Situations and Fluents: Examples

Mikhail Soutchanski

November 19, 2020

Coins on desk

Revisit the coins on a desk problem, but generalize to arbitrary many coins. Test with a simple instance of 3 coins that we discussed in class. Fluents -? Actions -?

Action (term): *flip*(C) - turn a coin C over

Fluents (predicates): *headUp*(C, S) - a coin C is head up in situation S
tailUp(C, S) - a coin C is tail up in situation S
onDesk(C, S) - a coin C is on desk in S. /* for the sake of example */

How to implement an initial state and goal states we discussed in class ? /* hht */

```
onDesk(c1,[]).  onDesk(c2,[]).  onDesk(c3,[]).
headUp(c1,[]).  headUp(c2,[]).  tailUp(c3,[]).

goal_state(S) :- headUp(c1,S), headUp(c2,S), headUp(c3,S).
goal_state(S) :- tailUp(c1,S), tailUp(c2,S), tailUp(c3,S).

poss(flip(Coin),S) :- onDesk(Coin,S).             /* Precondition Axiom */
```

How to write successor state axioms for the fluents ?

```
onDesk(Coin, [A | S]) :- onDesk(Coin,S).

headUp(Coin, [A | S]) :- A=flip(Coin), tailUp(Coin,S).
headUp(Coin, [A | S]) :- not A=flip(Coin), headUp(Coin,S).

tailUp(Coin, [A | S]) :- A=flip(Coin), headUp(Coin,S).
tailUp(Coin, [A | S]) :- not A=flip(Coin), tailUp(Coin,S).

useless(flip(Coin), [flip(Coin) | S]).     /* avoid useless repetitions */
```

Sorting by planning: 1

In the initial situation we are given English characters, e.g. *h, o, m, e.*
The goal situation is when letters are sorted, e.g. alphabetically: *e, h, m, o.*
We will solve this problem in a general case using situations and fluents approach to the planning problem and the predicate "useless(A,S)" to exclude redundant branches from the search tree.

We have one fluent *loc*(X,N,S). This predicate is true if a character X is located at a position N in situation S.

There is one action term *exchange*(X,Y) - change positions of characters X and Y when they are consecutive, i.e. if Y is located next to X.

In our instance, in initial situation, "h" is location at the 1st position, "o" is located at the 2nd position, "m" is located at the 3rd position, "e" is located at the 4th position.

```
loc(h,1,[ ]).  loc(o,2,[ ]).  loc(m,3,[ ]).  loc(e,4,[ ]).
```

In the goal situation, "e" must be located at the 1st position, "h" must be located at the 2nd position, "m" at the 3rd position, and "o" at the 4th position.

```
goal_state(S) :- loc(e,1,S), loc(h,2,S), loc(m,3,S), loc(o,4,S).
```

But we want goal state be more general and modular, i.e., a goal must be formulated for any instance. Then, only initial state must change for a new input.

Sorting by planning: 2

Library predicate: *char_code*(*Char*, *Code*). It succeeds if *Code* is the numeric character code of the character *Char*:

> *If Char is instantiated to a one-character constant, Code is unified with the corresponding numeric character code, depending on encoding in use.*

?- char_code(e,Val1). **?- char_code(m,Val2).**
Val1 = 101 Val = 109
How to use this predicate to say that in the goal state all characters are sorted? =
There are no chars C1,C2 such that C1 is before C2, but C1's code is greater than C2.

```
goal_state(S) :- not (loc(C1,N1,S), loc(C2,N2,S), not C1=C2,
     N1 < N2, char_code(C1,X1), char_code(C2,X2), X1 > X2).
```

When possible to exchange X and Y? characters ... | X | Y | ...
Hint: use fluent *loc*(*Char*, *Pos*) locations ... | N | M | ...

```
poss(exchange(X,Y),S):- loc(X,N,S), loc(Y,M,S), M is N+1.

loc(X,N,[A | S]) :- A=exchange(X,Y), loc(X,M,S), loc(Y,N,S).

loc(X,N,[A | S]) :- A=exchange(Y,X), loc(X,M,S), loc(Y,N,S).

loc(X,N,[A | S]):- not A=exchange(X,Any), not A=exchange(Any,X),
                   loc(X,N,S).
```

This is to show a common pattern in SSAs, but in the 1st and 2nd rules we can replace A in the head with action *exchange*(X, Y) and simplyfy the RHS.

Sorting by planning: 3

How to write meaningful rules that will exclude redundant sequences of actions?

Note that if *X* preceeds *Y* in alphabetic order, and *X* is already located before *Y* in situation *S*, then it's stupid to exchange characters *X* and *Y*.

```
useless(exchange(X,Y),S):- char_code(X,Val1), char_code(Y,Val2),
                Val1 < Val2,
                loc(X,N,S), loc(Y,M,S), N < M.
```

/*For simplicity, assume all characters are different */

?- solve_problem(4,Plan). /* a solution with at most 4 steps */
Plan = [exchange(o, m), exchange(o, e), exchange(m, e), exchange(h, e)]
Yes (0.01s cpu, solution 1, maybe more) ? ;
/* corresponds to: home –> hmoe –> hmeo –> hemo –> ehmo */

Plan = [exchange(m, e), exchange(o, e), exchange(h, e), exchange(o, m)]
Yes (0.01s cpu, solution 2, maybe more) ? ;
/* corresponds to: home –> hoem –> heom –> ehom –> ehmo */

Plan = [exchange(m, e), exchange(o, e), exchange(o, m), exchange(h, e)]
Yes (0.02s cpu, solution 3, maybe more) ?
/* corresponds to: home –> hoem –> heom –> hemo –> ehmo */

There are no other 4 step solutions

Tiles on an arbitrary grid

In the spirit of developing general programs that can solve any instance of the planning problem, we would like to generalize the 4×4 tiles problem to an arbitrary grid $n \times m$, where *n* is the number of cells in the row, and *m* is the number of rows:

```
horiz(3).                    /* number of tiles in horizontal dimension */
vertic(2).                   /* number of tiles in vertical dimension */
```

Fluent: the predicate `posit(P,T,S)` is true if a position *P* is occupied by a tile *T* in situation *S*. If a position is vacant, we use 0. Positions are consecutively numbered.

