Time-Dependent Programming In the Answer Set Programming Paradigm

logostheorist

April 24, 2017

Time-Dependent Programming In The Answer Set Programming

The Answer Set Programming Paradigm

- ASP is a form of non-monotonic reasoning based on stable-state semantics;

The Answer Set Programming Paradigm

- `ASP` is a form of non-monotonic reasoning based on stable-state semantics;
- `ASP` paradigm amounts to figuring out <u>how to state</u> a problem as opposed to <u>declaring how to solve</u> a problem

The Answer Set Programming Paradigm

- ASP is a form of non-monotonic reasoning based on stable-state semantics;
- ASP paradigm amounts to figuring out how to state a problem as opposed to declaring how to solve a problem
  1. Encode a problem I as a logic program P such that solutions of I are models of P

The Answer Set Programming Paradigm

- `ASP` is a form of non-monotonic reasoning based on stable-state semantics;
- `ASP` paradigm amounts to figuring out how to state a problem as opposed to declaring how to solve a problem
  1. Encode a problem I as a logic program P such that solutions of I are models of \$P\$
  2. Compute a model $\mathcal{M}$ of P using an Answer Set Solver such as `dlv` or `Prolog`

The Answer Set Programming Paradigm

- `ASP` is a form of non-monotonic reasoning based on stable-state semantics;
- `ASP` paradigm amounts to figuring out how to state a problem as opposed to declaring how to solve a problem
  1. Encode a problem I as a logic program P such that solutions of I are models of \$P\$
  2. Compute a model $\mathcal{M}$ of P using an Answer Set Solver such as `dlv` or `Prolog`
  3. Extract a solution for I from \$M\$

The Answer Set Programming Paradigm

- `ASP` is a form of non-monotonic reasoning based on stable-state semantics;
- `ASP` paradigm amounts to figuring out how to state a problem as opposed to declaring how to solve a problem
  1. Encode a problem I as a logic program P such that solutions of I are models of P
  2. Compute a model $\mathcal{M}$ of P using an Answer Set Solver such as `dlv` or `Prolog`
  3. Extract a solution for I from M
- A positive logic program P consists of a finite set of clauses called rules consisting of atoms a, b_i in a first order language of the form

$$a \leftarrow b_1, \ldots, b_m$$

The Answer Set Programming Paradigm

- ASP is a form of non-monotonic reasoning based on stable-state semantics;
- ASP paradigm amounts to figuring out how to state a problem as opposed to declaring how to solve a problem
  1. Encode a problem I as a logic program P such that solutions of I are models of P
  2. Compute a model $\mathcal{M}$ of P using an Answer Set Solver such as `dlv` or `Prolog`
  3. Extract a solution for I from M
- A positive logic program P consists of a finite set of clauses called rules consisting of atoms a, b_i in a first order language of the form

$$a \leftarrow b_1, \ldots, b_m$$

1. (Rules)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n \vee \varphi)} \ (n \in \mathbb{N}, \varphi_1, \ldots, \varphi_n, \varphi \text{ atomic})$$

Proof Calculi of Universal Horn Formulae

1. (Rules)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n \vee \varphi)} \; (n \in \mathbb{N}, \varphi_1, \ldots, \varphi_n, \varphi \text{ atomic})$$

   As in classical logic,
   $\varphi \leftarrow \varphi_0, \ldots, \varphi_n \equiv \varphi \vee \neg\varphi_0 \vee \neg\varphi_1 \vee \cdots \vee \neg\varphi_n$.

Proof Calculi of Universal Horn Formulae

1. (Rules)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n \vee \varphi)}\ (n \in \mathbb{N}, \varphi_1, \ldots, \varphi_n, \varphi \text{ atomic})$$

As in classical logic,

$\varphi \leftarrow \varphi_0, \ldots, \varphi_n \equiv \varphi \vee \neg\varphi_0 \vee \neg\varphi_1 \vee \cdots \vee \neg\varphi_n$.

