ASP ISMÉTLÉS

Az ASP-ismeretek felelevenítése

- A program: logikai állítások halmaza (esetleg paraméterekkel)
- A kérdés: kielégíthető-e az adott állításhalmaz
- A válasz: az adott állításhalmaz (egyetlen vagy összes) modellje, azaz az összes igaz elemi állítás halmaza

Egy program felépítése:

- szabályok
- formulák
- literálok
- atomi állítások (atomok)

ASP KÖVETKEZTETŐK - ÁTTEKINTÉS

Ismertebb implementációk

- `smodels` (`http://www.tcs.hut.fi/Software/smodels/`)
  - ez a legjobban elterjedt
  - C++ -ban írták
- `cmodels` (`http://www.cs.utexas.edu/users/tag/cmodels.html`)
  - az `smodels` -hez hasonlít, kevésbé általános, cserébe bizonyos esetekben gyorsabb tud lenni
  - C++ -ban írták
  - SAT megoldót használ
- `DLV` (`http://www.dbai.tuwien.ac.at/proj/dlv/`)
  - C++ -ban írták
- `noMoRe` (`http://www.cs.uni-potsdam.de/~linke/nomore/`)
  - ECLiPSe Prologban írták

SMODELS

smodels - az algoritmus nagy vonalakban

- Kimerítő keresést végzünk, visszalépésekkel
- Nyilvántartjuk literálok egy A halmazát, ami stabil modellek egy halmazának felel meg. Az A halmazban szereplő atomok részei, míg az itt szereplő negált atomok nem részei a stabil modelleknek. Következésképpen ha egy atom és a negáltja is megjelenik A-ban, akkor A az üres válaszhalmazt jelenti.
- Az algoritmus lépései
  - kiindulunk egy üres válaszhalmazból
  - ebbe a halmazba atomokat és negált atomokat veszünk fel
  - ellenőrizzük, hogy a halmazunk megfelel-e legalább egy stabil modellnek
  - ha nem, akkor visszalépünk, és eltávolítunk atomokat és negált atomokat a modellünkből, vagy a negációkat megváltoztatjuk
  - egyébként továbbmegyünk
  - az atomokat kifejtjük a rájuk vonatkozó szabályok szerint (pl. $a \leftarrow b, not\ c$)
- Lineáris tárigényt akarunk, amit ezzel a módszerrel el is érünk. Az algoritmus gyorsításához megfelelő heurisztikára van szükség, hogy hamar ki tudjuk választani azokat az ágakat, ahol gyorsan ellentmondásra jutunk

smodels - az algoritmus részletesebben

- Pszeudokód a következő oldalon, az eredeti dokumentációból
- Az algoritmus helyes működése bizonyított
- $expand(P, A)$: A szabályok alapján úgy bővíti A-t, hogy nem hoz létre választási pontot.
- Előretekintés: célszerűbb lenne elsőként azokkal az atomokkal foglalkozni, amelyek gyorsan konfliktushoz vezetnek.
  - A bővítéssel nem nyerünk nagyot, ha sok különböző stabil modellünk van (nem lehet determinisztikusan bővíteni)
  - Ha okosabbá (és ezzel lassabbá) tesszük a bővítőfüggvényt, akkor kis lépésszám-javulás mellett lelassítjuk a végrehajtást
  - Megoldás: gondolkodjunk előre!

smodels - az algoritmus részletesebben (folytatás)

```
function smodels(P, A)
  A := expand(P, A)
  A := lookahead(P, A)
  if conflict(P, A) then
    return false
  else if A covers Atoms(P) then
    return true {A+ is a stable model}
  else
    x := heuristic(P, A)
    if smodels(P, A ∪ {x}) then
      return true
    else
      return smodels(P, A ∪ {not(x)})
    end if
  end if.

function expand(P, A)
  repeat
    A' := A
    A := Atleast(P, A)
    A := A ∪ {not x | x ∈ Atoms(P) and x ∉ Atmost(P, A)}
  until A = A'
  return A.

function conflict(P, A)
  {Precondition: A = expand(P, A)}
  if A+ ∩ A− ≠ ∅ then
    return true
  else
    return false
  end if.

function lookahead(P, A)
  return A.
```

