CSE 3402: Intro to Artificial Intelligence Planning

- Readings: Sections 11.1, 11.2, and 11.4

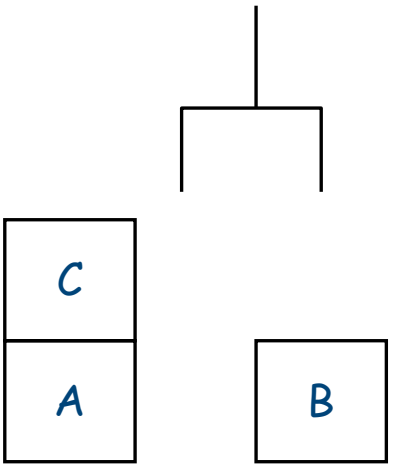
CWA

- "Classical Planning". No incomplete or uncertain knowledge.
- Use the "Closed World Assumption" in our knowledge representation and reasoning.
  - The Knowledge base used to represent a state of the world is a **list of positive ground atomic facts**.
  - CWA is the assumption that
    - if a ground atomic fact is not in our list of "known" facts, its negation must be true.
    - the constants mentioned in KB are all the domain objects.

CWA

- CWA makes our knowledge base much like a database: if employed(John,CIBC) is not in the database, we conclude that ¬employed(John, CIBC) is true.

CWA Example



KB = {handempty
clear(c), clear(b),
on(c,a),
ontable(a), ontable(b)}

1. clear(c) ∧ clear(b)?
2. ¬on(b,c)?
3. on(a,c) ∨ on(b,c)?
4. ∃X.on(X,c)? (D = {a,b,c})
5. ∀X.ontable(X)
   → X = a ∨ X = b?

Querying a Closed World KB

- With the CWA, we can evaluate the truth or falsity of arbitrarily complex first-order formulas.
- This process is very similar to query evaluation in databases.
- Just as databases are useful, so are CW KB's.

Querying A CW KB

Query(F, KB) /*return whether or not KB |= F */

if F is atomic
  return(F ∈ KB)

Querying A CW KB

if $F = F_1 \wedge F_2$
  return(Query(F_1) && Query(F_2))

if $F = F_1 \vee F_2$
  return(Query(F_1) || Query(F_2))

if $F = \neg F_1$
  return(! Query(F_1))

if $F = F_1 \rightarrow F_2$
  return(!Query(F_1) || Query(F_2))

Querying A CW KB

if $F = \exists X.F_1$
  for each constant $c \in KB$
    if (Query(F_1{X=c}))
      return(true)
  return(false).

if $F = \forall X.F_1$
  for each constant $c \in KB$
    if (!Query(F_1{X=c}))
      return(false)
  return(true).

Querying A CW KB

Guarded quantification (for efficiency).

if $F = \forall X.F_1$
  for each constant $c \in KB$
    if (!Query(F_1{X=c}))
      return(false)
  return(true).

E.g., consider checking
  $\forall$ X. apple(x) $\rightarrow$ sweet(x)

we already know that the formula is true for all "non-apples"

Querying A CW KB

Guarded quantification (for efficiency).

$\forall$ X:[p(X)] F_1 $\leftrightarrow$ $\forall$ X: p(X) $\rightarrow$ F_1
  for each constant c s.t. p(c)
    if (!Query(F_1{X=c}))
      return(false)
  return(true).

$\exists$ X:[p(X)]F_1 $\leftrightarrow$ $\exists$ X: p(X) $\wedge$ F_1
 for each constant c s.t. p(c)
    if (Query(F_1{X=c}))
      return(true)
  return(false).

STRIPS representation.

- STRIPS (Stanford Research Institute Problem Solver.) is a way of representing actions.
- Actions are modeled as ways of modifying the world.
  - since the world is represented as a CW-KB, a STRIPS action represents a way of updating the CW-KB.
  - Now actions yield new KB's, describing the new world—the world as it is once the action has been executed.

Sequences of Worlds

- In the situation calculus where in one logical sentence we could refer to two different situations at the same time.
  - $on(a,b,s_0) \wedge \neg on(a,b,s_1)$
- In STRIPS, we would have two separate CW-KB's. One representing the initial state, and another one representing the next state (much like search where each state was represented in a separate data structure).