2. (Goals)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n)}\ (n \in \mathbb{N}, \varphi_0, \ldots, \varphi_n \text{ atomic})$$

Proof Calculi of Universal Horn Formulae

1. (Rules)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n \vee \varphi)} \; (n \in \mathbb{N}, \varphi_1, \ldots, \varphi_n, \varphi \text{ atomic})$$

   As in classical logic,

   $\varphi \leftarrow \varphi_0, \ldots, \varphi_n \equiv \varphi \vee \neg\varphi_0 \vee \neg\varphi_1 \vee \cdots \vee \neg\varphi_n$.

2. (Goals)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n)} \; (n \in \mathbb{N}, \varphi_0, \ldots, \varphi_n \text{ atomic})$$

3. (Conjunction)

$$\frac{\varphi \qquad \psi}{(\varphi \wedge \psi)}$$

Proof Calculi of Universal Horn Formulae

1. (Rules)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n \vee \varphi)} \ (n \in \mathbb{N}, \varphi_1, \ldots, \varphi_n, \varphi \text{ atomic})$$

   As in classical logic,
   $\varphi \leftarrow \varphi_0, \ldots, \varphi_n \equiv \varphi \vee \neg\varphi_0 \vee \neg\varphi_1 \vee \cdots \vee \neg\varphi_n$.

2. (Goals)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n)} \ (n \in \mathbb{N}, \varphi_0, \ldots, \varphi_n \text{ atomic})$$

3. (Conjunction)

$$\frac{\varphi \qquad \psi}{(\varphi \wedge \psi)}$$

4. (Universal Extension)

$$\frac{\varphi}{\forall x, \varphi}$$

1. (Rules)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n \vee \varphi)}\ (n \in \mathbb{N}, \varphi_1, \ldots, \varphi_n, \varphi \text{ atomic})$$

   As in classical logic,
   $\varphi \leftarrow \varphi_0, \ldots, \varphi_n \equiv \varphi \vee \neg\varphi_0 \vee \neg\varphi_1 \vee \cdots \vee \neg\varphi_n$.

2. (Goals)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n)}\ (n \in \mathbb{N}, \varphi_0, \ldots, \varphi_n \text{ atomic})$$

3. (Conjunction)

$$\frac{\varphi \qquad \psi}{(\varphi \wedge \psi)}$$

4. (Universal Extension)

$$\frac{\varphi}{\forall x, \varphi}$$

5. **(Selective Linear Definite (SLD) resolution)**

Resolution Of Definite Clauses

- We resolve definite clauses by way of the most general unifier (mgu) θ (e.g. a substitution that makes two or more atoms identical)

Resolution Of Definite Clauses

- We resolve definite clauses by way of the most general unifier (mgu) θ (e.g. a substitution that makes two or more atoms identical)
- In `Prolog`, we produce a proof tree using SLDR and resolve via DFS.

Resolution Of Definite Clauses

- We resolve definite clauses by way of the most general unifier (mgu) θ (e.g. a substitution that makes two or more atoms identical)
- In `Prolog`, we produce a proof tree using SLDR and resolve via DFS.
- Each node is a stack of negative literals to be resolved

Resolution Of Definite Clauses

- We resolve definite clauses by way of the most general unifier (mgu) θ (e.g. a substitution that makes two or more atoms identical)
- In `Prolog`, we produce a proof tree using SLDR and resolve via DFS.
- Each node is a stack of negative literals to be resolved
- Prolog resolves the top literal from the stack against the head literal of every clause in the program, which are potentially complementary unifiable, and are searched top to bottom

Resolution Of Definite Clauses

- We resolve definite clauses by way of the most general unifier (mgu) θ (e.g. a substitution that makes two or more atoms identical)
- In `Prolog`, we produce a proof tree using SLDR and resolve via DFS.
- Each node is a stack of negative literals to be resolved
- Prolog resolves the top literal from the stack against the head literal of every clause in the program, which are potentially complementary unifiable, and are searched top to bottom
- If the top literal is unifiable by mgu θ, with the head of clause C, we pop L from the stack, and push $body(C)$ onto the stack, substituting θ to all literals

