

Domača naloga
Planiranje
Umetna inteligenca

Slavko Žitnik
63060254

18. april 2011

Uvod

Cilj naloge je, da nad domeno robotov na mreži poženemo regresijsko planiranje in delno urejeno planiranje (angl. Partial Order Planning). Izbrali si bomo nekaj ciljev, izmerili čase izvajanje za oba algoritma in jih primerjali.

V drugem delu bomo definirali akcije in prostor za mrežo, na kateri so roboti in škatle ter roboti lahko premikajo škatle.

1 Planiranje na domeni “robots_on_grid”

Na standardnem prostoru 6 polj, na katerih so roboti, si izberemo naslednje cilje:

- a) [at(a,3)] - prestavimo robota a na polje 3
- b) [at(c,1)] - enako, kot a), le da prestavljamo c v drugo smer
- c) [at(a,3), at(a,1)] - cilj, ki ga ne moremo doseči
- d) [at(a,4), at(b,5), at(c,6)] - vse tri robote premaknemo navzgor
- e) [at(a,3), at(b,2), at(c,1)] - zamenjamo robota a in c
- f) [at(a,5), at(b,4), at(c,1)] - robote spravimo v levi zgornji kot, kjer je v kotu b, spodaj c in desno a

Začetno stanje S_0 je definirano:

$S_0 = [at(a,1), at(b,2), at(c,3), clear(4), clear(5), clear(6)]$

Vsak plan bomo zagnali 500x in tako dobili boljše ocene o času izvajanj posameznega plana.

1.1 Planiranje z delno urejenim planom - POP

- a) [at(a,3)]
state1(S0), time(plan(S0, [at(a,3)], Plan), 500, TA, TB),
show_pop(Plan).
Actions = [1:go(b,2,5),1:go(c,3,6),2:go(a,1,2),3:go(a,2,3)]
Actions = [1:go(b,2,5),2:go(a,1,2),2:go(c,3,6),3:go(a,2,3)]
Čas izvajanja: 780ms.
- b) [at(c,1)]
state1(S0), time(plan(S0, [at(c,1)], Plan), 500, TA, TB),
show_pop(Plan).
Actions = [1:go(a,1,4),1:go(b,2,5),2:go(c,3,2),3:go(c,2,1)]
Actions = [1:go(b,2,5),2:go(a,1,4),2:go(c,3,2),3:go(c,2,1)]
Čas izvajanja: 749ms.
- c) [at(a,3), at(a,1)]
state1(S0), time(plan(S0, [at(a,3), at(a,1)], Plan), 500, TA, TB),
show_pop(Plan).
Planer tega problema ne reši, ker ne more doseči vseh ciljev, saj so si nasprotujoči. Če planer popravimo tako, da pred iskanjem plana preveri, če je sploh izpolnljiv, potem deluje pravilno. Primer, kako bi izgledala dopolnitev, je spodaj.

```
impossible(Goals) :-  
    member( G1, Goals),  
    member( G2, Goals),  
    G2 \== G1,  
    inconsistent( G1, G2), !.
```

```
plan( StartState, Goals, Plan) :-  
    impossible(Goals), !, fail  
    ;  
    add_intervals...
```

- d) [at(a,4), at(b,5), at(c,6)]
state1(S0), time(plan(S0, [at(a,4), at(b,5), at(c,6)], Plan), 500, TA, TB),
show_pop(Plan).
Actions = [1:go(a,1,4),1:go(b,2,5),1:go(c,3,6)]
Čas izvajanja: 390ms.
- e) [at(a,3), at(b,2), at(c,1)]
state1(S0), time(plan(S0, [at(a,3), at(b,2), at(c,1)], Plan), 500, TA, TB),
show_pop(Plan).
Actions = [1:go(b,2,5),1:go(c,3,6),2:go(a,1,2),3:go(a,2,3),
4:go(b,5,2),5:go(c,6,5),6:go(c,5,4),7:go(c,4,1)]
Actions = [1:go(b,2,5),2:go(a,1,2),2:go(c,3,6),3:go(a,2,3),
4:go(b,5,2),5:go(c,6,5),6:go(c,5,4),7:go(c,4,1)]
Čas izvajanja: 215593ms = 215s = 3,6min.

f) [at(a,5), at(b,4), at(c,1)]
state1(S0), time(plan(S0, [at(a,5), at(b,4), at(c,1)], Plan), 500, TA, TB),
show_pop(Plan).
Actions = [1:go(a,1,4),2:go(a,4,5),2:go(b,2,1),3:go(b,1,4),
3:go(c,3,2),4:go(c,2,1)]
Čas izvajanja: 18050ms = 18s.

