CSC384: Lecture 7

- Last time
  - game tree search (note: no text readings)
- Today
  - Intro to Planning; State and action representations
  - planning: start on STRIPS planning
- Readings:
  - Today: Ch.8.1, 8.2 (STRIPS, skim Situation Calculus, **skip** Event Calculus), 8.3 (forward planning)
  - Next week: 8.3 (STRIPS planning in depth, regression planning, briefly resolution-based planning)

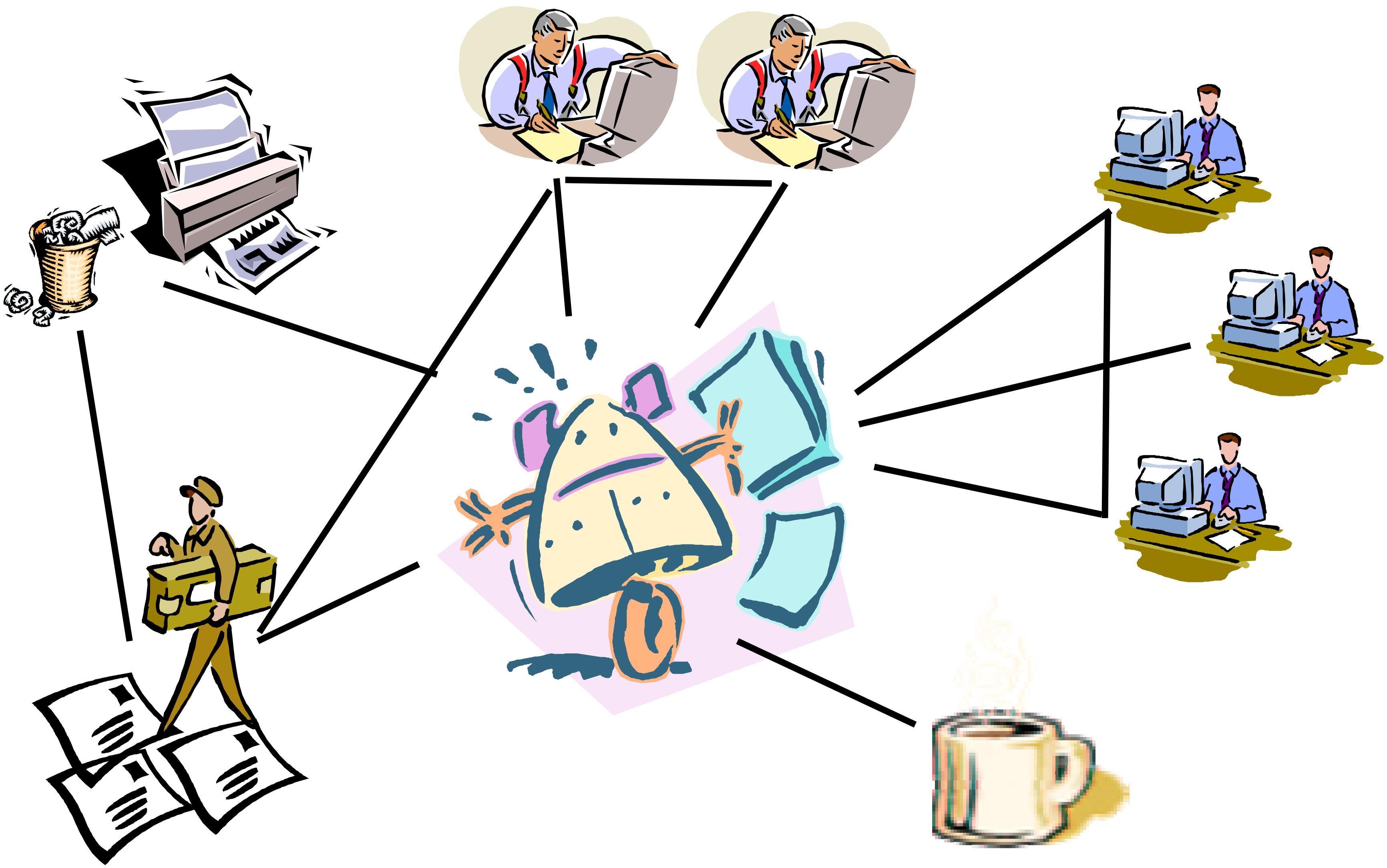
Search in Complex Situations

- Search algorithms so far are graph-based:
  - produce sequence of actions or moves that change the world in specific ways (start → goal)
  - has nothing to say about the actions other than that take you from one state to another
    - e.g., map location (courier), robot position
  - states have no structure to speak of
- Realistic problems involve altering complex states of the world
  - at each state, many different facts are true/false
  - actions change the truth of different facts

Planning

- Planning problems are like search problems
  - given an initial state of the world
  - given some goal *conditions* (facts we would like to make true, in contrast to goal *states*)
  - find a sequence of actions that will take you from the start state to some state satisfying the goal conditions (note: these **are** goal states, but specified differently)
- What's different?
  - states are complex entities (possible worlds)
  - actions tend to affect only certain facts
- End result: planning algorithms look a little different than search algorithms (usually)

A Planning Problem



Robot Example: Domain

- Predicates:
  - loc(X) - location of robot is X
  - adj(X,Y) - locations X,Y adjacent
  - rhk - robot has keys
  - cm - coffee is made
  - chc - craig has coffee (or maybe hc(P))
  - rhc - robot has coffee (or maybe carrying(O))
  - lt - lab is tidy
- Derived predicates possibly:

  accessible(X,Y) ← adj(X,Y) & rhk.

  accessible(X,Y) ← adj(X,Y) & unlck(X).

- Constants:
  - The locations: off, hall, lab, mr, cr
  - Domain objects? coffee, mail, keys, etc,
  - other things?

States of the World

- A state is an assignment of truth values to all (relevant) atoms
  - e.g., think of it as an "interpretation" or *complete* KB
  - it is a way the world could be
- Formally, given a finite number of predicates and domain objects, a **world state** is any assignment of truth values to *all ground atoms*
  - e.g., True: loc(off), adj(cr,hall),…, rhc, rhk, etc…

    False: loc(hall), loc(mr), …, adj(off,hall), … cm, chc, etc.