Resolution Of Definite Clauses

- We resolve definite clauses by way of the most general unifier (mgu) θ (e.g. a substitution that makes two or more atoms identical)
- In `Prolog`, we produce a proof tree using SLDR and resolve via DFS.
- Each node is a stack of negative literals to be resolved
- Prolog resolves the top literal from the stack against the head literal of every clause in the program, which are potentially complementary unifiable, and are searched top to bottom
- If the top literal is unifiable by mgu θ, with the head of clause C, we pop L from the stack, and push $body(C)$ onto the stack, substituting θ to all literals
- If no clause is unifiable with the L, the search backtracks to the last point (whence DFS)

Resolution Of Definite Clauses

- We resolve definite clauses by way of the most general unifier (mgu) θ (e.g. a substitution that makes two or more atoms identical)
- In `Prolog`, we produce a proof tree using SLDR and resolve via DFS.
- Each node is a stack of negative literals to be resolved
- Prolog resolves the top literal from the stack against the head literal of every clause in the program, which are potentially complementary unifiable, and are searched top to bottom
- If the top literal is unifiable by mgu θ, with the head of clause C, we pop L from the stack, and push $body(C)$ onto the stack, substituting θ to all literals
- If no clause is unifiable with the L, the search backtracks to the last point (whence DFS)
- If the stack is emptied, we derive nil, whence we return `true`; else, the stack is not emptied after our search, whence we return `false`

Model Semantics Of P

Let P be a (positive) logic program.

- A Herbrand universe of P, denoted by $\mathrm{HU}(P)$, consists of the set of all terms formed by the language $\mathcal{L}_P$.

Model Semantics Of P

Let P be a (positive) logic program.

- A Herbrand universe of P, denoted by HU(P), consists of the set of all terms formed by the language $\mathcal{L}_P$.
- A Herbrand base of P, denoted by HB(P), consists of all ground atoms formed from predicates in P and terms in HU(P), such that an interpretation over HU(P) is simply a subset $I \subseteq$ HB(P) may be understood a set of of grounds atoms true in a given *scenario*.

Model Semantics Of P

Let P be a (positive) logic program.

- A Herbrand universe of P, denoted by $\text{HU}(P)$, consists of the set of all terms formed by the language $\mathcal{L}_P$.
- A Herbrand base of P, denoted by $\text{HB}(P)$, consists of all ground atoms formed from predicates in P and terms in $\text{HU}(P)$, such that an interpretation over $\text{HU}(P)$ is simply a subset $I \subseteq \text{HB}(P)$ may be understood a set of of grounds atoms true in a given *scenario*.
- An interpretation M may be a model of
  1. a ground clause $C \equiv a \leftarrow b_1, \ldots, b_n$ if $\{b_1, \ldots, b_n\} \not\subseteq M$ or $a \in M$, denoted by $M \models C$;

Model Semantics Of P

Let P be a (positive) logic program.

- A Herbrand universe of P, denoted by $\text{HU}(P)$, consists of the set of all terms formed by the language $\mathcal{L}_P$.
- A Herbrand base of P, denoted by $\text{HB}(P)$, consists of all ground atoms formed from predicates in P and terms in $\text{HU}(P)$, such that an interpretation over $\text{HU}(P)$ is simply a subset $I \subseteq \text{HB}(P)$ may be understood a set of of grounds atoms true in a given *scenario*.
- An interpretation M may be a model of
  1. a ground clause $C \equiv a \leftarrow b_1, \ldots, b_n$ if $\{b_1, \ldots, b_n\} \not\subseteq M$ or $a \in M$, denoted by $M \models C$;
  2. a clause C if $M \models C'$ for all $C' \in grnd(C)$, the set of all ground instances of C appearing in $\text{HU}(P)$;

Model Semantics Of P

Let P be a (positive) logic program.

