Inteligenta Artificiala - Tema 2
Planificare in jocul "Game About Squares"

Teodor-Stefan Dutu

Universitatea Politehnica Bucuresti
Facultatea de Automatica si Calculatoare
Grupa 341C3

Abstract. Tema propune crearea unui plan de rezolvarea unor niveluri ale jocului *Game About Squares*, pe care il trateaza atat ca pe o problema de cautare in spatiul starilor, cat si ca pe o problema de planificare modelata in `STRIPS` si `ADL`.

Cuprins

1 Cerinta 1 - Operatorii STRIPS

Pentru a putea crea planuri pentru rezolvarea acelor niveluri din `Game About Squares` care contin maximum 2 patrate, am definit urmatorii operatori.

1.1 Operatorul `move_changer`

Acest operator muta un patrat de la pozita sa la o pozitie adiacenta la care se gaseste un *schimbator de directie*, modificand si directia in care se va misca, pe viitor, patratul.

```
move_changer(P, DirP, Xp, Yp, Xs, Ys, DirS)
LP: pos(P, Xp, Yp) & schimbator(Xs, Ys, DirS) & delta(DirP, Xp, Yp, Xs, Ys)
    & empty(Xs, Ys) & dir(P, DirP)
LE: empty(Xs, Ys) & dir(P, DirP) & pos(P, Xp, Yp)
LA: empty(Xp, Yp) & dir(P, DirS) & pos(P, Xs, Ys)
```

1.2 Operatorul `move_empty`

Acest operator muta un patrat la o pozitie adiacenta daca pe aceasta nu se gasesc niciun alt patrat si niciun schimbator de directie.

```
move_empty(P, DirP, Xi, Yi, Xf, Yf)
LP: pos(P, Xi, Yi) & fara_schimbator(Xf, Yf) & delta(DirP, Xi, Yi, Xf, Yf)
    & empty(Xf, Yf) & dir(P, DirP)
LE: empty(Xf, Yf) & pos(P, Xi, Yi)
LA: empty(Xi, Yi) & pos(P, Xf, Yf)
```

1.3 Operatorul `move_square_empty_empty`

Acest operator muta un patrat la o pozitie unde se afla deja un alt patrat, ceea ce va duce la "impingerea" acestuia din urma, si trateaza cazul in care in niciuna dintre noile pozitii ale patratelor nu se gaseste un schimbator de directie.

```
move_square_empty_empty(P1, X1, Y1, DirP1, P2, X2, Y2, X3, Y3)
LP: pos(P1, X1, Y1) & pos(P2, X2, Y2) & fara_schimbator(X2, Y2)
    & fara_schimbator(X2, Y2) & delta(DirP1, X1, Y1, X2, Y2)
    & delta(DirP1, X2, Y2, X3, Y3) & empty(X3, Y3) & dir(P1, DirP1)
LE: pos(P1, X1, Y1) & pos(P2, X2, Y2) & empty(X3, Y3)
LA: pos(P1, X2, Y2) & pos(P2, X3, Y3) & empty(X1, Y1)
```

1.4 Operatorul `move_square_changer_empty`

Acest operator muta un patrat la o pozitie unde se afla deja un alt patrat, ceea ce va duce la "impingerea" acestuia din urma, si trateaza cazul in care pe pozitia initiala a patratului "impins" se gaseste un schimbator de directie.

```
move_square_changer_empty(P1, X1, Y1, DirP1, P2, X2, Y2, X3, Y3, DirP2)
LP: pos(P1, X1, Y1) & pos(P2, X2, Y2) & schimbator(X2, Y2)
    & fara_schimbator(X3, Y3) & delta(DirP1, X1, Y1, X2, Y2)
    & delta(DirP1, X2, Y2, X3, Y3) & empty(X3, Y3) & dir(P1, DirP1)
LE: pos(P1, X1, Y1) & pos(P2, X2, Y2) & empty(X3, Y3) & dir(P1, DirP1)
LA: pos(P1, X2, Y2) & pos(P2, X3, Y3) & empty(X1, Y1) & dir(P1, DirP2)
```

1.5 Operatorul `move_square_empty_changer`

Acest operator muta un patrat la o pozitie unde se afla deja un alt patrat, ceea ce va duce la "impingerea" acestuia din urma, si trateaza cazul in care pe pozitia finala a patratului "impins" se gaseste un schimbator de directie.

```
move_square_empty_changer(P1, X1, Y1, DirP1, P2, X2, Y2, DirP2, X3, Y3, DirS)
LP: pos(P1, X1, Y1) & pos(P2, X2, Y2) & schimbator(X3, Y3) & dir(P1, DirP1)
    & fara_schimbator(X2, Y2) & delta(DirP1, X1, Y1, X2, Y2)
    & delta(DirP1, X2, Y2, X3, Y3) & empty(X3, Y3) & dir(P2, DirP2)
LE: pos(P1, X1, Y1) & pos(P2, X2, Y2) & empty(X3, Y3) & dir(P2, DirP2)
LA: pos(P1, X2, Y2) & pos(P2, X3, Y3) & empty(X1, Y1) & dir(P2, DirS)
```

