Prolog: Generator de orare

- Responsabili: Andrei Olaru [mailto:mailto:cs@andreiolaru.ro], Teodor Szente [mailto:mailto:teodor98sz@gmail.com], Bogdan Neacșu [mailto:mailto:neacsubogdan95@gmail.com], Vlad Tălmaciu [mailto:mailto:vlad.talmaciu@gmail.com]

- Deadline soft: **20.05.2019**, apoi depunctare 0.5p/zi
- Deadline hard: *22.05.2019 ora 23.59*
- Data publicării: 09.05.2019
- Data ultimei modificări: 15.05.20189 changelog
- Data testerului: 16.05.2019
- Tema se va încărca pe **vmchecker**
- Forum tema 3 [https://acs.curs.pub.ro/2018/mod/forum/view.php?id=13366]
- vmchecker [https://vmchecker.cs.pub.ro/ui/#PP]

Descriere

Tema are ca scop implementarea unui generator de orare folosind paradigma logică.

Probleme de satisfacere a constrângerilor (CSP)

CSP reprezintă o clasă de probleme definite prin:

- O mulțime de variabile
- O mulțime de domenii pentru fiecare variabilă
- O mulțime de constrângeri

Astfel avem posibilitatea de a formaliza și studia în mod matematic orice problemă care poate fi rezolvată prin satisfacerea unor constrângeri.

Descrierea problemei

Problema de rezolvat este de a realiza un orar al activităților (ca orarul de la facultate). Programul creat va putea crea orarul pentru mai multe **probleme (*contexte*)** diferite. Fiecare problemă (context) este caracterizată de următoarele elemente:

- există niște **zile** în care se pot programa activități.
- în fiecare dintre aceste zile există un număr de **intervale** (ore) în care se pot programa activități. Într-un anumit *context*, mulțimea de intervale este aceeași pentru toate zilele, și toate intervalele au aceeași lungime, care este un număr întreg de ore.
    - un interval dintr-o anumită zi este numit un **slot** (sau un slot orar). E.g. slotul de luni 10-12, sau slotul de miercuri 14-16.
- există mai multe **săli** în care se pot programa activitățile.
- există mai multe **grupe** pentru care trebuie programate activități.
    - fiecare grupă poate avea o mulțime diferită de activități care trebuie programate.
- există mai multe **persoane** care pot ține activitățile.
    - Fiecare persoană poate ține o anumită submulțime din mulțimea totală de activități.
- există mai multe **activități**.
    - Fiecare activitate trebuie să aibă un număr de **instanțe** în fiecare săptămână.
    - Durata fiecărei instanțe este egală cu durata intervalelor din *context*.
    - Numărul de instanțe necesar va fi același pentru fiecare dintre grupele care au activitatea respectivă.
    - Fiecare instanță a unei activități se desfășoară:
        - Într-un anumit interval dintr-o anumită zi;
        - Într-o anumită sală;

        - Ținută cu o anumită grupă;
        - Ținută de o anumită persoană;
- **constrângerile** sunt asociate diverselor **entități** – o entitate este o grupă, o persoană, sau o sală. Avem mai multe tipuri de constrângeri:
    - constrângeri *fizice*: o entitate nu poate avea mai multe activități în același slot (o persoană nu poate ține două activități simultan; două activități nu se pot desfășura simultan în aceeași sală; o grupă nu poate avea două activități simultan).
    - constrângeri de *curiculă*: activitățile pot avea mai multe instanțe pe săptămână, dar poate exista un număr maxim de *instanțe în aceeași zi*.
    - constrângeri de *preferință*: orice entitate poate avea diverse *preferințe* pentru activitățile care îi sunt atribuite. Pentru fiecare dintre preferințe există un mod de calcul al **costului** corespunzător situației în care constrângerea este încălcată. Tipurile de constrângeri sunt detaliate mai jos.
        - e.g. o persoană preferă activitățile dimineața; o grupă preferă ca toate orele să fie în continuare; într-o sală se preferă ca să nu existe mai mult de un număr de ore pe zi; etc.