- A Herbrand universe of P, denoted by HU(P), consists of the set of all terms formed by the language $\mathcal{L}_P$.
- A Herbrand base of P, denoted by HB(P), consists of all ground atoms formed from predicates in P and terms in HU(P), such that an interpretation over HU(P) is simply a subset $I \subseteq$ HB(P) may be understood a set of of grounds atoms true in a given *scenario*.
- An interpretation M may be a model of
  1. a ground clause $C \equiv a \leftarrow b_1, \ldots, b_n$ if $\{b_1, \ldots, b_n\} \not\subseteq M$ or $a \in M$, denoted by $M \models C$;
  2. a clause C if $M \models C'$ for all $C' \in grnd(C)$, the set of all ground instances of C appearing in HU(P);
  3. a program P if $M \models C$ for all clauses $C \in P$.

Model Semantics Of P

Let P be a (positive) logic program.

- A Herbrand universe of P, denoted by $\mathrm{HU}(P)$, consists of the set of all terms formed by the language $\mathcal{L}_P$.
- A Herbrand base of P, denoted by $\mathrm{HB}(P)$, consists of all ground atoms formed from predicates in P and terms in $\mathrm{HU}(P)$, such that an interpretation over $\mathrm{HU}(P)$ is simply a subset $I \subseteq \mathrm{HB}(P)$ may be understood a set of of grounds atoms true in a given *scenario*.
- An interpretation M may be a model of
  1. a ground clause $C \equiv a \leftarrow b_1, \ldots, b_n$ if $\{b_1, \ldots, b_n\} \not\subseteq M$ or $a \in M$, denoted by $M \models C$;
  2. a clause C if $M \models C'$ for all $C' \in grnd(C)$, the set of all ground instances of C appearing in $\mathrm{HU}(P)$;
  3. a program P if $M \models C$ for all clauses $C \in P$.
- A model M of P is minimal if there is no model N of P such that $N \subsetneq M$.

Model Semantics Of P

Let P be a (positive) logic program.

- A Herbrand universe of P, denoted by HU(P), consists of the set of all terms formed by the language $\mathcal{L}_P$.
- A Herbrand base of P, denoted by HB(P), consists of all ground atoms formed from predicates in P and terms in HU(P), such that an interpretation over HU(P) is simply a subset $I \subseteq$ HB(P) may be understood a set of of grounds atoms true in a given *scenario*.
- An interpretation M may be a model of
  1. a ground clause $C \equiv a \leftarrow b_1, \ldots, b_n$ if $\{b_1, \ldots, b_n\} \not\subseteq M$ or $a \in M$, denoted by $M \models C$;
  2. a clause C if $M \models C'$ for all $C' \in grnd(C)$, the set of all ground instances of C appearing in HU(P);
  3. a program P if $M \models C$ for all clauses $C \in P$.
- A model M of P is minimal if there is no model N of P such that $N \subsetneq M$. The answer set of P is the minimal model of P.

Minimal Model Computation and Negation

- We iteratively compute $\mathrm{LM}(P)$ by the immediate consequence operator, where $T_P : 2^{\mathrm{HB}(P)} \rightarrow 2^{\mathrm{HB}(P)}$ is defined by

$$I \mapsto \{a \mid \exists(a \leftarrow b_1, \ldots, b_n) \in Gnd(P), \{b_1, \ldots, b_m\} \subseteq I\}$$

Minimal Model Computation and Negation

- We iteratively compute $\mathrm{LM}(P)$ by the immediate consequence operator, where $T_P : 2^{\mathrm{HB}(P)} \to 2^{\mathrm{HB}(P)}$ is defined by

$$I \mapsto \{a \mid \exists(a \leftarrow b_1, \ldots, b_n) \in Gnd(P), \{b_1, \ldots, b_m\} \subseteq I\}$$

under T_P, for all founded atoms in the body of a rule r, then a will be founded.

