BLIA-MAS Laborator 08

Cuprins:

a. Planificarea automata
b. Un sistem de planificare liniara
c. Planificarea in lumea blocurilor

Planificare automata

Planificare presupune stabilirea unei secvente de actiuni care conduc la atingerea unui anumit scop. Cele mai multe probleme de planificare se refera la domenii complexe, pentru care reprezentarea si prelucrarea descrierii complete a starilor problemei de rezolvat poate deveni foarte costisitoare. Din aceasta cauza, de multe ori este nevoie sa se adopte o strategie de reprezentare partiala a universului problemei cat si o strategie de descompunere a problemei in subprobleme. **Reprezentarea partiala** a universului problemei se face modificand strategia de reprezentare integrala a starilor pe parcursul rezolvarii problemei in aceea de mentinere a unei stari curente si inregistrarea schimbarilor aduse de o actiune starii curente. Apare insa problema definirii a ce inseamna schimbari aduse de o actiune. Referitor la acest aspect, rationamentul despre actiuni trebuie sa rezolve trei probleme, clasice in inteligenta artificiala in special in cazul rationamentului de bun simt, cotidian:

- *problema cadrului* care presupune identificarea tuturor faptelor care nu s-au schimbat ca efect al executarii unei actiuni;
- *problema calificarii* care consta in inregistrarea tuturor preconditiilor necesare pentru executarea unei actiuni;
- *problema ramificarii* care se refera la necesitatea specificarii tuturor consecintelor unei actiuni.

De cele mai multe ori este imposibila specificarea integrala a tuturor acestor conditii si orice sistem de planificare automata trebuie sa recurga la ipoteze simplificatoare si implicite.

Descompunerea problemei in subprobleme si sinteza planurilor partiale pentru fiecare subproblema este o strategie utila pentru stapanirea complexitatii problemelor de planificare dar care implica anumite dificultati. Metodele de generare automata a planurilor pot fi impartite in doua categorii: metode de planificare liniara si metode de planificare neliniara. In cazul **planificarii liniare**, o secventa de scopuri este rezolvata prin satisfacerea fiecaui subcop, pe rand. Un plan generat de o astfel de metoda contine o secventa de actiuni care duce la satisfacerea primului subscop, apoi secventa de actiuni care duce la satisfacerea celui de-al doilea subscop, si asa mai departe. Aceasta metoda functioneaza in cazul in care subproblemele rezultate din descompunerea problemei initiale sunt independente sau pot fi astfel ordonate incat rezolvarea unei subprobleme sa nu afecteze rezolvarea unei subprobleme anterioare.

Aceasta metoda nu poate fi utilizata pentru rezolvarea problemelor de planificare in care subproblemele interactioneaza. In acest caz, trebuie utilizata o metoda de **planificare neliniara** care genereaza planuri prin considerarea simultana a mai multor subprobleme obtinute din descompunerea problemei initiale. Planul este intai incomplet specificat si rafinat ulterior considerand interactiunile existente. O astfel de metoda se numeste planificare neliniara deoarece planul nu este compus dintr-o secventa liniara de subplanuri complete.

Problemele de planificare rpesupun un proces de cautare. In general, aceasta cautare este condusa de scopuri: se porneste de la starea finala si se deduc secventele de operatori care fac trecerea dintre starea initiala si cea finala.

Un sistem de planificare liniara

In continuare prezentam o strategie de planificare liniara, bazata pe sistemul STRIPS (STandford Research Institute Problem Solver). Acesta utilizeaza conceptul de **analiza bazata pe modalitati** ("Means End Analysis"): la rezolvare unei anumite probleme se presupune scopul atins si se analizeaza care sunt actiunile ce trebuie executate pentru a realiza acest deziderat.

