uniform encoding of its specifications by means of a logic program with variables.

For example, an instance of the graph coloring search problem GC is a graph. All 3-colorings for a given graph form its solutions set. Consider any graph (V, E). By $D_{(V,E)}$ we denote facts in (9) that encode graph (V, E). By Π_{gc} we denote a program composed of rules in (10-12). This program captures specifications of 3-coloring problem so that answer sets of $\Pi_{gc} \cup D_{(V,E)}$ correspond to solutions to instance graph (V, E) of a problem. Recall graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ presented in Figure 1. Facts $D_{\mathcal{G}_1}$ representing $\mathcal{G}_1$ follow

$$vtx(a)\ \ vtx(b)\ \ vtx(c)\ \ vtx(d)\ \ e(a,b)\ \ e(b,c)\ \ e(c,d)\ \ e(d,a)\ \ e(b,d).$$

Program $\Pi_{gc} \cup D_{\mathcal{G}_1}$ has six answer sets, including

$$\begin{array}{l}\{vtx(a)\ vtx(b)\ vtx(c)\ vtx(d)\\ e(a,b)\ e(b,c)\ e(c,d)\ e(d,a)\ e(b,d)\\ color(1)\ color(2)\ color(3)\\ c(a,1)\ c(b,2)\ c(c,1)\ c(d,3)\},\end{array}$$

which captures solution $\mathcal{S}_{\mathcal{G}_1}$. Similarly, we can use encoding Π_{gc} to establish whether 3-colorings exist for graph $\mathcal{G}_2$. Facts $D_{\mathcal{G}_2}$ representing $\mathcal{G}_2$ consists of facts in $D_{\mathcal{G}_1}$ and an additional fact $e(a, c)$. Program $\Pi_{gc} \cup D_{\mathcal{G}_2}$ has no answer sets suggesting that no 3-colorings exist for graph $\mathcal{G}_2$.

It is important to mention that the languages supported by ASP grounders and solvers go beyond rules presented here. For instance, GRINGO versions 4.5+ support such constructs as aggregates, cardinality expressions, intervals, pools. It is beyond the scope of this lecture to formally discuss these constructs, but it is worth mentioning that they generally allow us more concise, intuitive, and elaboration tolerant encodings of problems. Also, they often permit to utilize more sophisticated and efficient procedures in solving.

For instance, a single rule

$$\leftarrow not\ 1\#count\{V, I : c(V, I)\}1,\ vtx(V). \tag{15}$$

can replace two rules (12) and (13) in program Π_{gc}. This rule states that

it must be the case that a vertex is assigned exactly one color. (16)

This shorter program will result in smaller groundings for instances of the GC problem paving the way to more efficient solving. Cardinality construct

$$1\#count\{V, I : c(V, I)\}1$$

intuitively suggests us to count, for a given value of V the tuples (V, I) for which atom of the form $c(V, I)$ belongs to the answer set. Number 1 to the right and to the left of this aggregate expression tells us a specific condition on the count, in particular, that it has to be exactly 1. Indeed, the number to the right suggests at most count whereas the number to the left suggest at least count. Cardinality expressions form only one example of multitude of constructs that GRINGO language offers for effective modeling of problem specifications.

It is worth noting that the GRINGO language also provides *syntactic sugar*, i.e., convenient abbreviations for groups of rules. For example, expression

$$edge(a, b; b, c; c, d).$$

abbreviates the set of facts

$$edge(a, b).\ edge(b, c).\ edge(c, d).$$

Expression

$$1\{c(V, I) : color(I)\} \leftarrow vtx(V).$$

abbreviates rules (11) and (12). So that we can encode Condition 1 listed in Section 2 by one rule.

Expression

$$1\{c(V, I) : color(I)\}1 \leftarrow vtx(V). \tag{17}$$

abbreviates a collection of two rules (11) and (15). Intuitively, we can read the meaning of this rule as stated in (16) that we used to characterize an intuitive meaning of constraint (15). Although English statements turn out to be the same for rules (15) and (17), the formal meaning of mathematical expression (17) obviously extends that of (15). In addition to a constraint on solutions captured by (15) [which is one of the rules abbreviated by expression (17)], choice rule (11) [the other rule abbreviated by expression (17)] provides ground for atoms of the form $c(\cdot, \cdot)$ be part of solutions. Or, in other words, choice rule (11) removes closed world assumption from $c(\cdot, \cdot)$ atoms.

Problem 3. *Note that directive #show c/2. added to a* CLINGO *program* Π_{gc} *allows one to instruct* CLINGO *to only output these atoms in the computed answer sets that have the form* $c(\cdot, \cdot)$.