Minimal Model Computation and Negation

- We iteratively compute $\mathrm{LM}(P)$ by the immediate consequence operator, where $T_P : 2^{\mathrm{HB}(P)} \to 2^{\mathrm{HB}(P)}$ is defined by

$$I \mapsto \{a \mid \exists(a \leftarrow b_1, \ldots, b_n) \in Gnd(P), \{b_1, \ldots, b_m\} \subseteq I\}$$

  under T_P, for all founded atoms in the body of a rule r, then a will be founded.
- We extend positive logic programs to normal logic programs by adding a notion of negation different from negation in classical logic, interpreted as Negation as failure with falsity denoted by *fail*, and where one considers $\texttt{not}a(\cdot)$ to be true if no corresponding positive literal $a(\cdot)$ can be finitely proved through SLD resolution.

Minimal Model Computation and Negation

- We iteratively compute $\mathrm{LM}(P)$ by the immediate consequence operator, where $T_P : 2^{\mathrm{HB}(P)} \to 2^{\mathrm{HB}(P)}$ is defined by

$$I \mapsto \{a \mid \exists (a \leftarrow b_1, \ldots, b_n) \in Gnd(P), \{b_1, \ldots, b_m\} \subseteq I\}$$

  under T_P, for all founded atoms in the body of a rule r, then a will be founded.

- We extend positive logic programs to normal logic programs by adding a notion of negation different from negation in classical logic, interpreted as Negation as failure with falsity denoted by *fail*, and where one considers $\mathtt{not}\,a(\cdot)$ to be true if no corresponding positive literal $a(\cdot)$ can be finitely proved through SLD resolution.

- An interpretation I of P with naf is an answer set if and only if I is the reduct program $P^I := \{head(r) \leftarrow pos(r) \mid r \in P, I \cap neg(r) = \emptyset\}$

Deciding whether a given program P has a stable model is $NP - complete$

- This amounts to guessing a stable candidate M, checking in polynomial time if M is stable by verifying that the set of unfounded atoms in M is empty, where an unfounded atom a is the head of some rule r such that either an atom b appears as a positive literal in the body of r which is such that either $b \notin M$ or b is also unfounded, or b appears as a negative literal in the body of r such that $b \in M$.

Deciding whether a given program P has a stable model is $NP - complete$

- This amounts to guessing a stable candidate M, checking in polynomial time if M is stable by verifying that the set of unfounded atoms in M is empty, where an unfounded atom a is the head of some rule r such that either an atom b appears as a positive literal in the body of r which is such that either $b \notin M$ or b is also unfounded, or b appears as a negative literal in the body of r such that $b \in M$.
- Introducing functions can make this undecidable, as we may have models of infinite size. Consider the program F:

$$p(a)$$

$$p(f(X)) \leftarrow p(X)$$

$Gnd(F) = \{p(a), p(f(a)) \leftarrow p(a), p(f(f(a))) \leftarrow p(f(a)), \ldots\}$ is infinite, and is the unique stable model. For non-ground programs with function symbols, this problem becomes as difficult as the Halting program.

Example: 3 Coloring

- We can consider the ASP approach to the problem of computing legal 3-colorings of a graph $G = (V, E)$.

Example: 3 Coloring

- We can consider the ASP approach to the problem of computing legal 3-colorings of a graph $G = (V, E)$.
- We store the facts of our graph as $node(n)$ for each $n \in V$ and $edge(n, m)$ for each $(n, m) \in E$.