Robot Example: Actions

- What actions might we consider?
  - move(X,Y), getkeys, dropkeys, makecoffee, grabcoffee or grab(X) ?, tidylab, etc…
- Almost all actions have **preconditions**
  - can't apply an action at state S unless all preconditions are satisfied
  - move(X,Y) requires accessible(X,Y)
  - givecoffee at S requires loc(off), rhc, etc.
- Actions change the state of the world
  - move(X,Y) makes loc(X) false, loc(Y) true
  - givecoffee makes rhc false, chc true
  - these actions affect no other facts!

Neighbor Representation

- For any state S, action A, we could specify the state resulting from applying S at A; but…
- How many states?
  - if k ground atoms, 2^k states (so far our trivial robot domain has about 40 ground atoms!)
- Action effects have a lot of structure
  - actions tend to affect only a few atoms, most unaffected
- But if we decided to do this (and ignore these issues), we could apply our favorite search algorithms to this planning problem!

Planning Basics

- Almost all work in planning relies on some basic principles
  a) use specific state representations that are easy to manipulate
  b) use specific action representations that exploit the fact that actions affect only a few atoms generally
  c) devise planning algorithms that exploit the “locality” of actions and the fact that we’re concerned with only a *few* goal propositions
- We’ll look at each of these in turn

State Representation

- A world state is just like a KB, but we can't just write a bunch of facts/rules to assert what's true
  - we'll need to look at a bunch of worlds, not just one
  - and we can't retract/assert things in KB (easily)
- So we need a *state representation*
- Consider the state:
  - True: loc(c), adj(o,m), adj(m,o), adj(o,l), adj(l,o), …, rhk, cm, lck(c), lck(l), …
  - False: loc(o), loc(l), loc(m), loc(h), adj(c,o), etc…, chc, rhc, lck(o), lck(h), …

Explicit World Represn'tion (EWR)

- EWR the simplest possible state representation
  - Keep two lists of ground atoms: *true list*, *false list*
  - or use negation and have one list:
  - e.g., [loc(c), neg(loc(o)), neg(loc(m)), rhk, neg(chc)…]
- Difficulties:
  - list has one entry for each ground atom in language
  - generally, there are far more negative literals than positive literals
  - e.g., adj(X,Y) false for many more pairs than true; loc(X) false for all but one location;
  - consider airline planner with relation flight(City1,City2,FltNum): many false instances!!!