Descrierea variabilelor şi domeniilor de definiţie

Vom reprezenta mulțimea de variabile cât și domeniile de definiție printr-o listă (numită `Context`-ul problemei) care conține următorii **compuși** (un *compus* este identică, ca formă, cu un *atom*, dar este un termen (apare ca *argument* al unui atom)):

- `days(L)` – zilele săptămânii disponibile pentru activități. Fiecare element din listă este un nume de zi.
- `times(L)` – intervalele disponibile pentru activități
    - `L` este o listă de perechi, fiecare pereche conținând numele intervalului și durata acestuia.
- `groups(L)` – grupele pentru care trebuie programate activitățile. Fiecare element din listă este numele unei grupe.
- `staff(L)` – persoanele care pot ține ore.
    - `L` este o listă de perechi, fiecare pereche conținând numele persoanei și lista de activități pe care le poate ține:

```
L = [(Persoană1, [Materie1, Materie2]),

     (Persoană2, [Materie2, Materie3]),

     ....
```

Observați că în acest exemplu `Materie2` poate fi predată și de `Persoană1` și de `Persoană2`.

- `activities(L)` – activitățile care pot fi programate în orar.
    - `L` este o listă de perechi, fiecare pereche conținând numele activității și numărul de instanțe care trebuie programate în fiecare săptămână.

```
L = [(Materie1, 2), (Materie2, 2), (Materie3, 1)]
```

- `constraints(L)` – constrângerile pentru orar.
    - Fiecare element din listă este un compus `tip(entitate, argumente...)` având un număr oarecare de argumente (dependent de tip). Tipurile de constrângeri sunt detaliate mai jos.
    - O constrângere este asociată cu (primul argument este) o anumită **entitate** – o entitate este o grupă, o persoană, sau o sală;

Constrângeri şi costuri

- `max_instances(Course, N)` - numărul maxim de instanțe (sloturi) permis pentru materia dată **într-o zi, pentru o anumită grupă**.
    - Restricțiile de acest tip sunt obligatorii, iar încălcarea lor implică invaliditatea soluției.
- `max_hours(Entity, Hours, Cost)` – numărul maxim de ore permis pentru entitatea dată **într-o zi**.
    - restricția se referă la numărul de ore fizice (deci numărul de sloturi alocate înmulțit cu dimensiunea unui interval).
    - pentru fiecare oră în plus față de numărul de ore specificat se va adăuga costul `Cost` la costul total al soluției.
- `min_hours(Entity, Hours, Cost)` - la fel ca mai sus; pentru fiecare oră în minus față de numărul de ore specificat se va adăuga costul `Cost` la costul total al soluției.

- `continuous(Entity, Cost)` - - fiecare fereastră – spațiu între orele din aceeași zi – adaugă costul `Cost` la costul total al soluției.
- `interval(Entity, Start, End, Cost)` – intervalul preferat pentru activitățile unei entități. Costul pentru o activitate asociată cu entitatea dată, programată în intervalul `A-B` se va calcula astfel:
    - dacă intervalul este inclus în intervalul preferat (`Start <= A` și `B <= End`) atunci Costul este 0.
    - dacă `A < Start`, costul va fi `Cost * (Start - A)`
    - dacă `End < B`, costul va fi `Cost * (B - End)`

Cerință

Se cere realizarea, pentru un *Context* dat, a orarului de cost minim,

Un orar va fi reprezentat ca o listă de *sloturi*, fiecare slot fiind reprezentat printr-un compus

`slot(A, G, D, T, R, P)`, unde

- `A` este numele activității
- `G` este numele grupei
- `D` este numele zilei
- `T` este perechea (nume interval, durată)
- `R` este numele sălii
- `P` este numele persoanei

O **soluție** pentru o problemă va fi considerată **perechea** `(Context, Slots)`, unde `Context` este descrierea problemei care a fost dată (poate fi exact cea dată), iar `Slots` este o listă de sloturi.

Se cere să lucrați în fișierul `orar.pl`.

Predicatele care se cer implementate sunt următoarele:

- `schedule(+Context, -Sol)` – pentru contextul descris, întoarce o soluție care respectă constrângerile fizice și de curiculă.
    - `findall` pentru predicatul `schedule` va obține *toate* soluțiile corecte pentru orar, **fără duplicate**.
- `cost(+Sol, -Cost)` – pentru soluția dată (pereche descrierea problemei - listă sloturi), calculează costul implicat de constrângerile de preferință care au fost încălcate.
- `schedule_best(+Context, -Sol, -Cost)` – pentru contextul descris, întoarce soluția validă cu cel mai bun (cel mai mic) cost (sau una dintre ele, dacă există mai multe cu același cost)

Bonus

- implementați generarea orarului într-un mod eficient, care găsește cea mai bună soluție fără a genera complet toate soluțiile.

