CSC384: Lecture 8

- Last time
  - Action Representation; planning as search
- Today
  - STRIPS Planning, Regression planning
- Readings:
  - Today: 8.3 (STRIPS planning in depth, regression planning, briefly resolution-based planning)
  - Next week: uncertainty 10.1, 10.2, start on 10.3

STRIPS Planner

- Last time, discussed intuitive sketch of STRIPS
  - a divide-and-conquer approach
  - tries to find independent plans for individual subgoals and then pieces these plans together
  - recursively tries to achieve necessary preconditions
- We'll sketch a version of the algorithm designed to work with the CWR-D representation
  - contrast with algorithm in text, which is designed to work with the situation calculus representation

STRIPS with CWR-D

- achieve_all(GList,S0,S1,Plan)
  - action sequence Plan applied at state S0 results in state S1, satisfying all goals in GList

```
achieve_all( [ ], S, S, [ ] ).

achieve_all( GList, S0, S2, Plan ) :-
   remove(Goal, GList, RestG),
   achieve(Goal, S0, S1, Plan1),
   achieve_all(RestG, S1, S2, Plan2),
   append(Plan1, Plan2, Plan).
```


STRIPS w/ CWR-D: Goal Selection

- remove(G, GList,RestG)
  - selects a goal G from goal list for achievement
  - implementation #1 below always selects first goal
  - note: we'll see that allowing different orderings is important---it should really be a "choose" not "select"
  - implementation #2 allows backtracking

#1
```
remove( G, [G | RestG], RestG ).
```

#2
```
remove( G, GList, RestG ) :-
   member( G, GList ),
   delete( G, GList, RestG ).
```

removes arbitrary element G from list GList

STRIPS w/ CWR-D: Goal Achievem't

- achieve(G, S0, S1, Plan)
  - action sequence Plan applied at state S0 results in state S1, satisfying all goal G (single goal)
  - all predicates used defined earlier except effect_of
  - effect_of(A,G): action A has G as an effect (exercise)

```
achieve( G , S, S, [ ] )  :-  holds(G, S).

achieve( G, S0, S2, Plan ) :-
   effect_of( A, G ),   preconds(A, PCList),
   achieve_all(PCList, S0, S1, Plan1),
   append(Plan1, [A], Plan),
   result(A, S1, S2).
```


STRIPS: Handling Derived Relations

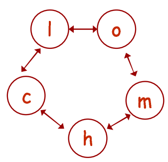
- If we have derived relations, STRIPS can't directly achieve such a fact (not mentioned as effects of any actions)
  - so simply set Body as subgoals to achieve

```
achieve( G , S0, S1, Plan )  :-
  derivedRel(G, Body),
  achieve_all(Body, S0, S1, Plan).
```

Issues with STRIPS (1)

- Order of goal selection can impact quality/length of plan
  - e.g., we picked mov(l,o) to achieve loc(o) in final plan step; but what if we had picked mov(m,o)?
  - might have picked mov(h,m), then mov(c,h), etc. and taken long way around
  - might have gotten in a cycle
- In general, goal selection ordering can benefit from heuristics; and can even require systematic search/backtracking

Start State: loc(o), lck(l), neg(rhk), neg(labtidy) ...
Goal: loc(o), labtidy



Issues with STRIPS (2)

- STRIPS can return incorrect plans!
  - suppose we chose goal loc(o) before labtidy
  - plan for loc(o) is [] (it's true in initial state s0)
  - plan for labtidy is [getkeys, mov(o,l), tidy]
  - the second plan destroys or *clobbers* the subgoal achieved by the first plan!
  - so returned plan [ ] + [gk,m(o,l),t] is incorrect
- **Subgoal protection:**
  - circumvents this problem by protecting achieved subgoals when producing plans for the next subgoals

Subgoal Protection

- Given k goals $[g_1, \ldots, g_k]$ in this order
  - produce a subplan that achieves g_1 (say p_1)
  - produce a subplan p_2 that produces that achieves g_2 *without affecting g_1*
  - in general, produce a plan p_i for g_i that does not affect any g_h ordered before g_i
  - Solution p_1; p_2;… p_k guaranteed to achieve all goals

```
achieve_all( GList, S0, S2, Plan, Protected ) :-
   remove(Goal, GList, RestG ),
   achieve(Goal, S0, S1, Plan1, Protected),
   achieve_all(RestG, S1, S2, Plan2, [Goal | Protected] ),
   append(Plan1, Plan2, Plan).
```

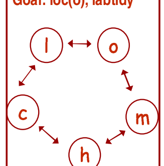

Subgoal Protection (con't)