Presupunerea care sta la baza constructiei sistemului este aceea ca domeniul problemei de rezolvat nu se schimba semnificativ prin executarea unei actiuni. Starea curenta a universului problemei este desscrisa printr-o multime de formule bine formate in logica cu predicate de ordinul I. Descrierea este completata de axiome specifice domeniului, cu ajutorul carora se pot infera caracteristici implicite ale unei stari. Trecerea dintr-o stare in alta se face prin aplicarea unui operator de plan, deci a unei actiuni. Acest operator este definit prin urmatoarele elemente:

- *Actiune* care reprezinta actiunea asociata operatorului.
- *Lista Preconditiilor* (LP) ce contine formule care trebuie sa fie adevarate intr-o stare a problemei pentru ca operatorul sa poata fi aplicat.
- *Lista Adaugarilor* (LA) ce contine formulele care vor deveni adevarate dupa aplicarea operatorului.
- *Lista Eliminarilor* (LE) ce contine formulele care vor deveni false dupa aplicarea operatorului.

```
;; "planificare.lsp" - rezolvarea problemelor utilizand planificarea liniara
;; (sistemul STRIPS)

(load "ansicl.lsp")

;;Implementarea in LISP defineste structura *OPERATORI* care are patru campuri ce corespund ;;elementelor de definire a unui operator de plan. O actiune este o lista LISP, avand pe prima pozitie ;;simbolul EXECUTA, urmat de actiunea propriuzisa.
```

```
(defvar *OPERATORI* nil)

(defstruct operator
  (actiune nil)
  (preconditii nil)
  (lista-adaugari nil)
  (lista-eliminari nil))

(defun utilizeaza (lista-operatori)
  "lista-operatori devine multimea operatorilor problemei. Intoarce numarul operatorilor"
    (length (setf *OPERATORI* lista-operatori)))

;;*op-scoala* este o lista de operatori pentru rezolvarea problemei parintelui ce trebuie
sa-si duca fiul la ;;scoala cu automobilul.

(defparameter *op-scoala*
 `(,(make-operator
       :actiune '(executa conduce-fiul-la-scoala)
       :preconditii '(fiu-acasa automobil-functioneaza)
       :lista-adaugari '(fiu-la-scoala)
       :lista-eliminari '(fiu-acasa))
   ,(make-operator
       :actiune '(executa service-instaleaza-baterie)
       :preconditii
        '(automobil-necesita-baterie
         service-cunoaste-problema service-primeste-bani)
        :lista-adaugari '(automobil-functioneaza))
   ,(make-operator
       :actiune '(executa spune-service-problema)
       :preconditii '(in-comunicare-cu-service)
       :lista-adaugari '(service-cunoaste-problema))
   ,(make-operator
       :actiune
          '(executa telefoneaza-lista-adaugari-service)
       :preconditii '(stie-numar-de-telefon)
       :lista-adaugari '(in-comunicare-cu-service))
   ,(make-operator :actiune '(executa cauta-numar)
       :preconditii '(are-carte-de-telefon)
       :lista-adaugari '(stie-numar-de-telefon))
   ,(make-operator :actiune '(executa plateste-service)
       :preconditii '(are-bani)
       :lista-adaugari '(service-primeste-bani)
       :lista-eliminari '(are-bani))))

(utilizeaza *op-scoala*)
```

```
;;============================================================
===========
```

;; nucleul programului de planificare foloseste reprezentarea unei stari printr-o lista de atomi LISP.

```
(defun strips (stare scopuri
        &optional (*OPERATORI* *OPERATORI*))
  (sterge #'atom (satisfacere-scopuri
                  (cons '(START) stare) scopuri nil)))
```