(a) Recall the statement of graph coloring problem. Imagine the following extension to that statement: there is at most one node in a given graph colored by color named 1. *Extend the program* Π_{gc} *so that this new statement is respected.*

(b) For graph $\mathcal{G}_1$*, how many solutions to the new graph coloring problem stated in (a) are there?*

(c) Use CLINGO *to test your solution in by running it on the program* $\Pi_{gc} \cup D_{\mathcal{G}_1}$ *extended with the code you developed in (a). List the answer sets that* CLINGO *computes (list only the atoms of the form* $c(\cdot, \cdot)$*).*

6 The Generate-Define-and-Test Modeling Methodology of ASP

Previously, we presented how the GENERATE and TEST methodology is applicable within answer set programming. Yet, an essential feature of logic programs is their ability to elegantly and concisely "define" predicates. Logic programs provide a convenient language for expressing inductive/recursive definitions.

The GENERATE, DEFINE, and TEST is a typical methodology used by ASP practitioners in designing programs. It generalizes the GENERATE and TEST methodology discussed earlier. The roles of the GENERATE and TEST parts of a program stay the same so that, informally,

- GENERATE defines a large collection of answer sets that could be seen as potential solutions, while
- TEST consists of rules that eliminate the answer sets of the GENERATE part that do not correspond to solutions.
- The DEFINE section expresses additional, auxiliary concepts and connects the GENERATE and TEST parts.

To illustrate the essence of DEFINE, consider a *Hamiltonian cycle* search problem:

> Given a directed graph (V, E), the goal is to find a *Hamiltonian cycle* — a set of edges that induce in (V, E) a directed cycle going through each vertex exactly once.

This is an important combinatorial search problem related to Traveling Salesperson problem. A solution can be described by a set of atoms of the form $in(v, w)$, $(v, w) \in E$ of the given graph (V, E); including $in(v, w)$ in the set indicates that an edge from vertex v to vertex w is part of a found Hamiltonian cycle. A solution is a set X satisfying the following conditions

1. the Hamiltonian cycle is formed by the edges of a given graph (V, E), or, in other words, the extension of predicate in is a subset of E,
2. X does not contain a pair of different atoms of the form $in(u, v)$, $in(u', v)$ (two selected edges end at the same vertex; thus we visit node u only once),
3. For each pair u, v of vertexes, X is such that (u, v) is a part of the transitive closure of in relation defined by X. (Thus, a found subset of edges of the graph is indeed a cycle.) Recall that
   - the *transitive closure* of a binary relation R (in our case relation in) on a set Y of elements (in our case the set of the vertexes of the given graph) is the smallest relation on Y that contains R and is transitive.
   - a *transitive* relation R on set Y of elements is such that for any three elements a, b, c (not necessarily distinct elements) in Y the following property holds if a is in relation R with b and b is in relation R with c then a is in relation R with c.

We now formalize the specifications of Hamiltonian cycle problem by means of GENERATE, DEFINE, and TEST methodology. As in the encoding Π_{gc} of the graph coloring problem, we use expressions of the form $vtx(v)$ and $e(v, w)$ to encode an input graph.

Choice rule

$$\{in(X, Y)\} \leftarrow e(X, Y) \tag{18}$$

forms the GENERATE part of the problem. This rule states that any subset of edges of a given graph may form a Hamiltonian cycle (*Condition 1*). Answer sets of a program composed of this rule and a set of facts encoding an input graph will correspond to all subsets of edges of the graph. For instance, program composed of facts $D_{\mathcal{G}_1}$ that encode directed graph $\mathcal{G}_1$ introduced in Figure 1 extended by rule (18) has 32 answer sets each representing a different subset of its edges.

The remaining *Conditions 2-3* are captured in the TEST part. To formulate Condition 3, an auxiliary concept of *reachable* (a transitive closure of

in) is required so that we can capture the restriction that a found subset of edges of the graph is also a cycle. The DEFINE part follows

$$\begin{array}{l} reachable(V,V) \leftarrow vtx(V) \\ reachable(U,W) \leftarrow in(U,V),\ reachable(V,W), \\ \qquad\qquad\qquad\qquad vtx(U),\ vtx(V),\ vtx(W) \end{array} \tag{19}$$

These rules define *transitive closure* of the predicate *in*: all pairs of vertexes (u, v) such that v can be reached from u by following zero or more edges that are *in*.

We are now ready to state the TEST part composed of three rules

$$\begin{array}{l|l} \text{Condition 2} & \leftarrow in(U,V),\ in(W,V),\ U \neq W,\ vtx(U;V;W) \\ \text{Condition 3} & \leftarrow not\ reachable(U,V),\ vtx(U;V). \end{array} \tag{20}$$

Rules (18), (19), and (20) form a program Π_{hc} that captures specifications of Hamiltonian cycle search problem. Extending Π_{hc} with facts representing a directed graph results in a program whose answer sets describe all Hamiltonian cycles of this graph. For example, program $\Pi_{hc} \cup D_{\mathcal{G}_1}$ has only one answer set. Set

$$\{in(a,b)\ in(b,c)\ in(c,d)\ in(d,a)\}$$

contains all *in*-edges of that answer set stating that edges (a, b), (b, c), (c, d), and (d, a) form the only Hamiltonian cycle for graph $\mathcal{G}_1$.