STRIPS Actions

- STRIPS represents actions using 3 lists.
  1. A list of action preconditions.
  2. A list of action add effects.
  3. A list of action delete effects.
- These lists contain variables, so that we can represent a whole class of actions with one specification.
- Each ground instantiation of the variables yields a specific action.

STRIPS Actions: Example

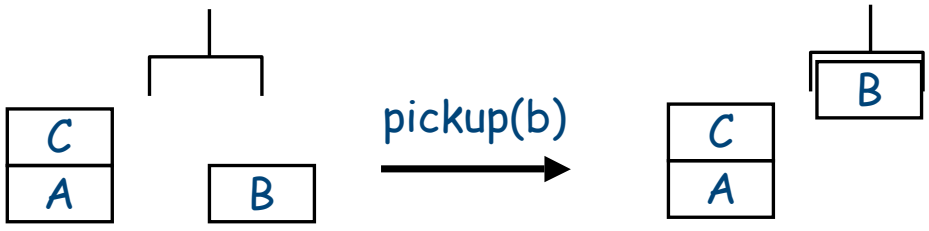
pickup(X):
Pre: {handempty, clear(X), ontable(X)}
Adds: {holding(X)}
Dels: {handempty, clear(X), ontable(X)}

"pickup(X)" is called a STRIPS operator.
a particular instance e.g.
"pickup(a)" is called an action.

Operation of a STRIPS action.

- For a particular STRIPS action (ground instance) to be applicable to a state (a CW-KB)
  - every fact in its precondition list must be true in KB.
    - This amounts to testing membership since we have only atomic facts in the precondition list.
- If the action is applicable, the new state is generated by
  - removing all facts in Dels from KB, then
  - adding all facts in Adds to KB.

Operation of a Strips Action: Example



pre = {handmpty, clear(b), ontable(b)}

add = {holding(b)}

del = {handmpty, clear(b), ontable(b)}

KB = {handempty
clear(c), clear(b),
on(c,a),
ontable(a), ontable(b)}

KB = { holding(b),
clear(c),
on(c,a),
ontable(a)}

STRIPS Blocks World Operators.

- pickup(X)
  Pre: {clear(X), ontable(X), handempty}
  Add: {holding(X)}
  Del: {clear(X), ontable(X), handempty}
- putdown(X)
  Pre: {holding(X)}
  Add: {clear(X), ontable(X), handempty}
  Del: {holding(X)}

STRIPS Blocks World Operators.

- unstack(X,Y)
  Pre: {clear(X), on(X,Y), handempty}
  Add: {holding(X), clear(Y)}
  Del: {clear(X), on(X,Y), handempty}
- stack(X,Y)
  Pre: {holding(X),clear(Y)}
  Add: {on(X,Y), handempty, clear(X)}
  Del: {holding(X),clear(Y)}

STRIPS has no Conditional Effects

- putdown(X)
  Pre: {holding(X)}
  Add: {clear(X), ontable(X), handempty}
  Del: {holding(X)}
- stack(X,Y)
  Pre: {holding(X),clear(Y)}
  Add: {on(X,Y), handempty, clear(X)}
  Del: {holding(X),clear(Y)}

- The table has infinite space, so it is always clear. If we "stack(X,Y)" if Y=Table we cannot delete clear(Table), but if Y is an ordinary block "c" we must delete clear(c).

Conditional Effects

- Since STRIPS has no conditional effects, we must sometimes utilize extra actions: one for each type of condition.
  - We embed the condition in the precondition, and then alter the effects accordingly.

Other Example Domains

- 8 Puzzle as a planning problem
  - A constant representing each position, P1,…,P9

P1	P2	P3
P4	P5	P6
P7	P8	P9

  - A constant for each tile. B,T1, …, T8.

8-Puzzle

- at(T,P) tile T is at position P.