1.6 Operatorul `move_square_changer_changer`

Acest operator muta un patrat la o pozitie unde se afla deja un alt patrat, ceea ce va duce la "impingerea" acestuia din urma, si trateaza cazul in care pe pozitiile finale ale ambelor patrate se gasesc schimbatoare de directie.

```
move_square_changer_changer(P1, X1, Y1, DirP1, P2, X2, Y2, DirP2, X3, Y3, DirS3)
LP: pos(P1, X1, Y1) & pos(P2, X2, Y2) & schimbator(X3, Y3)
    & schimbator(X2, Y2) & delta(DirP1, X1, Y1, X2, Y2)
    & delta(DirP1, X2, Y2, X3, Y3) & empty(X3, Y3) & dir(P2, DirP2) & dir(P1, DirP1)
LE: pos(P1, X1, Y1) & pos(P2, X2, Y2) & empty(X3, Y3) & dir(P2, DirP2)
    & dir(P1, DirP1)
LA: pos(P1, X2, Y2) & pos(P2, X3, Y3) & empty(X1, Y1) & dir(P1, DirP2)
    & dir(P2, DirS3)
```

2 Cerinta 2 - Operatorii ADL

Inainte de a defini un operator pentru apasarea pe un patrat intr-un joc cu oricate patrate, a fost nevoie sa definesc un predicat auxiliar, `between_dir`, ilustrat mai jos. Acesta este adevarat daca pozitia exprimata prin coordonatele `(x2, y2)` se afla intre pozitiile `(x1, y1)` si `(x3, y3)` pe directia `Dir`.

```
(define between_dir (?Dir ?x1 ?y1 ?x2 ?y2 ?x3 ?y3)
    (or
        (and
            (eq ?y2 ?y3)
            (eq ?y1 ?y2)
            (not (eq ?x1 ?x2))
            (not (eq ?x2 ?x3))
            (or
                (and
                    (eq ?Dir east)
                    (between ?x1 ?x2 ?x3))
                (and
                    (eq ?Dir west)
                    (between ?x3 ?x2 ?x1))))
        (and
            (eq ?x2 ?x3)
            (eq ?x1 ?x2)
            (not (eq ?y1 ?y2))
            (not (eq ?y2 ?y3))
            (or
                (and
                    (eq ?Dir south)
                    (between ?y3 ?y2 ?y1))
                (and
                    (eq ?Dir south)
                    (between ?y1 ?y2 ?y3))))))
```

In continuare, folosindu-ma de predicatul de pe pagina anterioara, am definiti operatorul `move_square` atat in `ADL` cat si in codl `Python` care rezolva tema. Operatorul muta un patrat in directia indicata de acesta si "impinge" toate patratele in acelasi mod, daca sunt pe directia de miscare a patratului apasat si daca intre acesta si ele nu exista nicio pozitie goala (fara un patrat). De mentionat este ca in `Python` functia `move_square` mai primeste inca un parametru pe langa patratul apasat si anume starea curenta. Acest lucru este necesar pentru a facilita cautarea celorlalte patrate, a schimbatoarelor etc.

```
(:action move_square
    :parameters (?P)
    :precondition (and
        (dir ?P ?Dir)
        (pos ?P ?x ?y)
        (delta ?Dir ?x ?y ?new_x ?new_y))
    :effect (and
        (forall (?P2)
            (when
                (and
                    (not (eq ?P ?P2))
                    (exists (?x2 ?y2 ?Dir2)
                        (and
                            (dir ?P2 ?Dir2)
                            (pos ?P2 ?x2 ?y2)))
                    (not (exists (?x3 ?y3)
                        (and
                            (empty ?x3 ?y3)
                            (between_dir ?Dir ?x ?y ?x3 ?y3 ?x2 ?y2)))))
                (and
                    (not (pos ?P2 ?x2 ?y2))
                    (pos ?P ?new_x2 ?new_y2)
                    (when (schimbator ?new_x2 ?new_y2 ?NewDir2)
                        (and
                            (not (dir ?P2 ?Dir2))
                            (dir ?P2 ?NewDir2))))))
        (not (pos ?P ?x ?y))
        (pos ?P ?new_x ?new_y)
        (when (schimbator ?new_x ?new_y ?NewDir)
            (and
                (not (dir ?P ?Dir))
                (dir ?P ?NewDir)))))
```