Concise encoding of transitive closure is a feature of answer set programming that constitutes an essential difference between ASP and formalisms based on classical logic. For example, transitive closure is not expressible by first-order formulas. Thus any subset of first-order logic taken as the language with variables for modeling search problems declaratively will fail at defining (directly) concepts that rely on transitive closure.

In mastering the art of answer set programming it is enough to develop intuitions about answer sets of the programs with variables that are formed in accordance with the GENERATE, DEFINE, and TEST methodology. Some of these intuitions will stem from the general properties you encountered earlier such as any element of answer set must appear in the head of some rule in a program. For the general definition of an answer set, it is more difficult to develop intuitions on what answer sets conceptually are and unnecessary.

Problem 4. *Recall that the rule*

$$\leftarrow in(U,V),\ in(W,V),\ U \neq W,\ vtx(U;V;W)$$

in Π_{hc} *states that no two selected edges end at the same node. Rewrite this rule using aggregate #count exemplified in rule* (15)*. You may wish to consult Clingo-Gringo manual Section 3.1.12 for more details on aggregates.*

7 ASP Formulation of n-Queens

We now turn our attention to another combinatorial search problem: n-queens problem.

> The goal is to place n queens on an $n \times n$ chessboard so that no two queens would be placed on the same row, column, and diagonal.

A solution can be described by a set of atoms of the form $q(i, j)$ $(1 \leq i, j \leq n)$; including $q(i, j)$ in the set indicates that there is a queen at position (i, j). A solution is a set X satisfying the following conditions:

1. the cardinality of X is n,
2. X does not contain a pair of different atoms of the form $q(i, j)$, $q(i', j)$ (two queens on the same row),
3. X does not contain a pair of different atoms of the form $q(i, j)$, $q(i, j')$ (two queens on the same column),
4. X does not contain a pair of different atoms of the form $q(i, j)$, $q(i', j')$ with $|i' - i| = |j' - j|$ (two queens on the same diagonal).

Here is the representation of this program in the input language of CLINGO:

```
number(1..n).

%Condition 1 and 2
1{q(K,J): number(K)}1:- number(J).

%Condition 3
:-q(I,J), q(I,J1), J<J1.

%Condition 4
:-q(I,J), q(I1,J1), J<J1, |I1-I|==J1-J.
```

We name this program *queens.clingo*.

Appending the line

```
# const n=8.
```

to the code in *queens.clingo* will instruct answer set system CLINGO to search for solution for 8-queens problem. Alternatively, the command line

```
clingo -c n=8 queens.clingo
```

instructs the answer set system CLINGO to find a single solution for 8-queens problem, whereas the command line

```
clingo -c n=8 queens.clingo 0
```

instructs CLINGO to find all solutions to 8-queens program. The command line

```
gringo -c n=8 queens.clingo >  queens.8.grounded
```

instructs the grounder GRINGO to ground 8-queens problem; the ground problem (ready for processing with CLASP) is stored in file *queens.8.grounded*. The command lines

```
gringo -t -c n=8 queens.clingo
```

or

```
clingo -t -c n=8 queens.clingo
```

will produce human-readable grounded 8-queens problem.

The command line

```
clasp < queens.8.grounded
```

will instruct the answer set solver CLASP to look for answer sets of a program in *queens.8.grounded*.

An extract from the output of the last command line follows

```
...
Answer: 92
number(1) number(2) number(3) number(4)
number(5) number(6) number(7) number(8)
q(5,8) q(7,7) q(2,6) q(6,5) q(3,4) q(1,3) q(8,2) q(4,1)
SATISFIABLE
```

This 92nd solution found by the solver encodes the following valid configuration of queens on the board

```
 1 2 3 4 5 6 7 8
1    Q
2          Q
3      Q
4Q
5              Q
6        Q
7            Q
8  Q
```

Similarly, appending the line

```
# const n=4.
```

to the code in *queens.clingo* will instruct CLINGO to solve 4-queens problem. The command line

```
clingo -c n=4 queens.clingo 0
```

instructs CLINGO to find all solutions for 4-queens problem.

Problem 5. *(a) Use* CLINGO *to find all solutions to the* 8*-queens problem that have a queen at* $(1, 1)$*. How many solutions of the kind are there?*

(b) Use CLINGO *to find all solutions to the* 12*-queens problem that have a queen at* $(1, 1)$*. How many solutions of the kind are there?*

Submit the lines of code that you wrote to solve these problems.

Problem 6. *(a) Use* CLINGO *to find all solutions to the* 8*-queens problem that have no queens in the* 4×4 *square in the middle of the board. How many solutions of the kind are there?*

(b) Use CLINGO *to find all solutions to the* 10*-queens problem that have no queens in the* 4×4 *square in the middle of the board. How many solutions of the kind are there?*

Submit the lines of code that you wrote to solve these problems.

Acknowledgments

Parts of this handout follow *What is answer set programming to propositional satisfiability*, Yuliya Lierler, Constraints, July 2017, Volume 22, Issue 3, pp 307337 available at `https://link.springer.com/article/10.1007/s10601-016-9257-7`.