1	2	5
7	8	
6	4	3

at(T1,P1), at(T2,P2), at(T5,P3), …

- adjacent(P1,P2) P1 is next to P2 (i.e., we can slide the blank from P1 to P2 in one move.
  - adjacent(P5,P2), adjacent(P5,P8), …

8-Puzzle

slide(T,X,Y)

Pre: {at(T,X), at(B,Y), adjacent(X,Y)}

Add: {at(B,X), at(T,Y)}

Del: {at(T,X), at(B,Y)}

at(T1,P1), at(T5,P3),
at(T8,P5), at(B,P6), …,

1	2	5
7	8	
6	4	3

slide(T8,P5,P6)

at(T1,P1), at(T5,P3),
at(B,P5), at(T8,P6), …,

1	2	5
7		8
6	4	3

Elevator Control

Figure 1: A Miconic-10™ keypad allows passengers to enter their destination before they enter the elevator. A display informs the passenger about the elevator that will offer the most suitable transport.

Elevator Control

- Schindler Lifts.
  - Central panel to enter your elevator request.
  - Your request is scheduled and an elevator is assigned to you.
  - You can't travel with someone going to a secure floor, emergency travel has priority, etc.
- Modeled as a planning problem and fielded in one of Schindler's high end elevators.

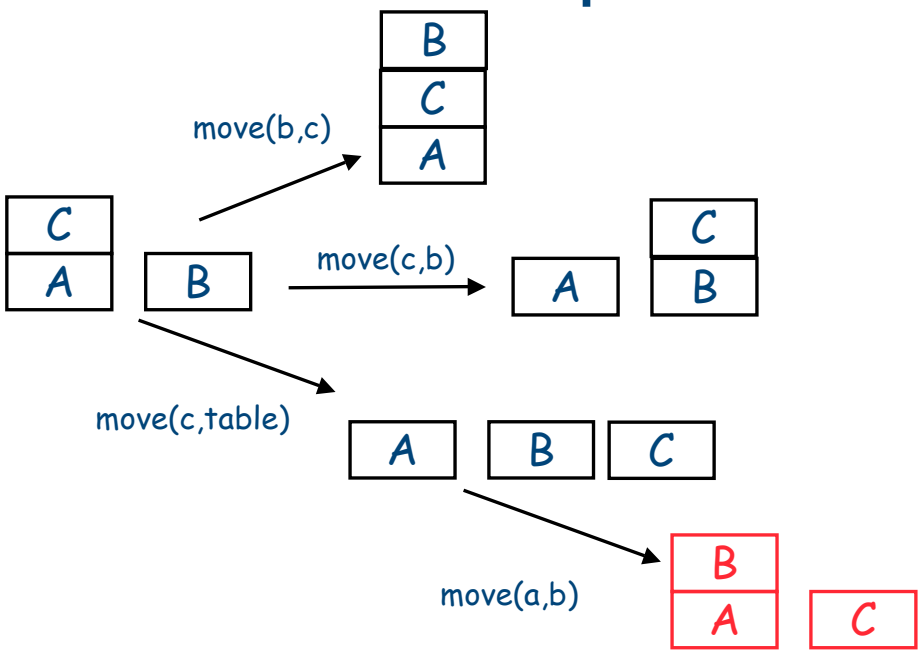
Planning as a Search Problem

- Given a CW-KB representing the initial state, a set of STRIPS or ADL (Action Description Language) operators, and a goal condition we want to achieve (specified either as a conjunction of facts, or as a formula)
  - The planning problem is determine a sequence of actions that when applied to the initial CW-KB yield an updated CW-KB which satisfies the goal.

Planning As Search

- This can be treated as a search problem.
  - The initial CW-KB is the initial state.
  - The actions are operators mapping a state (a CW-KB) to a new state (an updated CW-KB).
  - The goal is satisfied by any state (CW-KB) that satisfies the goal.

Example.



Problems

- Search tree is generally quite large
  - randomly reconfiguring 9 blocks takes thousands of CPU seconds.
- The representation suggests some structure. Each action only affects a small set of facts, actions depend on each other via their preconditions.
- Planning algorithms are designed to take advantage of the special nature of the representation.

Planning

- We will look at 1 technique
- Relaxed Plan heuristics used with heuristic search.

Reachability Analysis.

- The idea is to consider what happens if we ignore the delete lists of actions.
- This is yields a “relaxed problem” that can produce a useful heuristic estimate.

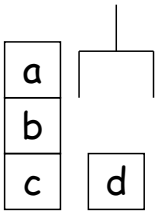
Reachability Analysis

- In the relaxed problem actions add new facts, but never delete facts.
- Then we can do reachability analysis, which is much simpler than searching for a solution.