Algorithm 1: A decision procedure for the stable model semantics

17

smodels - előretekintés

- Pszeudokód a következő oldalon, az eredeti dokumentációból
- Megnézzük, hogy miként viselkedne az $smodels$ egy adott x literál választásakor (egy lépést előre megcsinálunk)
- Ha konfliktust találunk, akkor rögtön felvehetjük a modellünkbe x negáltját (v.ö. borotválás)
- Nagyon nagy költségnek tűnik, hogy végre kell hajtani legrosszabb esetben $|Atoms(P) - Atoms(A)|$ számú $expand$-ot, de a fejlesztők szerint váratlanul nagy gyorsulást tud produkálni. (Ehhez persze az is szükséges, hogy csak a literálok egy megfelelő heurisztikával előállított részhalmazára alkalmazzuk)

smodels - előretekintés (folytatás)

```
function lookahead(P, A)
  repeat
    A' := A
    A := lookahead_once(P, A)
  until A = A'
  return A.

function lookahead_once(P, A)
  B := Atoms(P) − Atoms(A)
  B := B ∪ not(B)
  while B ≠ ∅ do
    Take any literal x ∈ B
    A' := expand(P, A ∪ {x})
    B := B − A'
    if conflict(P, A') then
      return expand(P, A ∪ {not(x)})
    end if
  end while
  return A.
```

Algorithm 2: Looking ahead

and seeing how *smodels* behaves when it has chosen a literal. Observe that if the stable model S agrees with the partial model A but not with $A \cup \{x\}$ for some literal x, then S agrees with $A \cup \{not\,(x)\}$. Hence, we make progress as soon as we find a literal x that causes a conflict. That is to say, we can then enlarge A by $not\,(x)$. In addition, since $x' \in expand(P, A \cup \{x\})$ implies

$$\begin{aligned} expand(P, A \cup \{x'\}) &\subseteq expand\big(P, expand(P, A \cup \{x\})\big) \\ &= expand(P, A \cup \{x\}) \end{aligned}$$

due to the monotonicity of *expand*, it is not even necessary to examine all literals in $Atoms(P)$ not covered by A. As we test a literal x we can directly rule out all atoms in $expand(P, A \cup \{x\})$. We implement lookahead according to these observations by rewriting the function *lookahead* as in Algorithm 2. Possibly calling *expand* on the order of $|Atoms(P) - Atoms(A)|$ times for each new literal might not seem like a good idea, but in practice it has proven amazingly effective.

The idea that one can use the detection of conflicts to prune the search space has been presented in the context of propositional satisfiability checkers by Zabih and McAllester [67]. It is interesting to note that they concluded that the pruning method seems promising but causes too much overhead. Modern satisfiability checkers avoid the overhead by only employing lookahead on a small heuristically chosen subset of all atoms.

24

smodels - heurisztikák

- A szerzők szerint sok heurisztikának az a baja, hogy bizonyos esetekben egy nem létező megoldást akar mindenáron felkutatni. Ezért úgy döntöttek, hogy a mihamarabb konfliktust próbálnak találni, és nem egy konkrét megoldás felé próbálják terelni az algoritmust. Ez másképpen megfogalmazva azt jelenti, hogy a heurisztika a keresési tér méretét próbálja csökkenteni.
- Célszerű a következő elem választását egyszerre megejteni az előretekintéssel, mert így kevesebbet kell számolni.
- Úgy választunk atomot, hogy a következő lépésben a modellünket a lehető legnagyobb mértékben tudjuk bővíteni (lásd kód)
- Előretekintésnél mindig csak az első szűkítési lehetőségig dolgozunk, ezért jó lenne a literáljainkat egy megfelelő sorrendben végigjárni. Ez a sorrend a „legritkábban használt” (*least recently used*) lesz, nagy programot és kis literálhalmazt feltételezve.