Handout on Algorithms in Backtracking Search behind SAT and ASP

Yuliya Lierler
University of Nebraska Omaha

Introduction

We now turn out attention to search algorithms underlying ASP technology. In particular, we will focus on the techniques employed by answer set solver such as CLASP. Recall that CLASP is only one building block of an answer set system CLINGO that also incorporates grounder called GRINGO. In the scope of this course we ignore the details behind grounders, but note that these are highly nontrivial systems solving a complex and computationally intense task of *intelligent* instantiation.

The algorithms behind majority answer set solvers fall into group of so called backtracking search algorithms.

> Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution. *(Wikipedia)*

The Davis-Putnam-Logemann-Loveland (DPLL) procedure is a classic example of backtracking search algorithms. DPLL is a method for deciding the satisfiability of propositional logic formula in conjunctive normal form, or, in other words, for solving the propositional satisfiability problem. Algorithms used by answer set solvers share a lot in common with DPLL. In this handout, we thus begin by presenting DPLL procedure. We then discuss its extensions suitable for computing answer sets of a program in place.

1 Satisfiability Solving: Davis-Putnam-Logemann-Loveland Procedure

Recall that a *literal* is an atom or a negated atom. A signature is a set of atoms. Given a propositional formula, the set of atoms occurring in it is considered to be its signature by default. A *clause* is a disjunction of literals (possibly the empty disjunction $\bot$). A formula is said to be in *conjunctive normal form (CNF)* if it is a conjunction of clauses (possibly the empty conjunction $\top$). The task of deciding whether a CNF formula is satisfiable is called a satisfiability (SAT) problem. Recall that an *interpretation/assignment* over a signature is a mapping from the elements of the signature to truth values `f` or `t`. For example, given formula

$$(p \wedge q) \vee r \tag{1}$$

there are 8 interpretations in its signature $\{p, q, r\}$ including the following

interpretation	p	q	r
I_1	f	t	t
I_2	f	t	f

A formula is called *satisfiable* if we can find an interpretation over its signature so that this formula is evaluated to true under this interpretation. (We assume the familiarity with the interpretation functions for the classical logic connectives $\top, \bot, \neg, \wedge$ and $\vee$.) We say that in such case an interpretation *satisfies* a formula and also call it a *model*. For instance, interpretation I_1 satisfies formula (1) while I_2 does not. In other words, I_1 is a model of formula (1). Hence this formula is also satisfiable. It is common to identify an interpretation over signature σ with the set of literals and also with the set of atoms in an intuitive way. For instance, the table below presents such a mapping for interpretations I_1 and I_2.

interpretation	p	q	r	set of literals	set of atoms w.r.t. $\sigma = \{p, q, r\}$
I_1	f	t	t	$\{\neg p, q, r\}$	$\{q, r\}$
I_2	f	t	f	$\{\neg p, q, \neg r\}$	$\{q\}$

Later in the discourse we frequently use the word interpretation to denote a set of literals.

1.1 DPLL by means of Pseudocode

The Davis-Putnam-Logemann-Loveland (DPLL) procedure is an algorithm for deciding the satisfiability of propositional logic formula in CNF. DPLL also allows to find a satisfying interpretation of a formula if it exists. Enhancements of DPLL form modern SAT solving technology.

We now state some terminology useful in presenting DPLL. For a literal of the form A we say that $\neg A$ is its *complement*, whereas for a literal of the form $\neg A$, atom A is its *complement*. A set M of literals is *consistent* when it does not contain complementary pairs A, $\neg A$. Otherwise, we call a set *inconsistent*. Sets $\{a, b, \neg c\}$ and $\{a, b, c, \neg c\}$ exemplify consistent and inconsistent sets of literals, respectively. In the sequel by *wrt* we abbreviate the phrase *with respect to*. We say that a clause $l_1 \vee \cdots \vee l_n$ is *unit-ary wrt* the set M of literals, when

- there is i such that $0 \leq i \leq n$ and $l_i \notin M$ (we call literal l_i a *unit literal*), and
- for every j such that $0 \leq j \leq n$ and $j \neq i$, $\overline{l_j} \in M$.

For instance, $a \vee \neg b \vee c$ is a unit-ary clause wrt $\{\neg a, b\}$ and c is its unit literal. Clause $a \vee \neg b$ is a unit-ary clause wrt set $\{\neg a, b\}$, where both a and $\neg b$ are unit literals. We say that a clause $l_1 \vee \cdots \vee l_n$ is *satisfied by* the set M of literals, when for some i such that $0 \leq i \leq n$ the following holds $l_i \in M$. We say that a literal l is *unassigned* by a set M of literals if neither l nor its complement $\bar{l}$ is in M; otherwise we say that literal l is *assigned* by M.