Reachability

- We start with the initial state S_0.
- We alternate between state and action layers.
- We find all actions whose preconditions are contained in S_0. These actions comprise the first action layer A_0.
- The next state layer consists of all of S_0 as well as the adds of all of the actions in A_0.
- In general
  - A_i is the set of actions whose preconditions are contained in S_i.
  - S_{i+1} is S_i union the add lists of all of the actions in A_i.

Example

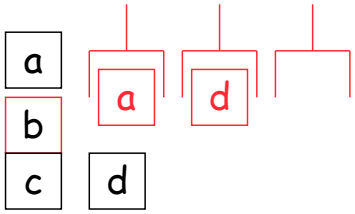


on(a,b),
on(b,c),
ontable(c),
ontable(d),
clear(a),
clear(d),
handempty

S_0

unstack(a,b)
pickup(d)

A_0



on(a,b),
on(b,c),
ontable(c),
ontable(d),
clear(a),
handempty,
clear(d),
holding(a),
clear(b),
holding(d)

S_1

this is not a state!

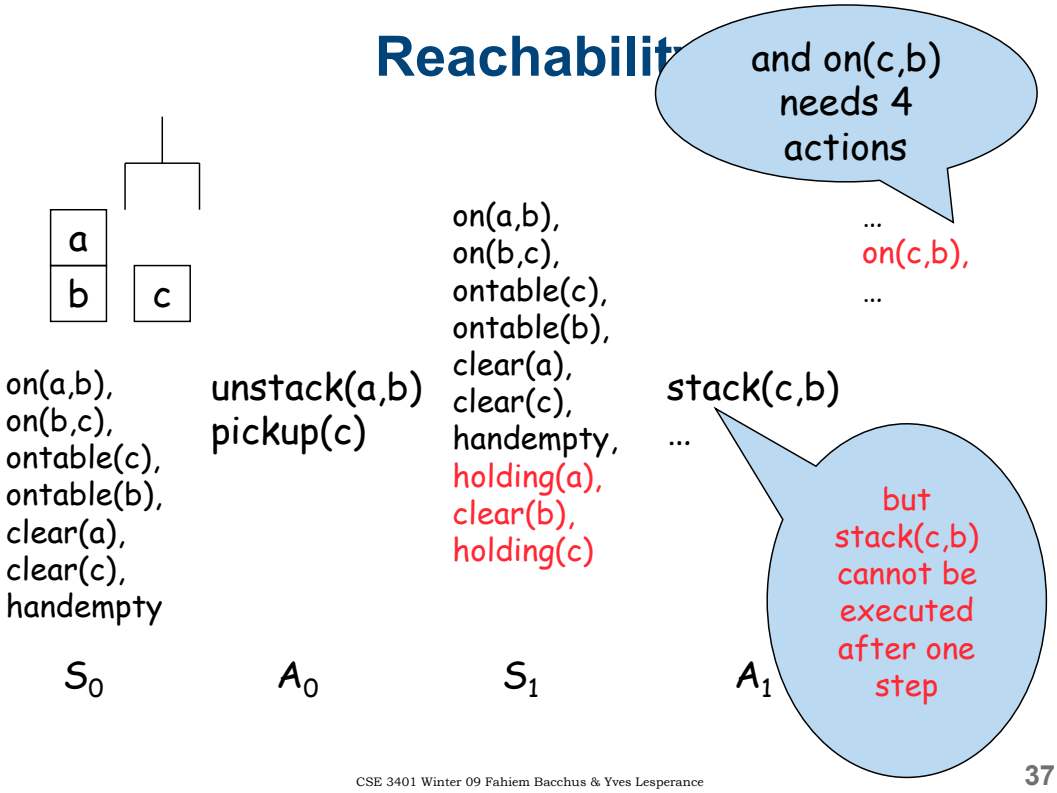
Example

on(a,b),	putdown(a),
on(b,c),	putdown(d),
ontable(c),	stack(a,b),
ontable(d),	stack(a,a),
clear(a),	stack(d,b),
clear(d),	stack(d,a),
handempty,	pickup(d),
holding(a),	…
clear(b),	unstack(b,c)
holding(d)	…
S_1	A_1

Reachabilty

- We continue until the goal G is contained in the state layer, or until the state layer no longer changes.
- Intuitively, the actions at level A_i are the actions that could be executed at the i-th step of some plan, and the facts in level S_i are the facts that could be made true after some i-1 step plan.
- Some of the actions/facts have this property. But not all!