Closed World Representation (CWR)

- CWR makes the closed world assumption
  - You specify every atom that is true (e.g., in a list)
  - If atom is not in list, it is assumed false
  - A state is a list of *positive ground facts*
  - e.g., [loc(c), rhk, cm, adj(o,m), adj(m,o)…]
- Give state S (list), when is literal L true at S?

```
holds(Atom,State) :- member(Atom,State).

holds(neg(Atom), State) :- not(member(Atom,State)).
```

Meta-Interpreter

- The *holds* predicate: a (simple) *meta-interpreter*
  - the list State is our own little ‘KB’, built into a list
  - the *holds* predicate asks a “query” of this ‘KB’: is the (pos’tv or neg’tv) Fact true in State (our mini ‘KB’)
  - we allow *holds* to ask queries about negative literals (and we exploit Prolog’s negation-as-failure mechanism to implement the CWA)
  - Thus we’ve implemented a (trivial) Prolog-like query interpreter in Prolog
- Note: Elements in State are “atoms” in our domain language, but are just *terms* in Prolog
  - allows us to treat a state (‘KB’) as an object in Prolog
  - why is this important?

Derived Relations in CWR (CWR-D)

- Certain facts derivable from other facts
  - e.g., the accessible(X,Y) relation
- Don't want these explicitly represented in state
  - they are redundant (can be derived from others)
  - they complicate action rep'n (potential inconsistency)
  - e.g., robot drops keys (-rhk): what happens to accsbl?
- So we can separate *basic* relations in our domain from *derived* relations
  - only ground atoms of *basic* type allowed in state
- For each domain "predicate", must specify type

Derived Relation: Example

- Basic relations specified as follows:

```
baseRel(rhk).
baseRel(loc(X)).
baseRel(lck(X)).   etc…
```

- Derived relations include how they are derived:

```
derivedRel( accessible(X), [ neg(lck(X)) ] ).

derivedRel( accessible(X), [ lck(X),  rhk] ).
```

Like Prolog rules:
accessible(X) :- not(lck(X))
accessible(X) :- lck(X), rhk.

A Meta-Interpreter for CWR-D

```
holds(Fact, State) :- baseRel(Fact),  member(Fact,State).

holds(neg(Fact), State) :- baseRel(Fact),
                           not(member(Fact, State)). *

holds(Fact, State) :- derivedRel(Fact, BodyList),
                      holdsAll(BodyList, State).

holds(neg(Fact), State) :- derivedRel(Fact, BodyList),
                           not(holds(Fact, State). *
```

Notes:
- holdsAll just applies holds to every atom in the "body" list
- Can replace both * with: holds(neg(F),S) :- not(holds(F,S)).

Static Facts

- Certain facts never change as actions performed
  - e.g., adj(X,Y) in our example is a **static fact**
- No need to carry these around in state list
- Static facts can be represented as derived relations with empty "bodies"

```
derivedRel( adj(o,m), [ ] ).
derivedRel( adj(o,l), [ ] ).   etc...
```

or possibly:

```
derivedRel( adj(X,Y), [ ])  :-  neighbor(X,Y).
```

with neighbor specified in Prolog KB.

The CWR-D Meta-Interpreter

- *holds(F,S)* for CWR-D looks a lot more like Prolog. This metainterpreter allows us to assert *facts* in the *state* (thus varying the “fact” part of the KB) and assert *rules* (via derived relations) that do not vary. The holds predicate essentially tries to find a proof for the query F using the KB S (and the rules).
- We can thus have many different “KBs” at once
- For more on meta-interpreters, see Ch.6 of the text (if you’re interested).