Consider the procedure called *unit propagation* presented in Figure 1. This procedure is invoked on a consistent set M of literals. To apply unit propagation to a given CNF formula F, UNIT-PROPAGATE is invoked with F and $M = \emptyset$. For instance, to apply unit propagation to

$$p \wedge (\neg p \vee \neg q) \wedge (\neg q \vee r) \tag{2}$$

we invoke UNIT-PROPAGATE with this formula as F and with $\emptyset$ as M. After the first execution of the body of the loop,

$$M = \{p\};$$

UNIT-PROPAGATE(F, M)
  while M *is a consistent set of literals*,
      and F has a unit-ary clause wrt M so that l is a unit literal of that clause
    $M \leftarrow M \cup \{l\}$
  end

Figure 1: Unit propagation

DPLL(F, M)
  UNIT-PROPAGATE(F, M);
  if M *is an inconsistent set of literals* **then** return;
  if every atom occurring in F is assigned by M **then** exit with *a model* of M;
  $l \leftarrow$ a literal containing an atom from F unassigned by M;
  DPLL$(F, M \cup \{l\})$;
  DPLL$(F, M \cup \{\bar{l}\})$

Figure 2: Davis-Putnam-Logemann-Loveland procedure

after the second iteration

$$M = \{p, \neg q\}.$$

This computation shows that any model of the given formula is such that p is assigned to t and q is assigned to f.

There are two cases when the process of unit propagation alone is sufficient for solving the satisfiability problem for given F. Consider the value of M upon the termination of UNIT-PROPAGATE$(F, \emptyset)$. First, if F is such that all of its clauses are satisfied by M, as in the example above, then F is satisfiable, and any satisfying interpretation can be easily extracted from M. In the example above

$$M = \{p, \neg q\}$$

gives rise to two models of (2):

interpretation	p	q	r
I_1	t	f	f
I_2	t	f	t

Second, if M is an inconsistent set of literals then F is not satisfiable.

Problem 1. *Use unit propagation to decide whether the formula*

$$p \wedge (p \vee q) \wedge (\neg p \vee \neg q) \wedge (q \vee r) \wedge (\neg q \vee \neg r)$$

is satisfiable.

The Davis-Putnam-Logemann-Loveland procedure presented in Figure 2 is an extension of the unit propagation method that can solve the satisfiability problem for *any* CNF formula. Like UNIT-PROPAGATE, it is initially invoked with F and $M = \emptyset$.

Example 1. *Consider, for instance, the application of the DPLL procedure to*

$$(\neg p \vee q) \wedge (\neg p \vee r) \wedge (q \vee r) \wedge (\neg q \vee \neg r). \tag{3}$$

First DPLL *is called with this formula as F and with $\emptyset$ as M (Call 1). After the call to* UNIT-PROPAGATE*, the value of M remains the same. Assume that the literal selected as l is p. Now* DPLL *is called recursively with F and $\{p\}$ as M (Call 2). After the call to* UNIT-PROPAGATE*, M turns into an inconsistent set $\{p, q, r, \neg r\}$ (or an inconsistent set $\{p, q, r, \neg q\}$). Thus* DPLL *returns. Next* DPLL *is called with $\{\neg p\}$ as M (Call 3). After the call to* UNIT-PROPAGATE*, M remains the same. Assume that the literal selected as l is q. Then* DPLL *is called with $\{\neg p, q\}$ as M (Call 4). After the call to* UNIT-PROPAGATE*, $M = \{\neg p, q, \neg r\}$. Since M is consistent and assigns every atom occurring in F,* DPLL *returns M as a model:*

p	q	r
f	t	f

Problem 2. *How would this computation be affected by selecting $\neg p$ as l in Call 1? By selecting $\neg q$ as l in Call 3?*

1.2 DPLL by means of Transition Systems

In the previous section we described the DPLL procedure using pseudocode. Here we use a different method to present this algorithm. In particular, we use a *transition system* that can be viewed as an abstract representation of the underlying DPLL computation. This transition system captures what "states of computation" are, and what transitions between states are allowed. In this way, a transition system defines a directed graph such that every execution of the DPLL procedure corresponds to a path in this graph. Some edges may correspond to unit propagation steps, some to decision, some to backtracking. Later in the handout, we will follow this approach for describing a search algorithm suitable for computing answer sets of a program.

For a set σ of atoms, a *record* relative to signature σ is a sequence $l_1 \dots l_{n-1}\ l_n$ of distinct literals over σ, with some literals possibly annotated by Δ, which marks them as *decision* literals, so that

1. $l_1 \dots l_{n-1}$ contains no complementary pairs $A, \neg A$, and
2. a decision literal may not be preceded by the complement of this literal in the sequence.

A *state* relative to σ is either a distinguished state *FailState* or a record relative to σ. For instance, the states relative to a singleton set $\{p\}$ are

$$\mathit{FailState},\ \ \emptyset,\ \ p,\ \ \neg p,\ \ p^{\Delta},\ \ \neg p^{\Delta},\ \ p\ \neg p,\ \ p^{\Delta}\ \neg p,\ \ \neg p\ p,\ \ \neg p^{\Delta}\ p, .$$

Note how sequences of literals such as $p\ p$, $p\ p^{\Delta}$, $p\ \neg p^{\Delta}$, $p^{\Delta}\ \neg p^{\Delta}$ do not form records (the former two sequences are not formed by distinct literals; the later two sequences do not satisfy Condition 2). Similarly, while sequence $p\ q\ \neg q$ forms a record relative to signature $\{p, q\}$, sequence $p\ q\ \neg q\ \neg p$ is not a record (it does not satisfy Condition 1).