- Key to above algorithm:
  - achieve(G, S0, S1, Plan, Protected) is not allowed to construct a subplan that "touches" any literal in the protected list
  - exercise: try it (tricky to do this with derived rel'ns)

Example of Protection (1)

- If we choose loc(o) first:
  - we get plan p1 = [ ] (loc(o) true in S0)
  - we protect loc(o) --- it's already achieved
  - attempt to find plan to achieve labtidy without altering loc(o)
  - impossible because of protection
- Once it fails, we retry with labtidy as first goal
  - this will succeed as in original example
  - notice that it's critical to allow algorithm to backtrack over goal choices so it can try a different ordering

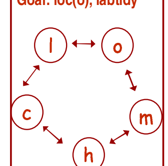
Start State: loc(o), lck(l)
Goal: loc(o), labtidy



Example of Protection (2)

- Same example, but suppose action mov(l,o) magically makes labtidy false!
- Choose labtidy as first goal
  - we get plan p1 = [getkys, mov(o,l), tidy]
  - we protect labtidy
  - attempt to find plan to achieve loc(o) without altering labtidy
  - try to achieve loc(o) using mov(l,o); but this undoes labtidy, so fails due to protection
  - try to achieve loc(o) using mov(m,o); this works; sets up subgoal of loc(m); etc.
  - soln: tidy the lab then go back to office the long way around
- Subgoal protecttion has desired effect

Start State: loc(o), lck(l)
Goal: loc(o), labtidy



Is STRIPS with SGP "Complete"?

- STRIPS with subgoal protection is sound
  - if it returns a plan, the plan is correct (achieves goals)
- But STRIPS with SGP is not complete
  - it may not find a plan even if it exists
  - this is true even if it searches over all goal orderings
  - this is due to its notion of achievement
- Why? Let's consider an example…

Problems with STRIPS (3)

- Example using only two locations -- loc(o), loc(c)
  - but if robot in office and Craig has coffee, if robot leaves office, C throws coffee against wall in megalomaniacal fit of rage (robot must *watch* C drink)
  - so action mov(o,c) has effect neg(chc)
  - Start: neg(cm), neg(chc), neg(rhc), loc(c)
  - Goal: **chc, cm**
- To solve, STRIPS must solve with
  - ordering #1: cm then chc; or
  - ordering #2: chc then cm

Problems with STRIPS (3)

- Ordering #1: cm then chc is not suitable
  - you could achieve cm by simply making coffee
  - If you did that, any way of achieving chc would clobber cm. Robot must grabcoffee – neg(cm) – to give it to Craig
  - Note: you could [makecof, grabcof, makecof] and then take coffee to Craig; but STRIPS won't consider this, since once you achieve cm you can't clobber it. The only reason to consider it is if STRIPS looks ahead to next goal
- Ordering #2: chc then cm is not suitable
  - once you make chc true by the usual plan (make, grab, move, give) , can't leave office to make more
  - Note: you could [makecf, grabcf, makecf,mov,givecf]; but unless it looks ahead to next goal, STRIPS has no reason to consider this

Serializability

- A set of goals G is *serializable* (wrt s0) if there is some ordering of the goals [g_1, …, g_k] s.t.
  - you can achieve g_1 from s0
  - you can achieve g_2 without clobbering g_1 *no matter what plan you used to achieve g_1*
  - you can achieve g_3 without clobbering g_1, g_2 *no matter what plan you used to achieve g_1, g_2* , etc…
- STRIPS-SGP can solve any serializable goal set
  - backtracking over goal orderings must be allowed
- Note: earlier example is not serializable
  - success depends on the plan chosen
  - but we can't allow STRIPS to consider arbitrary plans or we lose the benefits of divide and conquer

STRIPS Summary

- STRIPS biggest problem:
  - forced to completely solve one subgoal before considering how it affects other goals
  - with subgoal protection we get correct plans, but only if subgoal set is serializable
  - but this prevents you from finding plans where goals interact strongly
- A different view: regression planning
  - when you insert an action into a plan, you *consider* how it influences all current subgoals
  - but you still focus on achieving one subgoal

Regression Planning: Intuitions

- Basic idea behind regression is quite simple:
  - given a goal list G, the *regression* of G through action A is the *weakest set of preconditions* WC that ensure G is true after A is performed
  - In other words:
    - if WC holds at state S, then G holds at result(A,S)
    - no logically weaker set of conditions satisfies this property
- This leads to an obvious subgoaling strategy
  - given G, find an action A "that makes progress" on G
  - find a plan P' that achieves WC
  - then return the plan P = [P', A]

Regression Example

- Let's look at intuitions before getting into details
  - consider nonserializable example with G = [chc,cm]