- We can consider the ASP approach to the problem of computing legal 3-colorings of a graph $G = (V, E)$.
- We store the facts of our graph as $node(n)$ for each $n \in V$ and $edge(n, m)$ for each $(n, m) \in E$.
- The general specification for solutions is then

$$red(X) \leftarrow node(X), \texttt{not}\, green(X), \texttt{not}\, blue(X)$$

$$green(X) \leftarrow node(X), \texttt{not}\, blue(X), \texttt{not}\, red(X)$$

$$blue(X) \leftarrow node(X), \texttt{not}\, red(X), \texttt{not}\, green(X)$$

with a single disjunctive rule

$$blue(X) \vee red(X) \vee green(x) \leftarrow node(X)$$

Example: 3 Coloring

- We can consider the ASP approach to the problem of computing legal 3-colorings of a graph $G = (V, E)$.
- We store the facts of our graph as $node(n)$ for each $n \in V$ and $edge(n, m)$ for each $(n, m) \in E$.
- The general specification for solutions is then

$$red(X) \leftarrow node(X), \mathtt{not}\, green(X), \mathtt{not}\, blue(X)$$

$$green(X) \leftarrow node(X), \mathtt{not}\, blue(X), \mathtt{not}\, red(X)$$

$$blue(X) \leftarrow node(X), \mathtt{not}\, red(X), \mathtt{not}\, green(X)$$

with a single disjunctive rule

$$blue(X) \vee red(X) \vee green(x) \leftarrow node(X)$$

- The Answer Sets will correspond to all legal 3-colorings of G.

Time Dependent Programs

- A time-dependent program \$$\langle$ P, $\tau\rangle$ \$ over σ consists of P, an answer set program over σ, and $\tau \subseteq \pi$ is a set of time dependent predicates.

Time Dependent Programs

- A time-dependent program \$$\langle$ P, $\tau\rangle$ \$ over σ consists of P, an answer set program over σ, and $\tau \subseteq \pi$ is a set of time dependent predicates.
- A t-grounding of a time-dependent literal l, denoted by $Gnd(l)_t$, is either l if $l \in Lit(\mathbf{P}) \backslash \mathcal{F}_{\mathbf{P}}$, and otherwise, the variable in $t_{arg}(l)$ is replaced by t.

Time Dependent Programs

- A time-dependent program \$$\langle$ P, $\tau\rangle$ \$ over σ consists of P, an answer set program over σ, and $\tau \subseteq \pi$ is a set of time dependent predicates.
- A t-grounding of a time-dependent literal l, denoted by $Gnd(l)_t$, is either l if $l \in Lit(\mathbf{P}) \backslash \mathcal{F}_{\mathbf{P}}$, and otherwise, the variable in $t_{arg}(l)$ is replaced by t.The t-grounding of the literals L is $Gnd(L)_t = \bigcup\limits_{l \in L} Gnd(l)_t$.

Time Dependent Programs

- A time-dependent program \$$\langle$ P, $\tau\rangle$ \$ over σ consists of P, an answer set program over σ, and $\tau \subseteq \pi$ is a set of time dependent predicates.
- A t-grounding of a time-dependent literal l, denoted by $Gnd(l)_t$, is either l if $l \in Lit(\mathbf{P}) \backslash \mathcal{F}_{\mathbf{P}}$, and otherwise, the variable in $t_{arg}(l)$ is replaced by t.The t-grounding of the literals L is $Gnd(L)_t = \bigcup\limits_{l \in L} Gnd(l)_t$.
- The t-grounding of a rule is \$Gnd(r)$_{\mathrm{t}}$=Gnd(head(r)$_{\mathrm{t}}$)$\leftarrow$ Gnd(pos(r))$_{\mathrm{t}}$, not Gnd(neg(r))$_{\mathrm{t}}$\$

Time Dependent Programs

- A time-dependent program \$$\langle$ P, $\tau\rangle$ \$ over σ consists of P, an answer set program over σ, and $\tau \subseteq \pi$ is a set of time dependent predicates.
- A t-grounding of a time-dependent literal l, denoted by $Gnd(l)_t$, is either l if $l \in Lit(\mathbf{P}) \backslash \mathcal{F}_{\mathbf{P}}$, and otherwise, the variable in $t_{arg}(l)$ is replaced by t.The t-grounding of the literals L is $Gnd(L)_t = \bigcup\limits_{l \in L} Gnd(l)_t$.
- The t-grounding of a rule is \$Gnd(r)$_{\mathrm{t}}$=Gnd(head(r)$_{\mathrm{t}}$)$\leftarrow$ Gnd(pos(r))$_{\mathrm{t}}$, not Gnd(neg(r))$_{\mathrm{t}}$\$
- The t-grounding of **P** is

$$Gnd(\mathbf{P})_{t_{max}} = Gnd(\{Gnd(r)_{t'} \mid r \in P, t' \in \mathbb{N}, t' \leq t_{max}\})$$