Frequently, we identify a record M with a set of literals, ignoring both the annotations and the order among its elements. This allows us to use notation stemming from set theory. For example, let M be a record $p\ q\ \neg q$, we identify an expression $p \in M$ with the condition $p \in \{p, q, \neg q\}$ that "checks" whether p is a member of set $\{p, q, \neg q\}$ Similarly we can speak

of literals being unassigned by a record, or a record being inconsistent. These terms were defined earlier in the handout for the sets of literals. For instance, states $p^\Delta\ \neg p$ and $p\ q\ \neg p$ are inconsistent. Also both q and $\neg q$ are unassigned by state $p^\Delta\ \neg p$, whereas both of them are assigned by $p\ q\ \neg p$.

Each CNF formula F determines its *DPLL graph* DP_F. The set of nodes of DP_F consists of the states relative to the signature of F. The edges of the graph DP_F are specified by four transition rules:

Unit Propagate: $M \Rightarrow M\ l$ if { there is a unit-ary clause in F w.r.t. M so that literal l is its unit literal

Decide: $M \Rightarrow M\ l^\Delta$ if l is unassigned by M

Fail: $M \Rightarrow \mathit{FailState}$ if { M is inconsistent, and M contains no decision literals

Backtrack: $P\ l^\Delta\ Q \Rightarrow P\ \bar{l}$ if { $P\ l^\Delta\ Q$ is inconsistent, and Q contains no decision literals.

A node (state) in the graph is *terminal* if no edge originates in it. The transition rule *Unit Propagate* is also often called a *propagator/inference* rule of DPLL.

The following proposition gathers key properties of the graph DP_F.

Proposition 1. *For any CNF formula F,*

(a) graph DP_F is finite and acyclic,

(b) any terminal state of DP_F other than FailState is a model of F,

(c) FailState is reachable from $\emptyset$ in DP_F if and only if F is unsatisfiable.

Thus, to decide the satisfiability of a CNF formula F it is enough to find a path leading from node $\emptyset$ to a terminal node M. If $M = \mathit{FailState}$, F is unsatisfiable. Otherwise, F is satisfiable and M is a model of F.

For instance, let $F_1 = \{p \vee q, \neg p \vee r\}$. Below we show a path in DP_{F_1} with every edge annotated by the name of the transition rule that gives rise to this edge in the graph:

$$\emptyset \overset{\mathit{Decide}}{\Rightarrow} p^\Delta \overset{\mathit{Unit\ Propagate}}{\Rightarrow} p^\Delta\ r \overset{\mathit{Decide}}{\Rightarrow} p^\Delta\ r\ q^\Delta. \tag{4}$$

The state $p^\Delta\ r\ q^\Delta$ is terminal. Thus, Proposition 1(b) asserts that F_1 is satisfiable and $\{p, r, q\}$ is a model of F_1. Another path in DP_{F_1} that leads us to concluding that set $\{p, r, q\}$ is a model of F_1 follows

$$\emptyset \overset{\mathit{Decide}}{\Rightarrow} p^\Delta \overset{\mathit{Decide}}{\Rightarrow} p^\Delta\ r^\Delta \overset{\mathit{Decide}}{\Rightarrow} p^\Delta\ r^\Delta\ q^\Delta. \tag{5}$$

We can view a path in the graph DP_F as a description of a process of search for a model of a formula F by applying transition rules of the graph. Therefore, we can characterize an algorithm of a SAT solver that utilizes the inference rules of DP_F by describing a strategy for choosing a path in DP_F. A strategy can be based, in particular, on assigning priorities to some or all transition rules of DP_F, so that a solver will never apply a transition rule in a state if a rule with higher priority is applicable to the same state. The DPLL algorithm can be captured by the following priorities:

$$\mathit{Backtrack}, \mathit{Fail} >> \mathit{Unit\ Propagate} >> \mathit{Decide}.$$

Note how path (6) in the graph DP_{F_1} respects priorities above, while path (5) does not. Thus DPLL will never explore the latter search trajectory given input F_1.