If an initial configuration is this one, how can we represent the initial state?

```
| 0 | 1 | 5 |
| 4 | 3 | 2 |
```

```
posit(1,0,[]).  posit(2,1,[]).  posit(3,5,[]).
posit(4,4,[]).  posit(5,3,[]).  posit(6,2,[]).
```

How can we define the goal state for our instance ?

```
goal_state(S) :- posit(1,1,S), posit(2,2,S), posit(3,3,S),
       posit(4,4,S), posit(5,5,S), posit(6,0,S).
```

Actions (they are terms): *up*(*Tile*), *down*(*Tile*), *left*(*Tile*), *right*(*Tile*). First of all, we have to write the precondition axioms for them, assuming an arbitrary-sized grid.

Tiles on a grid: precondition rules

```
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 |10|
| 11| 12| 13| 14| 15|
| 16| 17| 18| 19| 20|
```

When can a tile move up ?
Note it cannot move up from the top row.
Recall we use *horiz*(*N*) to say how many cells we have in a horizontal dimension. On this slide, N should be 5.

```
poss( up(Tile), S) :- posit(Init,Tile,S), horiz(N),
                         (Init - N) > 0,          /* From Init to Dest */
            Dest is Init - N, posit(Dest,0,S), Tile > 0.
```

Next precondition rule: when is it possible to move a tile right ?
Notice that the tiles from the right most column 5,10,15,20 cannot move right.

```
poss( right(Tile), S) :- posit(Init,Tile,S), horiz(N),
                         (Init mod N) > 0,        /* From Init to Dest */
            Dest is Init + 1, posit(Dest,0,S), Tile > 0.
```

What about action *down*(*T*) ? When is it possible ? Recall *vertic*(*M*); here M is 4.
Obviously, we cannot move a tile down from the bottom row.

```
poss(down(Tile),S):- posit(Init,Tile,S), horiz(N), vertic(M),
                         (Init + N) =< N*M,
            Dest is Init + N, posit(Dest,0,S), Tile > 0.
poss( left(Tile), S) :-                         /*Exercise: homework */
```

Tiles on a grid: SSA rules

First, consider pseudo-code for kind of generic successor state axioms.

```
fluent(X,Y,[ActionThatMakesItTrue | S]):- if_condit_holds(X,Y,S).
```

/* This rule describes when a fluent becomes true after executing an action */

```
fluent(X,Y,[A | S]):- fluent(X,Y,S), not A=actionThatMakeItFalse.
```

/* This is a persistence rule: it describes when a fluent remains true, unless the most recent action A was the action that made it false */

Suppose a tile *T* is at a postion *P* in *S*, i.e., *posit*(*P*, *T*, *S*) is true. Where will it be after moving left? Where will it be after actions *right*(*T*), *down*(*T*), *up*(*T*) ?

```
__|___|__
 _|_P_|__
 _|___|__
```

```
posit(New,Tile,[left(Tile) | S]):- posit(P,Tile,S), New is P-1.
posit(New,Tile,[right(Tile) | S]):- posit(P,Tile,S), New is P+1.
posit(New,Tile, [up(Tile) | S]) :- posit(P,Tile,S), horiz(N),
                        New is P-N.
posit(New,Tile,[down(Tile) | S]) :- posit(P,Tile,S), horiz(N),
              New is P+N.
```

Recall that we need SSA for all changes that happen after each action.
What else will change when a tile *T* moves on a grid from a position *P* ?

Tiles on a grid: SSA rules and heuristics

/* These 4 rules characterize how the vacant position changes */

```
posit(P, 0, [up(Tile) | S]) :- posit(P,Tile,S).
posit(P, 0, [down(Tile) | S]) :- posit(P,Tile,S).
posit(P, 0, [left(Tile) | S]) :- posit(P,Tile,S).
posit(P, 0, [right(Tile) | S]) :- posit(P,Tile,S).
```

Finally, we need a persistence rule saying that the only tile that changes its position is the tile we move. But positions of other tiles remain the same. How can we say this?

```
posit(P,Tile,[A|S]):- posit(P,Tile,S),not Tile=0, not A=up(Tile),
      not A=down(Tile), not A=right(Tile), not A=left(Tile).
```

`useless(M,[LastAct | History]) :-` "if it makes no sense to do move M given LastAct and History",
i.e., *useless*(*M*, *History*) is true if M is a redundant action to do given the list History of actions that have been performed already.

```
useless(up(X),[down(X) | List]).
useless(down(X),[ up(X) | List]).
useless(left(X),[ right(X) | List]).
useless(right(X),[ left(X) | List]).
```

Can combine declarative heuristics with numerical based on the Manhattan heuristic.