Example of a Time Dependent Program

```
time(0...M)
q@T :- p@(T-1),time(T),T(T-1)
v@T :- q@(T-1), not w@T,time(T),time(T-1)
q@T :- not v@T,r(X), time(T), time(T-1)
p@T :- time(T)
r(start).
```

Example of a Time Dependent Program

```
time(0...M)
q@T :- p@(T-1),time(T),T(T-1)
v@T :- q@(T-1), not w@T,time(T),time(T-1)
q@T :- not v@T,r(X), time(T), time(T-1)
p@T :- time(T)
r(start).
```

- This program depends on the time boundary M, and grows exponentially with M

Example of a Time Dependent Program

```
time(0...M)
q@T :- p@(T-1),time(T),T(T-1)
v@T :- q@(T-1), not w@T,time(T),time(T-1)
q@T :- not v@T,r(X), time(T), time(T-1)
p@T :- time(T)
r(start).
```

- This program depends on the time boundary M, and grows exponentially with M
- Finding steady states by brute force by estimating a time upper bound, grounding, and solving the program with the bound generally leads to a suboptimal solving time.

States and Trajectories

- Given **P**, and an answer set I of $Gnd(\mathbf{P})_{t_{max}}$, the state of I at t is $I^t := \{l \mid l \in I, t_{arg}(l) = t\}$,

States and Trajectories

- Given **P**, and an answer set I of $Gnd(\mathbf{P})_{t_{max}}$, the state of I at t is $I^t := \{l \mid l \in I, t_{arg}(l) = t\}$,i.e. the state of ground time-dependent literals in I grounded with T in the time argument.

States and Trajectories

- Given **P**, and an answer set I of $Gnd(\mathbf{P})_{t_{max}}$, the state of I at t is $I^t := \{l \mid l \in I, t_{arg}(l) = t\}$,i.e. the state of ground time-dependent literals in I grounded with T in the time argument.
- The trajectory of I is defined as \$T^I=$\langle$ I$^{0\dots}$ I$^{t_{max}}$$\rangle$\$

States and Trajectories

- Given **P**, and an answer set I of $Gnd(\mathbf{P})_{t_{max}}$, the state of I at t is $I^t := \{l \mid l \in I, t_{arg}(l) = t\}$,i.e. the state of ground time-dependent literals in I grounded with T in the time argument.
- The trajectory of I is defined as $T^{I} = \langle I^{0\cdots} I^{t_{max}} \rangle$
- We can find all steady states and cycles efficiently by transforming our program into a Markovian program and then solving for ground programs incrementally.

States and Trajectories

- Given **P**, and an answer set I of $Gnd(\mathbf{P})_{t_{max}}$, the state of I at t is $I^t := \{l \mid l \in I, t_{arg}(l) = t\}$,i.e. the state of ground time-dependent literals in I grounded with T in the time argument.
- The trajectory of I is defined as $T^I = \langle I^0 \dots I^{t_{max}} \rangle$
- We can find all steady states and cycles efficiently by transforming our program into a Markovian program and then solving for ground programs incrementally.