Problem 3. *Let G be formula (3). Then a pass in DP_G that can be seen as capturing the computation of DPLL described in Example 1 follows:*

$$\begin{array}{l} \emptyset \overset{\textit{Decide}}{\Rightarrow} p^{\Delta} \overset{\textit{Unit Propagate}}{\Rightarrow} p^{\Delta}\ q \overset{\textit{Unit Propagate}}{\Rightarrow} p^{\Delta}\ q\ r \overset{\textit{Unit Propagate}}{\Rightarrow} \\ p^{\Delta}\ q\ r\ \neg q \overset{\textit{Backtrack}}{\Rightarrow} \neg p \overset{\textit{Decide}}{\Rightarrow} \neg p\ q^{\Delta} \overset{\textit{Unit Propagate}}{\Rightarrow} \neg p\ q^{\Delta}\ \neg r \end{array} \tag{6}$$

*(a) List an alternative path to (6) in DP_G that also can be seen as capturing the computation of DPLL. (*Hint*: think of nondeterminism in* Unit-Propagate *procedure.)*

(b) Consider node q in graph DP_G. List all the edges that leave this node in DP_G. Annotate these edges by transition rules that they are due. Specify nodes to which these edges lead. For instance,

$$q \overset{\textit{Decide}}{\Rightarrow} q\ p^{\Delta}$$

is one of these edges. (c) Consider node $p^{\Delta}\ q\ r\ \neg q$ in graph DP_G. List all the edges that leave this node in DP_G (as in the previous question).

2 From ASP to SAT

A number of transformations from logic programs under answer set semantics to SAT exist. Given a propositional logic program Π, there are two kinds of transformations:

- transformations that preserve the vocabulary of Π and form a propositional theory F_{Π} that is *equivalent* to Π. In other words, models of Π and F_{Π} coincide.
- transformations that may contain "new atoms" so that the answer sets for Π can be obtained by removing these atoms from the models of constructed F_{Π}.

Remarkable transformation of the former kind is called *completion*. For a large syntactic class of programs ("tight" programs), the models of program's completion coincide with the answer sets of a program. This fact is exploited in several state-of-the-art answer set solvers including clasp (a solver of clingo). For example, for tight programs clasp practically runs a (significantly enhanced) DPLL procedure on program's completion to obtain answer sets of a program.

Answer Set Solving. We are now ready to present an extension to the DPLL algorithm that captures a family of backtrack search procedures for finding answer sets of a propositional logic program.

Recall that a *propositional logic program* is a finite set of *rules* of the form

$$a_0 \leftarrow a_1, \ldots, a_k, \mathit{not}\ a_{k+1}, \ldots, \mathit{not}\ a_m, \tag{7}$$

where a_0 is a propositional atom or symbol $\bot$; $a_1, \ldots, a_n$ are propositional atoms. For a rule r of the form (7), by r^{cl} we denote a clause

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_k \vee a_{k+1} \vee \cdots \vee a_m \tag{8}$$

when a_0 is an atom; and a clause

$$\neg a_1 \vee \cdots \vee \neg a_k \vee a_{k+1} \vee \cdots \vee a_m, \tag{9}$$

when a_0 is $\bot$. For a program Π, by Π^{cl} we denote a CNF formula composed of the respective clauses r^{cl} for rules r in Π. For example let Π stand for program

$$\begin{array}{l} p \\ r \leftarrow p, q \end{array} \tag{10}$$

then Π^{cl} follows

$$\begin{array}{ll} p & \wedge \\ r \vee \neg p \vee \neg q. & \end{array} \tag{11}$$

For a program Π, by σ_Π we denote the set of atoms occurring in it. We call σ_Π a program's signature. For a program Π, we call an interpretation M over σ_Π a *classical model* of Π if it is a model of Π^{cl}. For example, program (10) has three classical models $\{p, \neg q, \neg r\}$, $\{p, \neg q, r\}$, and $\{p, q, r\}$. In a sense, a concept of a classical model generalizes the definition of what does it mean for a set of atoms to satisfy a definite program to arbitrary programs.

A set U of atoms occurring in a propositional program Π is *unfounded* on a consistent set M of literals with respect to Π if for every atom $a \in U$ the following condition holds: for every rule in Π of the form

$$a \leftarrow a_1, \ldots, a_k, \mathit{not}\ a_{k+1}, \ldots, \mathit{not}\ a_m$$

(note that a is the head atom in this rule) the property below holds:

- either $M \cap \{\neg a_1, \ldots, \neg a_k, a_{k+1}, \ldots, a_m\} \neq \emptyset$
- or $U \cap \{a_1, \ldots, a_k\} \neq \emptyset$

For instance, set $\{r\}$ is unfounded on set $\{p, \neg q, r\}$ with respect to program (10), while set $\{q\}$ is unfounded on any set of literals with respect to program (10). We may also note that any set of atoms containing atom p will not be unfounded on any set of literals with respect to program (10) (this fact is explained by the presence of fact $p.$ in the program). It is easy to see that the $\emptyset$ of atoms is unfounded on any set of literals with respect to any program.

For a set M of literals, by M^+ we denote the set composed of all the literals that occur without classical negation in M. E.g., $\{p, q, \neg r\}^+ = \{p, q\}$.

We now state a formal result that relates the notions of an unfounded set and answer sets. This result is crucial for understanding key inference rules used in propagators of modern answer set solvers.