;; functia *satisface-scopuri* primeste trei parametri: *stare*, reprezentand starea curenta, *scopuri* – lista ;;scopurilor care trebuiesc satisfacute pe rand, pornind de la starea curenta si *stiva-scopuri* – lista scopurilor ;;nesatisfacute pe calea curenta de cautare. Functia incearca satisfacerea fiecarui scop din lista *scopuri* prin ;;apelul functiei *realizeaza-scop*.

```
(defun satisfacere-scopuri (stare scopuri stiva-scopuri)
  (let ((stare-curenta stare))
    (when
      (and (every
            #'(lambda (scop)
                (setf stare-curenta
                  (realizeaza-scop stare-curenta scop
                                   stiva-scopuri)))
            scopuri)
          (subsetp scopuri stare-curenta :test #'equal))
     stare-curenta)))

(defun realizeaza-scop (stare scop stiva-scopuri)
  (cond ((member scop stare :test #'equal) stare)
        ((member scop stiva-scopuri :test #'equal) nil)
        (t (some #'(lambda (operator)
                   (aplica-operator stare scop
                     operator stiva-scopuri))
             (extrage scop *OPERATORI* #'potrivit-p)))))
```

;; *aplica-operator* apeleaza recursiv functia de pornire (*satisfacere-scopuri*) pentru a satisface, pe rand, ;;toate scopurile din lista de preconditii a operatorului. Apelul recursiv primeste o noua stiva de scopuri ;;nesatisfacute, obtinuta prin adaugarea scopului curent din lista de preconditii la stiva primita ca argument ;;de functia *aplica-operator*. Se observa ca stiva urmareste lantul apelurilor recursive ale functiei ;;*satisfacere-scopuri*.

```
(defun aplica-operator (stare scop operator stiva-scopuri)
  (let ((stare2 (satisfacere-scopuri stare
                 (operator-preconditii operator)
                 (cons scop stiva-scopuri))))
```

```
    (when stare2
      (append
        (sterge
          #'(lambda (x)
              (member x
                (operator-lista-eliminari operator)
                :test #'equal))
          stare2)
        (list (operator-actiune operator))
        (operator-lista-adaugari operator)))))

(defun potrivit-p (scop operator)
  (member scop (operator-lista-adaugari operator)
    :test #'equal))

(defun extrage (element secventa functie-test)
  (delete nil
    (mapcar
      #'(lambda (i)
          (when (funcall functie-test element) i))
      secventa)))

(defun sterge (functie-test lista)
  (delete nil
    (mapcar
      #'(lambda (elem)
          (unless (funcall functie-test elem) elem))
      lista)))

(defun subsetp (subset set &rest args)
  (every
    #'(lambda (element)
        (apply #'member element set args)) subset))
```