De asemenea, in `Python`, am inlocuit portiunea de mai jos, care verifica sa nu existe vreo pozitie libera pe linia de la patratul apasat pana la `P2` daca aceasta linie e pe directia descrisa de `Dir`, cu o functie echivalenta, dar scrisa imperativ, asa cum e mai natural in `Python`.

```
(not (exists (?x3 ?y3)
    (and
        (empty ?x3 ?y3)
        (between_dir ?Dir ?x ?y ?x3 ?y3 ?x2 ?y2))))
```

3 Bonus - Micsorarea numarului de stari descoperite

Pentru a reduce numarul de stari descoperite am actionat pe doua planuri: pe de o parte am eliminat unele stari redundante sau din care nu se putea ajunge la solutie inca de la inceput, fara a le mai procesa, si pe de alta parte, am folosit o cautare in spatiul starilor de tip `Best-First`, deoarece nu conta calea de cost minim pana la solutie, ci gasirea cat mai rapida a acesteia.

Astfel, am utilizat urmatoarele 3 functii, doua pentru a decide daca o stare trebuie explorata sau nu, iar a treia fiind functia euristica aplicata in cadrul algoritmului `Best-First`.

3.1 Functia euristica

Functia folosita este **suma distantelor euclidiene de la fiecare patrat la scopul acestuia**. Distanta `Manhattan` pare mai potrivita in acest context, dat fiind ca miscarea patratelor se face exclusiv paralel cu *Ox* si *Oy*. Totusi, avand in vedere ca din cauza posibilitatii de a misca mai multe patrate apasand pe unul singur, distanta `Manhattan` nu este admisibila, deoarece poate supraestima numarul real de operatii necesare pentru a ajunge intr-o stare finala. Nici distanta `Euclid` nu este o euristica admisibila, dar dat fiind ca este dominata de cea `Manhattan`, supraestimeaza mai putin numarul real de operatii necesare pentru ca toate patratele sa ajunga in scopurile lor. Din acest motiv, euristica `Euclid` este mai precisa decat `Manhattan` si a fost aleasa astfel incat algoritmul `Best-First` folosit pentru gasirea solutiei sa prefere intotdeauna starile care au patratele cat mai aproape de scopurile lor.

Un contraexemplu care demonstreaza ca niciuna dintre euristicile antementionate nu este admisibila in cazul acestei probleme se gaseste chiar in primul nivel al jocului. Fie starea descrisa de predicatele de mai jos, considerand ca predicatul `goal(Culoare, X, Y)` este adevarat atunci cand la pozitia `(X, Y)` se gaseste scopul cu culoarea `Culoare`.

```
pos(blue, 0, 2)
goal(blue, 0, 1)
dir(blue, south)
pos(red, 0, 1)
goal(red, 0, 0)
dir(red, north)
```

In acest caz, atat `h_euclid(state)` cat si `h_manhattan(state)` intorc valoarea 2, dar este suficienta o singura mutare pentru a plasa ambele patrate in starile lor-scop, si anume:
`move_square_empty_empty(blue, 0, 2, south, red, 0, 1)`.

3.2 Eliminarea starilor redundante

Am identificat doua tipuri de stari redundante si am creat functii pentru detectarea acestora pentru ca mai apoi sa le pot elimina. Aceste tipuri de stari sunt pe de o parte cele in care un patrat ajunge intr-o pozitie din care nu mai poate ajunge in pozitia sa scop si, pe de alta parte, cele in care patratele se departeaza prea mult de *bounding box-ul* nivelului rezolvat. Acest *bounding box* este dreptunghiul de arie minima in interiorul caruia se gasesc toate elementele dintr-un nivel (patrate, schimbatoare de directie, scopuri), in starea sa initiala.

In primul caz, elimin toate starile in care exista cel putin un patrat astfel incat in semiplanul inchis determinat de directia sa nu se gaseste nici scopul acestuia, nici un alt patrat, nici un schimbator cu o alta directie decat cea a patratului in cauza. In cel de-al doilea caz, elimin toate starile in care exista cel putin un patrat care se afla in afara *bounding box-ului* nivelului.

Bibliografie

1. *Cursul de Inteligenta Artficiala: Cursul 7 - Planificare*
   `https://curs.upb.ro/pluginfile.php/441622/mod_resource/content/1/IA_Lect_7_Planif.pdf`
   Data ultimei accesari: 20 Dec 2020
2. *Planning Domain Definition Language - CSCI 431*
   `http://csci431.artifice.cc/notes/pddl.html#conditional-preconditions-and-effects`
   Data ultimei accesari: 20 Dec 2020
3. *Editor de PDDL*
   `http://editor.planning.domains/`
   Data ultimei accesari: 20 Dec 2020