Exemplu

Să presupunem că materiile se desfășoară timp de o oră de la 8 la 12, în intervale de 1 oră, atunci intervalele vor fi: `[(8-9, 1), (9-10, 1), (10-11, 1), (11-12, 1)]`.

Există două săli `Sala1` și `Sala2` și două grupe `ClasaXA` și `ClasaXB`.

Exista doi profesori: `Popescu` care predă `matematică` și `Georgescu` care preda `romană`.

Fiecare clasă trebuie să facă `4` ore de matematică și `2` de romană.

Nu trebuie să existe ferestre și orice clasă nu trebuie să aibă mai mult de `3` ore pe zi.

Putem reprezenta această problemă în felul următor:

```
problem(scoala, [
```

```
  days([lu, ma, mi, jo, vi]),
  times([(8-9, 1), (9-10, 1), (10-11, 1), (11-12, 1)]),
  rooms([sala1, sala2]),
  groups([clasaAB, clasaXB]),
  staff([
      (popescu, [matematica]),
      (georgescu, [romana])
  ]),
  activities([(matematica, 4), (romana, 2)]),
  constraints([
      max_hours(clasaAB, 3, 1),
      max_hours(clasaXB, 3, 1),
      continuous(popescu, 3),
      continuous(clasaAB, 3),
      interval(georgescu, 8, 10, 2)
  ])
]).
```

Pentru această problemă, o soluție ar fi următoarea:

```
|                         |         clasaAB          |         clasaXB          |
|        lu:(8-9,1)       | matematica:popescu:sala1 |  romana:georgescu:sala2  |
|       lu:(9-10,1)       |  romana:georgescu:sala2  | matematica:popescu:sala1 |
|       lu:(10-11,1)      | matematica:popescu:sala1 |  romana:georgescu:sala2  |
|       lu:(11-12,1)      |  romana:georgescu:sala2  | matematica:popescu:sala1 |
|        ma:(8-9,1)       | matematica:popescu:sala1 |                          |
|       ma:(9-10,1)       |                          | matematica:popescu:sala1 |
|       ma:(10-11,1)      | matematica:popescu:sala1 |                          |
|       ma:(11-12,1)      |                          | matematica:popescu:sala1 |
|        mi:(8-9,1)       |                          |                          |
```

Costul acestei soluții se calculează astfel:

- ambele clase (grupe) au lunea 4 ore, care depășește preferința de 3. Astfel se adaugă un cost de 1 pentru fiecare dintre cele 2 clase.
- profesorul georgescu are 2 ore în afara intervalului său preferat. Ora cu clasaAB este la distanță de 2 de intervalul preferat, iar ora cu clasaXB este la distanță 1 de intervalul preferat. Fiecare dintre distanțe se înmulțește cu costul de 2.
- clasaAB preferă să nu existe ferestre, dar marți are fereastră în intervalul 9-10, deci se adaugă un cost de 3.
- Rezultă astfel un cost de 11 pentru această soluție.

Predicate ajutătoare

- Dată fiind o soluție `Sol = (Context, Slots)`, predicatul `printSched(Sol)` va afișa orarul prezentat în soluție.
- Oricând aveți nevoie să apelați un predicat care cere descrierea problemei, folosiți o interogare de form `?- problem()`
- Pentru a afișa o valoare fără ca aceasta să fie sumarizată de către Prolog (prin …), folosiți predicatul `write(S)`.
- Pentru a vă consulta soluțiile, puteți folosi următoarele predicate:
  - `sol(Pb)` – află pe rând și afișează soluțiile pentru o problemă cu numele `Pb` (e.g. `zero`).
  - `sols(Pb)` – află toate soluțiile și le afișează.
  - `nsols(Pb)` – afișează numărul de soluții.
- Pentru a rula 1 test, utilizați predicatul `vmtest(+NumeTest)`.