1.2 Regresijsko planiranje

Pred izvedbo regresijskega planiranja smo spremenili predikat *effects*, ki ga uporabljata STRIPS, v *adds* in *deletes* ter preimenovali predikat *inconsistent* v *impossible*:

```
adds( go(R,A,B), [ at(R,B), clear(A)]).
deletes( go(R,A,B), [ at(R,A), clear(B)]).
```

- a) [at(a,3)]
state1(S0), time(plan(S0, [at(a,3)], Plan), 500, TA, TB).
Plan = [go(c,3,6),go(b,2,5),go(a,1,2),go(a,2,3)]
Čas izvajanja: 967ms.
- b) [at(c,1)]
state1(S0), time(plan(S0, [at(c,1)], Plan), 500, TA, TB).
Plan = [go(a,1,4),go(b,2,5),go(c,3,2),go(c,2,1)]
Čas izvajanja: 1388ms.
- c) [at(a,3), at(a,1)]
state1(S0), time(plan(S0, [at(a,3), at(a,1)], Plan), 500, TA, TB).
Za te cilje planer ne najde ustreznega plana. Morali bi ga popraviti enako,
kot smo to storili pri primeru s planiranjem POP.
- d) [at(a,4), at(b,5), at(c,6)]
state1(S0), time(plan(S0, [at(a,4), at(b,5), at(c,6)], Plan), 500, TA, TB).
Plan = [go(c,3,6),go(b,2,5),go(a,1,4)]
Čas izvajanja: 920ms.
- e) [at(a,3), at(b,2), at(c,1)]
Naslednje cilje s tem planerjem izvedemo le enkrat!
state1(S0), time(plan(S0, [at(a,3), at(b,2), at(c,1)], Plan), 1, TA, TB).
Teh ciljev planer ne izvede v doglednem času (>10min). To je najbrž zaradi
ščitenja ciljev, saj bi moral robota *b* umakniti iz pozicije 2, česar pa
planer ne dovoli. Če izmed ciljev odstranimo cilj *at(b,2)* (cilji za točko
e.1)), dobimo naslednji plan:
Plan = [go(c,3,6),go(c,6,5),go(c,5,4),go(b,2,5),go(a,1,2),go(c,4,1),go(a,2,3)]
Čas izvajanja: 98265ms = 98s.
- f) [at(a,5), at(b,4), at(c,1)]
Naslednje cilje s tem planerjem izvedemo le enkrat!
state1(S0), time(plan(S0, [at(a,5), at(b,4), at(c,1)], Plan), 1, TA, TB).
Plan = [go(b,2,5),go(b,5,4),go(a,1,2),go(a,2,5),go(c,3,2),go(c,2,1)]
Čas izvajanja: 43259ms = 43s.

1.3 Časovna primerjava

Če primerjamo čase izvajanja, ugotovimo, da je bilo planiranje s POP vedno hitrejša od regresiranja ciljev. V tabeli 1 so prikazani časi izvajanj za posamezne primere.

	a)	b)	c)	d)	e)	e.1)	f)
POP	780ms	749ms	/	390ms	3,6min	0,1s	18s
REGRESS	967ms	1388ms	/	920ms	>10min	98s	43s