;; modificarea functiei *realizeaza-scop* pentru a considera operatorii aplicabili in ordinea numarului de ;;preconditii care nu sunt adevarate in starea curneta, introducand astfel o euristica in satisfacerea ;;scopurilor. Functia predefinita *sort* ordoneaza lista primita ca argument in ordine crescatoare, criteriul de ;;comparatie fiind functia transmisa ca parametru. Lista initiala este distrusa.

```
(defun realizeaza-scop (stare scop stiva)
  (cond
    ((memberg scop stare) stare)
    ((memberg scop stiva) nil)
    (t (some
```

```
        #'(lambda (operator)
            (aplica-operator stare scop operator stiva))
        (operatori-potriviti scop stare)))))

(defun operatori-potriviti (scop stare)
"Intoarce o lista de operatori potriviti sortata dupa numarul de preconditii nesatisfacute in
stare"
  (sort (extrage scop *OP* #'potrivit)
    #'(lambda (x y)
        (< (criteriu x stare)(criteriu y stare)))))

(defun criteriu (op stare)
  (length (remove-if #'(lambda (x) (memberg x stare))
            (operator-preconditii op))))
(defun memberg (elem lista)
  (member elem lista :test #'equal))

(defun adaug-daca (test lista)
  (remove-if #'(lambda(x) (not (funcall test x))) lista))

(defun este-actiune (element)
  (and (consp element)
       (or (equal (first element) 'executa)
          (equal (first element) 'START))))

;; Exemplu de apel
;; (strips '(fiu-acasa are-carte-de-telefon are-bani) '(fiu-la-scoala) *OPERATORI*)

;;============================================================
============================
(defun strips (stareinit scopuri)
  (adaug-daca #'este-actiune
    (caut-strips (list stareinit) nil
      #'strategie-eur scopuri)))

;; Exemplu de apel
;; (strips '(fiu-acasa are-carte-de-telefon are-bani) '(fiu-la-scoala) )

(defun caut-strips (open closed strategie scopuri)
  (format t "~& CAUT: ~a" open)
  (cond ((null open) nil)
      ((subsetp scopuri (first open)) (first open))
      (t (caut-strips (funcall strategie
                       (expand open closed)
                       (rest open) scopuri)
         (cons (first open) closed) strategie scopuri))))
```

```
(defun expand (open closed)
  (remove-if
    #'(lambda (stare)
        (or (memberg stare open)(memberg stare closed)))
    (succ-strips (first open))))

(defun strategie-eur (stari-noi rest-stari scopuri)
  "Intoarce noua lista OPEN cu starile sortate in ordine crescatoare a numarului de conditii din stare care nu sunt in scopuri (un fel de distanta a starii fata de scopuri)"
  (sort (append stari-noi rest-stari)
    #'(lambda (x y)
        (< (criteriu x scopuri)(criteriu y scopuri)))))

(defun criteriu (st scopuri)
  (length (remove-if #'(lambda (conditie)
                         (memberg conditie scopuri))
                     st)))

(defun succ-strips (stare)
  (mapcar
    #'(lambda(op)
        (append
          (remove-if
            #'(lambda(x) (memberg x (operator-le op)))
            stare)
          (list (operator-actiune op))
          (operator-la op)))
    (operatori-aplicabili stare)))

(defun operatori-aplicabili (stare)
  "Intoarce o lista de operatori aplicabili in stare"
  (adaug-daca
    #'(lambda (op)
        (subset (operator-preconditii op) stare))
    *OP*))
```

Planificare in lumea blocurilor

Exista o suprafata plana, numita masa, si un numar de cuburi de aceeasi dimensiune. Un cub poate fi plasat pe masa sau peste alt cub cu ajutorul bratului unui robot. Bratul nu poate tine decat un singur cub la un moment dat. Se pune problema generarii unui plan de transformare dintr-o configuratie initiala intr-o configuratie finala.

Actiune: muta x de pe y pe z (x,y si z sunt blocuri)

Preconditii: liber(x); liber(z); pe(x,y)
Lista Eliminari: liber(z); pe(x,y)
Lista Adaugari: liber(y); pe(x,z)

Actiune: muta x de pe masa pe z (x si z sunt blocuri)
Preconditii: liber(x); liber(z); pe(x,masa)
Lista Eliminari: liber(z); pe(x,masa)
Lista Adaugari: pe(x,z)

Actiune: muta x de pe y pe masa (x si y sunt blocuri)
Preconditii: liber(x); pe(x,y)
Lista Eliminari: pe(x,y)
Lista Adaugari: pe(x,masa); liber(y)

```
(defun constructor (blocuri)
  (let ((operatori))
    (dolist (a blocuri)
     (dolist (b blocuri)
       (unless (equal a b)
          (dolist (c blocuri)
             (unless (or (equal c a)(equal c b))
                (push (mutare a b c) operatori)))
          (push (mutare a 'masa b) operatori)
          (push (mutare a b 'masa) operatori))))
    operatori))

(defun mutare (a b c)
  (make-operator
     :actiune `(executa (muta ,a de pe ,b pe ,c))
     :preconditii `((liber ,a)(liber ,c)(,a pe ,b))
     :lista-adaugari (muta a b c)
     :lista-eliminari (muta a c b)))

(defun muta (a b c)
(if (eq b 'masa) `((,a pe ,c))  `((,a pe ,c)(liber ,b))))

(utilizeaza (constructor '(a b c)))
```