smodels - specifi kus stabil modellek keresése

- Bizonyos esetekben nem egy tetszőleges, hanem bizonyos tulajdonságokkal rendelkező, esetleg az összes stabil modellre szükségünk van.
- Ha valamely modelleket ki akarunk zárni, akkor ennek triviálisan adódó módja a kofliktusvizsgálatba egy újabb vizsgálat beiktatása, ami csak a kritériumainkak megfelelő modelleket engedélyezi.
- Amennyiben az összes stabil modellre kíváncsiak vagyunk, a leálló ágba rakott „meghiúsulással" rávehetjük az algoritmust, hogy a legutolsó választási ponttól folytassa a keresést (v.ö. Prologban `fail`-lel minden megoldás keresése)
- Minimális stabil modellt úgy kereshetünk, hogy a literálokhoz pozitív súlyokat rendelve a modellünkben szereplő súlyokat összegezzük, és az előző értékkel összevetjük. Ha kisebb, akkor aktuális jelöltnek felvesszük, de mindenképpen keresünk tovább (lásd kód).

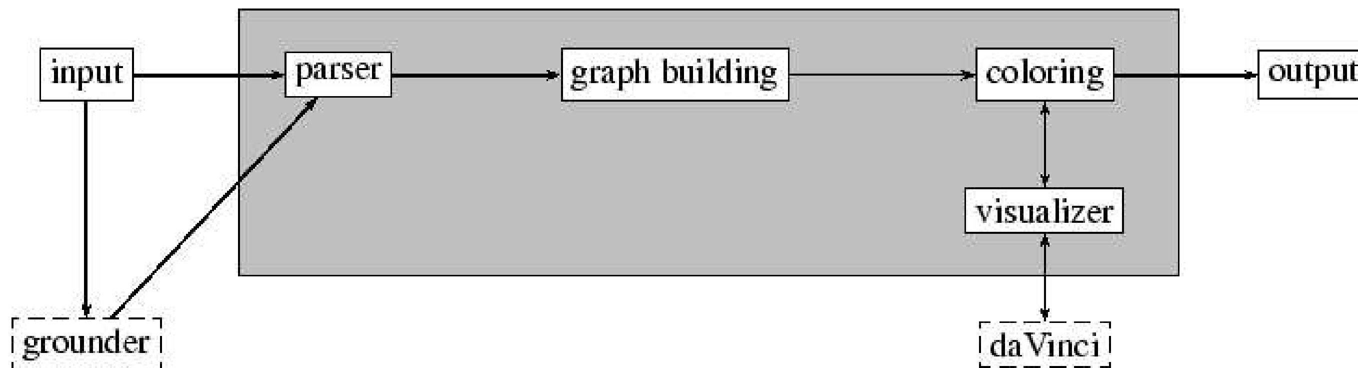
smodels - optimalizációk

- Legrosszabb esetben a keresési tér mérete $2^{|Atoms(P)|}$ lehet egy P programra. Ezen szeretnénk javítani (bár worst-case-re mindenképpen exponenciális marad, mivel a probléma NP-teljes)
- Egy kis gyorsítást jelenthet, ha azon atomokat kivesszük a szabályok törzséből, amik már biztosan nem lehetnek egy stabil modell részei.
- $atleast$ és $atmost$ megfelelő implementálása sokat segíthet - megfelelő körültekintéssel jó eséllyel el lehet kerülni a worst-case-t.
- A visszalépéseket a végrehajtott változtatásokként tartjuk nyilván, mert lineáris tárigényt szeretnénk elérni.
- Elképzelhető olyan helyzet, hogy két program unióját tekintjük a programunknak (az atomok diszjunktak). Az elsőnek sok megoldása van, míg a másodiknak egy sem. Jó lenne, ha észre tudnánk venni, hogy két független részprogram közül az egyiknek nincs megoldása, és emiatt felesleges tovább keresni. Ilyen esetben nem kell egyenként visszalépni, hanem több rekurziós szinttel is visszamehetünk: visszaugrás.

NOMORE

noMoRe áttekintése

- A Graph-Based System for Non-Monotonic Reasoning with Logic Programs under Answer Set Semantics
- A kereséseket a megfelelő frontend használatával grafikusan is meg lehet jeleníteni.
- Egy gráf-színező algoritmust használ a következtetéshez



DLV

DLV áttekintése

- Valós negációt is támogat (lásd előző ASP előadás)
- Többféle adatbázishoz is biztosít interfészt
- „Csak” binárist lehet tőlük letölteni, valamint az implementáció részleteiről és a belső működésről nem írnak túl sokat