Tabela 1: Primerjava časov izvajanj

Planer POP išče akcije, ki izpolnijo končne cilje ali predpogoje že izvedenih akcij. Med posameznimi akcijami določi odvisnosti, da lahko neodvisne tudi vzporedno izvajamo. Pri regresiranju ciljev planer za vsak cilj preko regresiranja poišče ustrezno akcijo, ki lahko zahteva predpogoje, ki jih tudi mora izpolniti, če že niso izpolnjene. Naš regresor ima implementirano ščitenje ciljev, kar pomeni, da med iskanjem plana ne more uničiti že najdene rešitve. Planiranje s POP je hitrejša zato, ker akcije v primeru neodvisnosti obravnava neodvisno in lahko zaradi tega pregleda manjšo višino drevesa. Na primer da imamo nek plan s 5 akcijami, od katerih lahko 2 izvedemo vzporedno z ostalimi tremi. V tem primeru bo POP planer pregledal drevo akcij do višine 3, regresor pa do višine 5. Slabost regresorja je torej, da gleda vse možne razvrstitve akcij po vrsti, četudi so akcije med seboj popolnoma neodvisne.

Primer a) in b) sta si obratna. Enkrat premikamo robota *a*, drugič pa *c*. Oba planerja hitro poiščeta ustrezna plana.

Primer c) je takšen, da je cilj nemogoč. Oba planerja bi hotela brez ustreznih sprememb preiskati celoten prostor.

Pri primeru d) se morajo roboti premakniti za eno polje navzgor. POP planer skoraj 3x hitreje najde plan, kjer lahko sočasno premakne vse tri robote navzgor.

V primeru e) zamenjamo robota *a* in *c*. POP planer se izkaže hitrejši za 0,5 minute pri 500 ponovitvah. Med iskanjem plana je planer z regresiranjem ciljev hotel premakniti robota *b* iz že ciljne pozicije, česar ni dovolil, zato je poskušal z drugimi plani.

Pri zadnjem primeru f moramo vse robote spraviti v zgornji levi kot. Planer POP 500x izvede plan v 18s, medtem ko planer z regresiranjem ciljev potrebuje 43s samo za 1x-no planiranje.

2 Razširitev domene

Naloga je bila z uporabo planerja premikati robote in kocke po polju $N \times N$ v stilu igre sokoban.

V namen temu je bilo treba definirati akciji *go* in *push*. Akcija *go* definira premikanje robota med sosednjimi celicami, *push* pa akcijo, da določen robot lahko porine kocko. Cilje lahko predstavimo z relacijami *at* in *clear*, s katerimi povemo, kje se nahaja določen robot ali kocka in katera polja so prosta.

2.1 Primeri uporabe

Za namen testiranja sem uporabil polje, dimenzije 5x5, na katerega sem postavil kocke a , b in c ter robota $r1$ in $r2$. Prvotna postavitvev S je predstavljena v tabeli 2.

5			r1		
4					
3		a			r2
2			b		
1				c	
	1	2	3	4	5

Tabela 2: Prostor

2.2 Planer z regresiranjem ciljev

- Robota $r1$ premaknemo na polje 1.0/4.0.
`sample_state(S), time(plan(S, [at(r1, 1.0/4.0)], Plan), 100, _, TB).`

Plan:

```
go(r1,3.0/5.0,3.0/4.0),
go(r1,3.0/4.0,2.0/4.0),
go(r1,2.0/4.0,1.0/4.0)
```

Čas izvajanja: 5678ms = 5s.

- Robota $r1$ premaknemo na polje 2.0/3.0. Pri tem si bo moral umakniti kocko a .
`sample_state(S), time(plan(S, [at(r1, 2.0/3.0)], Plan), 100, _, TB).`

Plan:

```
go(r1,3.0/5.0,3.0/4.0),
go(r1,3.0/4.0,3.0/3.0),
push(r1,a,2.0/3.0,1.0/3.0)
```

Čas izvajanja: 50264ms = 50s.

- Kocko a premaknemo na polje 1.0/3.0.
`sample_state(S), time(plan(S, [at(a, 1.0/3.0)], Plan), 100, _, TB).`

Plan:

```
go(r1,3.0/5.0,3.0/4.0),
go(r1,3.0/4.0,3.0/3.0),
push(r1,a,2.0/3.0,1.0/3.0)
```

Čas izvajanja: 12059ms = 12s.