- Pe parcursul temei sunt **foarte utile** predicatele `findall/3` și `forall/2` predefinite în Prolog. Folosiți-le!
- Exemplu: având o listă de perechi care au pe a doua poziție o listă, vreau să strâng într-o listă valorile de pe prima poziție a perechilor în care lista este formată dintr-un singur element, care este mai mare decât 5, iar acest valori vreau să le asamblez prin operatorul `-` cu unicul element din listă:

```
?- L = [(5, [6]), (a, [2,3,4]), (7, [2]), (3, [9])],

   findall(X-Y, (member((X, [Y]), L), Y > 5), LOut)

LOut = [5-6, 3-9].
```

- Exemplu: vreau să verific că într-o listă toate elemente sunt la rândul lor liste, și aceste liste au ca elemente numere pozitive:

```
?- L = [ [1,2,3], [4,5,6], [3,5,7], [2] ],

    forall(member(E, L), (

                        is_list(E),

                        forall(member(X, E), ( number(X), X > 0 ))

                        )).
```

- Nu uitați de predicatul `trace` care activează modul de tracing – poate fi adăugat inclusiv în interiorul codului pentru a activa depanarea. Se iese din modul de depanare cu `notrace` și, eventual, `nodebug`.
- Pentru a folosi depanatorul vizual, activați-l cu `guitracer`; îl puteți dezactiva cu `noguitracer`.

CSP în Prolog

Cel mai simplu mod de a rezolva probleme **CSP** este **backtracking-ul**.

Prolog are încorporat un sistem de căutare a soluțiilor deci poate rezolva ușor probleme de tip CSP generând toate posibilitățile posibile și apoi validându-le.

De exemplu dacă cunoaștem numele și vârsta unor persoane și vrem să la găsim doar pe cele majore putem reprezenta problema în felul următor:

```
problem(age, [

    people([

        person(ana, 23),

        person(alin, 17)

    ]),

    constraints([

        min_age(any_person, 18)
```

```
    ])
]).
```

Putem accesa un element din context în felul următor:

```
get_people(Problem, People) :-
    problem(Problem, Context),
    member(T, Context),
    T = people(People).
```

Relaxăm (ignorăm) constrângerile și generăm toate posibilitățile:

```
solve(Problem, Solution) :-
    get_people(Problem, People),
    member(Solution, People),
```

Accesăm constrângerile și verificam fiecare posibilitate:

```
solve(Problem, Solution) :-
    get_people(Problem, People),
    get_constrains(Problem, Constraints),
    member(Solution, People),
    valid(Solution, Constraints),
```

Unde funcția `valid` verifică dacă o soluție îndeplinește toate constrângerile.

CSP eficient în Prolog

Deși este foarte simplu să generăm toate soluțiile posibile și apoi să vedem care este cea mai bună, acest lucru este foarte ineficient.

O metodă mai eficientă este următoarea:

- într-un predicat recursiv, se generează un slot pentru fiecare instanță de activitate ce trebuie programată, generându-se soluții parțiale (doar o parte din activități programate).
- după generarea fiecărui slot, se calculează costul constrângerilor încălcate de sloturile generate până la acest nivel.
- dacă acest cost este mai mare decât cel mai bun cost găsit până acum, nu continuăm pe această ramură.
- dacă avem o soluție completă și costul este zero, aceasta este sigur o soluție bună și nu mai continuăm căutarea.

Resurse

- arhiva de pornire

Changelog

- 9.05 – publicare temă
- 9.05 7:30 – redenumire și clarificare constrângere de tip `continuous`; evidențiere ultima secțiune
- 9.05 14:00 – Teste pentru numărul de soluții. Predicate ajutătoare pentru consultarea facilă a soluțiilor.
- 10.05
    - Am eliminat cerința ca predicatele `schedule` să funcționeze pe un argument `Sol` deja legat. Am înlocuit predicatul cerut `schedule/3` cu `cost/2`.

    - Am eliminat wildcard-urile `any*` pentru entități.
    - Am adăugat teste pentru până la 100 de puncte.
    - Am îmbunătățit secțiunea de Predicate ajutătoare din enunț.
- 15.05 0:00
    - corectare specificație constrângere `continuous` și corectare exemplu.
    - corectare teste `best`
    - adăugare teste bonus.
- 16.05 25:55
    - punctajul se primește doar dacă exercițiul a fost rezolvat.
    - vmchecker