Action Representation

- Actions are **concrete** things an agent can do
  - e.g., move(l,o), tidylab, getkeys, etc.
- *Action schema*: a collection of “related” actions
  - e.g., move(X,Y) is a schema
  - ground instances of a schema are true actions
- As discussed, actions cause state changes
- A *neighbor* of state *S* is any state *S’* such that an action *A* can be applied at *S* and results in *S’*
- So we need to describe (and represent):
  - when *A* can be applied at *S*; i.e., *A*’s *preconditions*
  - the effect *A* has on state *S*; i.e., *A*’s *effects*

Precondition Representation

- Precondition for action A: a logical condition that must hold at S for A to be applicable
  - precondition representation: a list of ground literals
  - for an action schema, preconditions literals may include variables mentioned in the action schema

Action (schema)	Preconditions
move(X,Y)	loc(X), accessible(Y), adj(X,Y)
tidylab	loc(l), neg(tidy(l))
or: tidy(X)	*loc(X), neg(tidy(X))*
getkeys	loc(o), neg(rhk) ...

- Assert *precond* predicate (one fact per action)
  - e.g., precond(move(X,Y), [loc(X), acc(Y), adj(X,Y)]).

Effect Representation

- Actions affect only a few world facts, so we're best off describing only *what changes* when an action is performed
  - assume unmentioned facts are unaffected (like CWA)
- Effects are either *positive* or *negative*
  - action makes an atom true or false
- In CWR or CWR-D state rep'n:
  - we should *add* positive effects to the state list
  - we should *delete* negative effects from the state list

STRIPS Action Representation

- STRIPS representation uses add and delete lists (and precondition lists) to represent an actions
  - addlist(givecoffee, [chc, craighappy]).
  - deletelist(givecoffee, [rhc]).
  - addlist(move(X,Y), [loc(Y)]).
  - deletelist(move(X,Y), [loc(X)]).
- Important: if you use CWR-D relation, add and delete lists should not include derived relations
  - changes to derived facts due to changes in base facts
- STRIPS: Stanford Res. Inst. (SRI) Planning System (1970-72)

Domain Descriptions

- A **domain description**: a specification of the actions (and language) required in your specific planning domain
  - for each action: precondition, addlist, deletelist statements should be asserted in KB
  - if CWR-D, state which relations are basic, derived
  - might have a list of all action names (schemas) in KB

Neighbors (Result of Action)

- Exercises
  - define predicate results(Action,State,NewState)
    - given State in CWR-D, Action, what is resulting state
    - if Action preconditions not satisfied at State, fails
  - define neighbor predicate for STRIPS/CWR-D
    - nb(State,NBList) : S' is a neighbor of S if exists an A' s.t. preconds of A' satisfied at S, and results(A',S,S')
    - assume actionList([move(X,Y), givecof, …]) in KB

Planning Problems

- Given a domain description
  - preconditions, addlist, deletelist, actionlist, derivedRel, baseRel, all specified in KB
- Given an initial state s_0 in CWR or CWR-D
  - initial state is just a list of positive ground atoms
- Given a goal set G
  - G is a list of (post'v or negt'v) literals to be made true
- Find a sequence of actions (a ***plan***) $a_1, a_2, \ldots, a_n$ s.t. applying that sequence to s_0 leads to a state s_n satisfying all goal literals
  - results(a_1,s_0,s_1), results(a_2,s_1,s_2), … results(a_n,s_{n-1},s_n) and holdsall(G,s_n) are all true