2.3 Program

2.3.1 Definicija prostora

Za definicijo prostora sem napisal funkcijo, ki generira polja. Uporabil sem knjižnico *clpr* za računanje z vrednostmi polj.

```
%define place literal
:- dynamic place/1.

%Zgenerira polje 5x5 - primer:
%prepare_fields(1.0, 1.0, 1.0, 5.0).
prepare_fields(X, Y, From, To):-
{X<=To, Y<=To, X1=X+1},
assertz(place(X/Y)),
prepare_fields(X1, Y, From, To),
!
;
{X>To, Y<=To, Y1=Y+1},
prepare_fields(From, Y1, From, To),
!
;
{X>To, Y>To},
true.

prepare_fields(1.0, 1.0, 1.0, 5.0). %generiranje polja 5x5

% define sample state
sample_state([at(a,2.0/3.0), at(b,3.0/2.0), at(c,4.0/1.0), at(r1,3.0/5.0),
at(r2, 5.0/3.0), clear(1.0/1.0), clear(2.0/1.0), clear(3.0/1.0),
clear(5.0/1.0), clear(1.0/2.0), clear(2.0/2.0), clear(4.0/2.0),
clear(5.0/2.0), clear(1.0/3.0), clear(3.0/3.0), clear(4.0/3.0),
clear(1.0/4.0), clear(2.0/4.0), clear(3.0/4.0), clear(4.0/4.0),
clear(5.0/4.0), clear(1.0/5.0), clear(2.0/5.0), clear(4.0/5.0),
clear(5.0/5.0)]).

%define robots
is_robot(r1).
is_robot(r2).
%define blocks
is_block(a).
is_block(b).
is_block(c).
```

2.3.2 Definicija premikanja robotov

```
%definition of action go
adjacent(X1/Y1, X2/Y2) :-
X1 == X2, %must not be the same
Y1 == Y2,!,
fail
```

```

;
{X1 = X2 - 1},!, %P1 left
Y1 == Y2
;
{X1 = X2 + 1},!, %P1 right
Y1 == Y2
;
{Y1 = Y2 + 1},!, %P1 up
X1 == X2
;
{Y1 = Y2 - 1},!, %P1 down
X1 == X2.

can(go(Robot, P1, P2), [at(Robot, P1), clear(P2)]) :-
is_robot(Robot),
place(P1),
place(P2),
adjacent(P1, P2).

adds(go(Robot, P1, P2), [clear(P1), at(Robot, P2)]).

deletes(go(Robot, P1, P2), [clear(P2), at(Robot, P1)]).

```

2.3.3 Definicija potiskanja kock

Funkcija *push* definira, da mora robot premakniti kocko iz položaja *P1* na *P2*. Pri tem uporabim funkcijo *robot_pos*, ki glede na smer premikanja enolično določi lokacijo robota *P3*.

```

%definition of action push
robot_pos(X1/Y1, X2/Y2, X3/Y3) :-
%we have to know which way is being moved to position Robot, only one possible option
{X2 < X1, X3 = X1 + 1}, %moving left
{Y3 = Y1},!
;
{X2 > X1, X3 = X1 - 1}, %moving right
{Y3 = Y1},!
;
{Y2 > Y1, Y3 = Y1 - 1}, %moving up
{X3 = X1},!
;
{Y2 < Y1, Y3 = Y1 + 1}, %moving down
{X3 = X1},!.

can(push(Robot, Block, P1, P2), [at(Block, P1), clear(P2), at(Robot, P3)]) :-
is_robot(Robot),
is_block(Block),
place(P1),
place(P2),
adjacent(P1, P2),

```

```
robot_pos(P1, P2, P3),  
place(P3),  
adjacent(P1, P3).
```

```
adds(push(Robot, Block, P1, P2), [at(Robot, P1), at(Block, P2), clear(P3)]) :-  
robot_pos(P1, P2, P3).
```

```
deletes(push(Robot, Block, P1, P2), [at(Robot, P3), at(Block, P1), clear(P2)]) :-  
robot_pos(P1, P2, P3).
```