Proposition 2. *For a program Π and a set M of literals over σ_Π, M^+ is an answer set of Π if and only if M is a classical model of Π and no non-empty subset of M^+ is an unfounded set on M with respect to Π.*

This proposition gives an alternative characterization of an answer set. I.e., we may bypass the reference to a reduct in our argument that a set of atoms is an answer set. It is sufficient to verify that (i) this set of atoms corresponds to a classical model of a program and (ii) no non-empty subset of this set is unfounded. For example, this proposition asserts that

- classical models of program (10) are the only interpretations that may correspond to answer sets of (10)
- sets $\{p, \neg q, \neg r\}$, $\{p, \neg q, r\}$, $\{p, q, r\}$ of literals are the classical models of program (10). Thus, sets $\{p, \neg q, \neg r\}^+ = \{p\}$, $\{p, \neg q, r\}^+ = \{p, r\}$, $\{p, q, r\}^+ = \{p, q, r\}$ of atoms form the candidates for being answer sets,

- sets $\{p, r\}$ and $\{p, q, r\}$ are not answer sets of the program due to unfounded sets $\{r\}$ and $\{q\}$ respectively. Set $\{p\}$ is an answer set (since the only nonempty subset of it, namely, $\{p\}$, is not an unfounded set on $\{p, \neg q, \neg r\}$ with respect to program (10)).

We define the transition graph $aset_\Pi$ for a program Π as follows. The set of nodes of the graph $aset_\Pi$ consists of the states relative to atoms occurring in Π. There are five transition rules that characterize the edges of $aset_\Pi$. The transition rules *Unit Propagate*, *Decide*, *Fail*, *Backtrack* of the graph $\text{DP}_{\Pi^{cl}}$, and the transition rule

$$\textit{Unfounded:} \quad M \;\Rightarrow\; M\ \neg a \ \text{ if } \begin{cases} a \in U \text{ for a set } U \text{ unfounded on } M \\ \text{with respect to } \Pi. \end{cases}$$

The graph $aset_\Pi$ can be used for deciding whether a logic program has answer sets:

Proposition 3. *For any program* Π*,*

(a) graph $aset_\Pi$ *is finite and acyclic,*

(b) for any terminal state M *of* $aset_\Pi$ *other than FailState,* M^+ *is an answer set of* Π*,*

(c) FailState is reachable from $\emptyset$ *in* $aset_\Pi$ *if and only if* Π *has no answer sets.*

A Peek at Important Enhancements of ASP (and SAT) solvers The key difference of system CLINGO from the SMODELS algorithm that we presented lays in implementation of such advanced solving techniques as *learning and forgetting*, *backjumping* and *restarts*. Below we provide some intuitions behind these.

The *learning* technique allows a solver to extend its knowledge base (that originally is composed of a given program) by additional constraints so that certain inferences become readily available in the later states of search via propagation rules (eliminating the need for intermediate applications of decide rules). The *forgetting* allows the solver to make the learning process dynamic so that sometimes learned constraints are forgotten/removed to eliminate the chance of solver's knowledge base becoming of a prohibitive size.

Backjumping enhances backtracking mechanism by allowing to identify the decision level different from the last one that is safe to jump to so that (i) no solution is lost and (ii) part of the search space is escaped.

Restarting allows a solver to drop currently searched path and start over again with a hope to make better choices on a new path that lead to a solution quicker.

Problem 4. *Let* Π_1 *be a program*

$$\begin{array}{l} r. \\ p \leftarrow \ not\ q, r \\ q \leftarrow \ not\ p, r \end{array}$$

(a) List all classical models of Π_1*.*
(b) List all unfounded sets on set $\{r, p, q\}$ *with respect to program* Π_1*.*
(c) List all unfounded sets on set $\{r, \neg p, \neg q\}$ *with respect to program* Π_1*.*
(d) List all unfounded sets on set $\{r, p, \neg q\}$ *with respect to program* Π_1*.*
(e) List all answer sets of Π_1*.*
(f) List some five states in graph $aset_{\Pi_1}$*.*
(g) List some path in $aset_{\Pi_1}$ *from* $\emptyset$ *to state* $r\neg q^{\Delta}p$*. Think of another possible path in this graph from* $\emptyset$ *to the same state* $r\neg q^{\Delta}p$*. List that path. In both cases annotate all the transitions/edges in your path by the names of the respective rules.*

(h) Is state $r\neg q^{\Delta}p$ terminal in the graph $aset_{\Pi_1}$? If so what can you conclude about program Π_1 and state $r\neg q^{\Delta}p$ given Proposition 3.

Acknowledgments

Parts of this handout follow

- the lecture notes on Logic-based AI course, UT, Spring 2011[1] by Vladimir Lifschitz.
- *What is answer set programming to propositional satisfiability*, Yuliya Lierler, Constraints, July 2017, Volume 22, Issue 3, pp 307337 available at `https://link.springer.com/article/10.1007/s10601-016-9257-7`.

In class discussions of Fall 2020 course on *Introduction to AI* at UNO contributed to several examples listed in these notes.

[1] `http://www.cs.utexas.edu/~vl/teaching/lbai`