G = [chc,cm]
  Regress G through givecoffee: SG1 = [rhc, loc(o), cm]
SG1 = [rhc, loc(o), cm]
  Regress through mov(c,o): SG2 = [rhc, loc(c), cm]
SG2 = [rhc, loc(c), cm]
  Regress through makecoffee: SG3 = [rhc, loc(c)]
SG3 = [rhc, loc(c)]
  Regress through grabcoffee: SG4 = [cm, loc(c)]
SG4 = [cm, loc(c)]
  Regress through makecoffee: SG5 = [loc(c)]
SG5: True at initial state

If S satisfies [rhc, loc(o), cm] then result(A,S) satisfies [chc,cm]

Regression Planning

- We first need to define the notion of regression formally (and basic idea behind implementation)
- We then need to define a planner that relies on the notion of regression

Regression Planning

- Basic structure of the algorithm:
  - start with subgoal (SG) list equal to goal list
  - Loop:
    - *choose* an action A that:
      a. achieves at least one subgoal on SG list
      b. doesn't destroy any other subgoals on list
      c. preconds are consistent with other subgoals
    - regress SG list through action A to obtain SGNew
    - set SG list to SGNew
  - until all elements in SG list are true in S0
- Conditions b, c necessary, otherwise A cannot make SG list true (more to come)

Why Conditions (b) and (c)

- Why we need condition (b)
  - action a: precond x; effects y, neg(z)
  - subgoals SG = [y, z]
  - impossible to do action a and (immediately) result in a state where SG is true: a achieves y, but makes z false
- Why we need condition (c)
  - action a: precond x, neg(z); effects y
  - subgoals SG = [y, z]
  - impossible to do action a and (immediately) result in a state where SG is true: a achieves y, but requires z to be false when executed; since a doesn't affect z, z must be false immediately after doing a
- Note: (b) and (c) ensure regression is "possible"

Defining Regression for CWR (1)

- regress(A,GL,WP)
  - true if: WP is the weakest precondition such that executing A when WP holds results in goal list GL becoming true; WP is consistent
  - fails if no such consistent WP exists
  - assumes GL is consistent already

```
regress(A,GL,WP) :-
   removeeffects(A,GL,WGL),
   preconds(A,PC),
   addpreconds(PC,WGL,WP).
```


Defining Regression for CWR (2)

- removeeffects(A,GL,WGL)
  - true if WGL obtained by removing effects of A from goallist GL (fails if any goal contradicts effect of A)
  - version for "nonboolean" predicates posted online

```
removeeffects(_,[],[]).
removeeffects(A,[G|RestG],WGL) :-
    achieves(A,G),
    removeeffects(A,RestG,WGL).
removeeffects(A,[G|RestG],[G|RestWGL]) :-
    G \= neg(_),
    not(achieves(A,G)),
    not(achieves(A,neg(G))),
    removeeffects(A,RestG,RestWGL).
removeeffects(A,[neg(G)|RestG],[neg(G)|RestWGL]) :-
    not(achieves(A,G)),
    not(achieves(A,neg(G))),
    removeeffects(A,RestG,RestWGL).
```

Defining Regression for CWR (3)

- Simple auxilliary predicate "achieves"

```
% action A achieves goal G
achieves(A,G) :- addlist(A,AList),
                 member(G,AList).

achieves(A,neg(G)) :- deletelist(A,DList),
                      member(G,DList).
```


Defining Regression for CWR (4)

- regress(A,GL,WP)
  - true if: WP is the weakest precondition such that executing A when WP holds results in goal list GL becoming true; WP is consistent
  - fails if no such consistent WP exists
  - assumes GL is consistent already

```
regress(A,GL,WP) :-
  removeeffects(A,GL,WGL),
  preconds(A,PC),
  addpreconds(PC,WGL,WP).
```


Defining Regression for CWR (4)

- addpreconds(PC,WGL,WP)
  - true if: WP results from adding preconditions PC to (weakened) goal list WGL, and result is consistent
  - fails if preconditions conflict with WGL.
  - WGL is assumed consistent.

```
addpreconds([],WP,WP).
addpreconds([P|RestP],L,WP) :-
  addconsistent(P,L,L1),
  addpreconds(RestP,L1,WP).
```


Defining Regression for CWR (5)

- To add precondition consistently (fails if precondition contradicts subgoal list)

```
addconsistent(P,L,L) :-  member(P,L).

addconsistent(P,L,[P|L]) :-
  P \= neg(_),
  not(member(P,L)),
  not(member(neg(P),L)).

addconsistent(neg(P),L,[neg(P)|L]) :-
  not(member(P,L)),
  not(member(neg(P),L)).
```