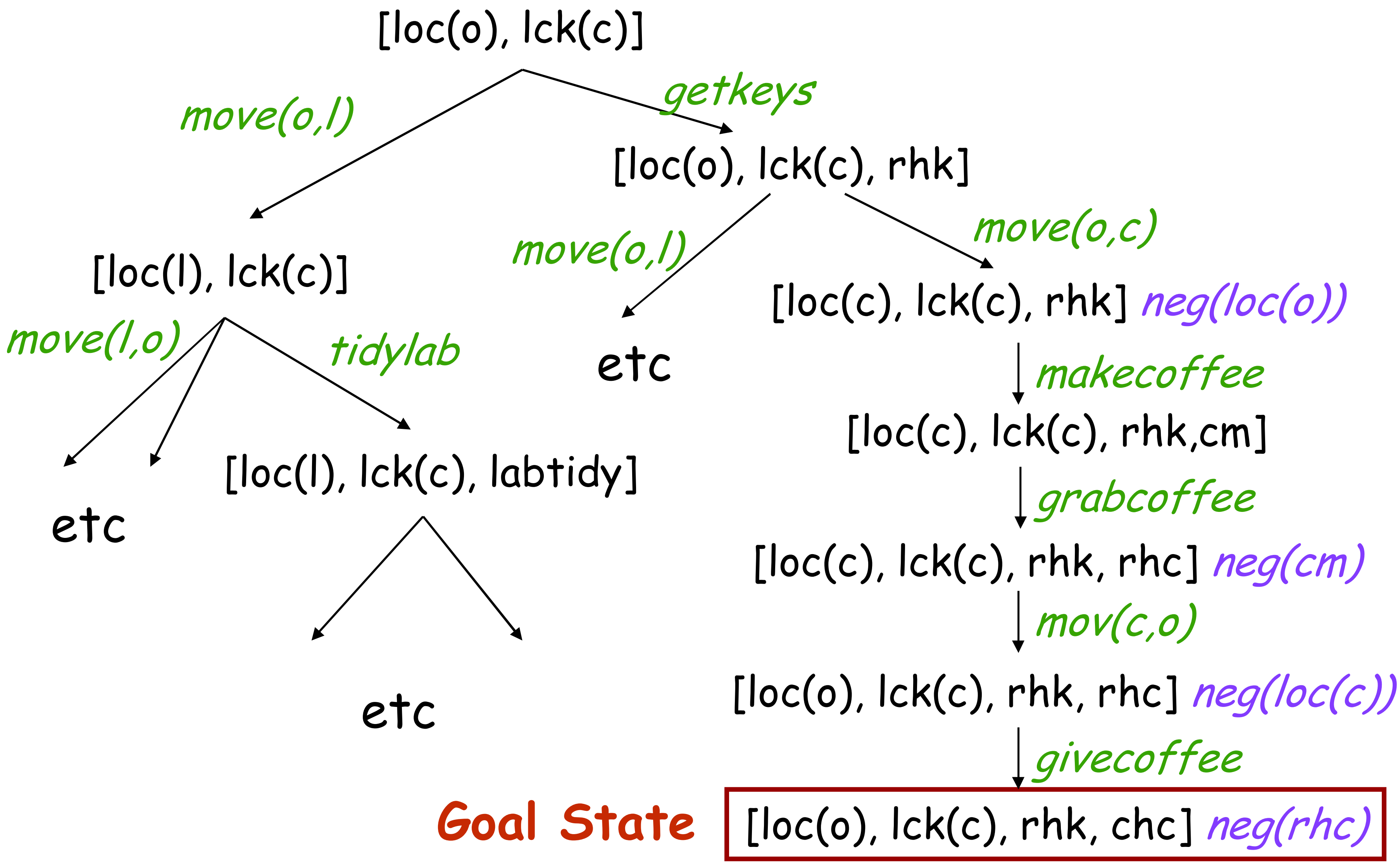
Example Planning Problem

- Initial State (CWR-D): [loc(o), lck(c)]
  - unmentioned: neg(cm), neg(rhk), neg(chc), etc…
  - unmentioned: adj(o,l), adj(o,c), neg(adj(l,c)), accessible(l), neg(accessible(c)), etc…
- Goal: [loc(o), chc]
  - note: this is *not a state* in CWR-D, it is just a set of conditions we want true

Planning as Search

- Given this formulation of neighbors, goals, it is easy to see how planning can be solved using any standard search algorithm
  - initial state s_0 is root of search tree
  - for any s in the tree, it's children (neighbors) are determined by nb (states reachable using one action)
  - is_goal(S) determined by holdsAll(G,S)
- Your favorite search algorithm builds a sequence of states from start to (some) goal state
  - note: you don't want state sequence, but an *action* sequence (so modify nb predicate, search algorithm to extract actions you took to reach these states)