Markovian Programs

- A Markovian program is a time dependent program **P** if and only if for every $r \in P$ with $h(r) \in Lit(\mathbf{P})^\tau$ and $t \in \mathbb{N}$
  1. $t_{arg}(head(r)) \in \mathcal{C} \cup \mathcal{V}$
  2. for all $l \in Lit(r) \cap Lit(\mathbf{P})^\tau$, either $t_{arg}(Gnd(head(r))_t)$ or $$t_{\text{arg}}(\text{Gnd}(\text{head}(r))_t)=t_{\text{arg}}(\text{Gnd}(l)_t)+1$

- A Markovian program is a time dependent program $\mathbf{P}$ if and only if for every $r \in P$ with $h(r) \in Lit(\mathbf{P})^{\tau}$ and $t \in \mathbb{N}$
  1. $t_{arg}(head(r)) \in \mathcal{C} \cup \mathcal{V}$
  2. for all $l \in Lit(r) \cap Lit(\mathbf{P})^{\tau}$, either $t_{arg}(Gnd(head(r))_t)$ or $$t_{arg}(Gnd(head(r))_t)=t_{arg}(Gnd(l)_t)+1$
- Rules are divided into two subsets: those that describe temporal relationships $P^{\tau} = \{r \mid r \in P, (head(r) \cup Lit(r)) \cap Lit(\mathbf{P})^{\tau} \neq \emptyset\}$, and those that don't.

Partial Groundings and Reducts

- The partial temporal grounding of **P** at t is defined as $P_t = \{Gnd(r)_t \mid r \in P, head(r) \in Lit(\mathbf{P})^\tau, t_{arg}(Gnd(head(r))_t) = t\}$

- The partial temporal grounding of **P** at t is defined as $P_t = \{Gnd(r)_t \mid r \in P, head(r) \in Lit(\mathbf{P})^\tau, t_{arg}(Gnd(head(r))_t) = t\}$ i.e. the set of t-grounds rules whose head depends on t

Partial Groundings and Reducts

- The partial temporal grounding of **P** at t is defined as $P_t = \{Gnd(r)_t \mid r \in P, head(r) \in Lit(\mathbf{P})^\tau, t_{arg}(Gnd(head(r))_t) = t\}$ i.e. the set of t-grounds rules whose head depends on t
- A partial reduct of a ground program P wrt interpretation I, with $P_I = \{l \leftarrow . \mid l \in I\}$ and $head(P \backslash P_I =$ is defined as $R^I(P) := \{head(r) \leftarrow (pos(r) \backslash I, notneg(r). \mid r \in P \backslash P_I, neg(r) \cap I = \emptyset\}$

- THEOREM Let **P** be a Markovian program and let $Gnd(\mathbf{P})_{t_{max}}$ be a t_{max} grounding of **P** for $t_{max} \in \mathbb{N}$. Then the set of answer sets for $Gnd(\mathbf{P}_{t_{max}})$ is $\{ \bigcup_{i=-1}^{t_{max}} B^i \mid B^{-1} \in AS(P^e)$ and for$t \in [t_{max}], B^t \in AS(R^{B^{t-1} \cup B^{-1}}(P'_t)\}$ with $P'_t = Gnd(P_t \cup \{l \leftarrow . \mid l \in B^{t-1} \cup B^{-1}\})$.

Efficiency of Markov Programs

- THEOREM Let **P** be a Markovian program and let $Gnd(\mathbf{P})_{t_{max}}$ be a t_{max} grounding of **P** for $t_{max} \in \mathbb{N}$. Then the set of answer sets for $Gnd(\mathbf{P}_{t_{max}})$ is $\{ \bigcup_{i=-1}^{t_{max}} B^i \mid B^{-1} \in AS(P^e)$ and for$t \in [t_{max}], B^t \in AS(R^{B^{t-1} \cup B^{-1}}(P'_t)\}$ with $P'_t = Gnd(P_t \cup \{l \leftarrow . \mid l \in B^{t-1} \cup B^{-1}\})$.
  1. Solve P^e with environmental conditions, and initialize t=0
  2. Obtain partial groundings for t and states at t
  3. Update the list of trajectories with states found in 2.
  4. Increment t
  5. If any trajectories are not in a steady state or cycle, go to 2.