Regression Planner

- rplan(GoalList,State,Plan)
  - true if Plan achieves GoalList starting at State
- Basic intuition: see slide 4

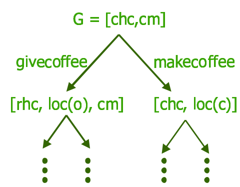
```
rplan(GList,State,[]) :-
  holdsall(GList,State).

rplan(GList,State,Plan) :-
  member(Goal,GList),
  achieves(A,Goal),
  regress(A,GList,NewGList),
  rplan(NewGList,State,Plan1),
  append(Plan1,[A],Plan).
```


Some Notes on rplan

- We assume that initial goal list is consistent
  - we ensure subgoal lists remain consistent in "regress"
- Search occurs with goal and action choices
  - member(Goal,GList) chooses a goal to achieve
  - achieves(A,Goal) chooses action to achieve it
  - backtracking taken care of by Prolog
- This implementation will never work in practice!
  - by allowing Prolog to do the search, we're committing to DFS without cycle checking!
- Exercise (asst?): fix this by controlling search yourself (don't hand it off to Prolog)
  - e.g., use BrFS or iterative deepening

Wrap up of Regression Planning

- Main idea: we are reasoning backward from the goal conditions to S0
  - choose a goal and an action that achieves it
  - search space is not the set of states, but the set of subgoal lists (nbrs are subgoal lists we can reach by regressing consistently through some "useful" action)



"Goal" in this search space is any subgoal list that is true in the initial state

Planning

- Most modern planners more sophisticated than STRIPS/regression
  - but most rely on basic ideas of decomposition and the idea of "regressing" (reasoning backward) from goal
- Partial-order planning (see 8.3 of text)
  - exploits "least commitment" idea by choosing actions without commiting to their order right away
  - nice ideas, but computationally expensive in practice
- Planning as search quite common (fast)
  - use backchaining ideas to guide search/generate heuristics
  - sophisticated search used (e.g., stochastic search)

Situation Calculus

- SC an alternative representation for actions
- A logical language in which
  - situations are terms (e.g., init, s27, S0)
    - *init* a special constant referring to initial state
  - actions are terms (e.g., mov(X,Y), mov(o,l), getkeys)
  - do(A,S) refers to situation that results from doing A in situation S (*do* a special function symbol)
  - domain predicate all have a situation argument (e.g., rhc(s27), or loc(m,init)

SitCalc: Example expressions

- Situations (states of world, but w/ action history)
  - init
  - do(grabcof, init)
  - do(grabcof, S)
  - do(givecof, do(mov(c,o), do(grabcof, init)))
    - corresponds to sequence: grab, move, give at init
- Statements about what's true
  - loc(c, init)
  - cm(init)
  - rhc(do(grabcof, init))
  - chc(do(givecof, do(mov(c,o), do(grabcof, init))))

Actions in the SitCalc

- We specify actions by specifying preconditions and effects
- Preconditions specified using the poss predicate

```
poss(givecof, S) <- rhc(S), loc(o,S).

poss(mov(X,Y), S) <- adj(X,Y,S), acc(Y,S), loc(X,S).
```


Effect Axioms

- Action effects specified using effect axioms

```
chc(do(givecof,S)) <- poss(givecof,S).
loc(Y,do(mov(X,Y),S)) <- poss(mov(X,Y), S).
```

- Sadly, specifying effects alone logically insufficient
  - how do we know if labtidy is true after doing givecof?
  - must explicitly specify *frame axioms* stating that unaffected things remain unchanged after an action

```
lbtdy(do(givecof,S)) <- poss(givecof,S), labtdy(S).
rhk(do(givecof,S)) <- poss(givecof,S), labtdy(S).
```

Reasoning in the SitCalc

- You can use these axioms to prove that certain things are true or false after performing a sequence of actions
  - provide action specifications (incl. frame axioms)
  - state what is true at init (e.g., loc(c,init), cm(init),…)
  - ask query, e.g.,
    - ?- chc(do(givecof, do(mov(c,o), do(grabcof, init)))).
- If axioms allow proof, then you know chc is true after this sequence from init
  - but must rely on negation as failure for false things
  - no way to prove "neg(cm)" after grabcof otherwise

Planning as Theorem Proving

- You can now use SLD to construct a plan
  - given init specification, poss and effect axioms
  - given a goal G such as [chc, cm]
  - ask the query: ?- chc(S), cm(S).
- SLD will return an answer in which variable S is bound to a situation term from which plan can be extracted; e.g.
  - S= do(givecf, do(mov(c,o), do(makecf, do(grabcf, do(makecf,init)))))
- Computationally: this relies on SLD/Prolog doing usual DFS (so may not work very well)