Partial Search Tree in Example



Difficulties with Generic Search

- Search tree is generally quite large
  - our branching factor roughly 3; solution depth is 7
  - BFS: about 3300 node expansions
  - DFS is easily lead astray in this problem
- Goal provides no guidance
  - path selection uninfluenced by goal (e.g., why even consider going to lab as the first step?)
  - suggests we need heuristics (people do work on this)
- Can we get by without heuristics?
  - heuristics generally handcrafted for *specific* domains
  - want a *general purpose planner* that is directed toward the goal somehow

Goal-Influenced Planning

- In our example, intuitively the action tidylab is irrelevant to the literals in the goal list
  - it doesn't achieve a goal literal, it doesn't achieve the precondition of any action that achieves a goal literal, etc…
- Most planning algorithms work backward from the goal list. The basic idea/intuition:
  - find an action a_n that achieves a goal literal
  - then find an action a_{n-1} that acheives one of a_n's preconditions, etc…
- Focuses attention on goal literals or things that need to be made true to achieve goal literals
  - actions that have some (indirect) impact on the goal

STRIPS Planner

- STRIPS one of earliest AI planning algorithms
  - note: distinguish STRIPS action repn (a good idea) from STRIPS planning algorithm (not so great)
  - focuses on actions relevant to your goals (so more informed than blind search)
- Basically a divide-and-conquer approach to “solving a goal list”
  - tries to find independent plans for individual subgoals and then pieces these plans together
  - recursively tries to achieve necessary preconditions

STRIPS: Intuitive Sketch

- Given a goal list [g1, g2], initial state s0
  - Select a goal to achieve, let's say g1
  - Find a sequence of actions that achieves g1 from s0, let's say a1, a2, a3
  - Compute state s1 that results from applying a1,a2,a3 to s0: s1 = result(a3,result(a2,result(a1,s0)))
  - Select a remaining goal (here only g2 remains)
  - Find a sequence that achieves g2 from s1, let's say b1, b2, b3
  - Return solution: Plan P = a1,a2,a3,b1,b2,b3

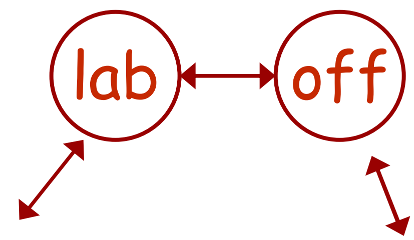
How to Solve a Subgoal

- How to find sequence to achieve g1 from s0?
  - find an action (say, a3) that achieves g1 (so g1 must be an effect of a3)
  - to ensure we can execute a3, we make all of its preconditions new goals, or *subgoals* (say p1, p2)
  - recursively call STRIPS on subgoals [p1,p2] with start state s0, to get sequence a1, a2 that acheives them
  - sticking a3 on the end of a1,a2 ensures achievement of g1
- Process terminates when all subgoals true at s0

A Simple Example of STRIPS

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

Goal: loc(o), labtidy



Goal: loc(o),labtidy

1. Pick subgoal labtidy
   - Action tidylab achieves it
   - Subgoals: loc(l)
     - Pick subgoal loc(l)
       - Action mov(o,l) achieves it
         - Subgoals: loc(o), rhk
           - Pick subgoal rhk
             - Action getkeys achieves it
               - Subgoal: loc(o)
                 - Pick subgoal loc(o)
                   - True in s0
           - Plan: [getkeys] goes from s0 to s1
           - Pick subgoal loc(o)
             - True in s1
     - Plan: [getkeys,mov(o,l)] goes from s0 to s2
   - Plan: [getk,mov(o,l),tidylb] goes from s0 to s3 and achieves first subgoal labtidy

Continued…

2. Pick subgoal loc(o)
   - Action mov(l,o) achieves it
   - Subgoals: loc(l)
     - Pick subgoal loc(l)
       - True in s3
   - Plan: [gk,mov(o,l),tl,mov(l,o)]