Reachabilit



Heuristics from Reachability Analysis

Grow the levels until the goal is contained in the final state level S[K].

- If the state level stops changing and the goal is not present. The goal is unachievable. (The goal is a set of positive facts, and in STRIPS all preconditions are positive facts).

- Now do the following

Heuristics from Reachability Analysis

CountActions(G,S_K):

/* Compute the number of actions contained in a relaxed plan achieving the goal. */

- Split G into facts in S_{K-1} and elements in S_K only. These sets are the previously achieved and just achieved parts of G.
- Find a minimal set of actions A whose add-effects cover the just achieved part of G. (The set contains no redundant actions, but it might not be the minimum sized set.)
- Replace the just achieved part of G with the preconditions of A, call this updated G, NewG.
- Now return CountAction(NewG,S_{K-1}) + number of actions needed to cover the just achieved part of G.

Example

legend: [pre]act[add]

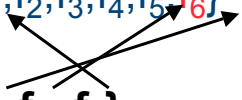
$S_0 = \{f_1, f_2, f_3\}$

$A_0 = \{[f_1]a_1[f_4],\ [f_2]a_2[f_5]\}$

$S_1 = \{f_1,f_2,f_3,f_4,f_5\}$

$A_1 = \{[f_2,f_4,f_5]a_3[f_6]\}$

$\mathbf{S_2} = \{f_1,f_2,f_3,f_4,f_5,f_6\}$



$G = \{f_6,f_5, f_1\}$

We split G into G_P and G_N:

CountActs(G,S_2)

$G_P = \{f_5, f_1\}$ //already in S1

$G_N = \{f_6\}$ //New in S2

$A = \{a_3\}$ //adds all in G_N

//the new goal: $G_P \cup$ Pre(A)

$G_1 = \{f_5,f_1,f_2,f_4\}$

Return

1 + CountActs(G_1,S_1)

Example

Now, we are at level S1

$S_0 = \{f_1, f_2, f_3\}$

$A_0 = \{[f_1]a_1[f_4],\ [f_2]a_2[f_5]\}$

$\mathbf{S_1} = \{f_1,f_2,f_3,f_4,f_5\}$

$A_1 = \{[f_2,f_4,f_5]a_3[f_6]\}$

$S_2 = \{f_1,f_2,f_3,f_4,f_5,f_6\}$

$G_1 = \{f_5,f_1,f_2,f_4\}$

We split G1 into G_P and G_N:

CountActs(G_1,S_1)

$G_P = \{f_1,f_2\}$ //already in S0

$G_N = \{f_4,f_5\}$ //New in S1

$A = \{a_1,a_2\}$ //adds all in G_N

//the new goal: $G_P \cup$ Pre(A)

$G_2 = \{f_1,f_2\}$

Return

2 + CountActs(G_2,S_0)

= 2 + 0

So, in total CountActs(G,S2)=1+2=3

Using the Heuristic

- To use CountActions as a heuristic, we build a layered structure from a state S that reaches the goal.
- Then we CountActions to see how many actions are required in a relaxed plan.
- We use this count as our heuristic estimate of the distance of S to the goal.
- This heuristic tends to work better as a best-first search, i.e., when the cost of getting to the current state is ignored.

Admissibility

- An optimal length plan in the relaxed problem (actions have no deletes) will be a lower bound on the optimal length of a plan in the real problem.
- However, CountActions **does NOT compute** the length of the optimal relaxed plan.
- The choice of which *action set* to use to achieve G_P ("just achieved part of G") is not necessarily optimal.
- In fact it is NP-Hard to compute the optimal length plan even in the relaxed plan space.
- So CountActions will not be admissible.

Empirically

- However, empirically refinements of CountActions performs very well on a number of sample planning domains.

GraphPlan

- GraphPlan is an approach to planning that is built on ideas similar to “reachability”. But the approach is not heuristic: delete effects are not ignored.
- The performance is not at good as heuristic search, but GraphPlan can be generalized to other types of planning, e.g., finding optimal plans, planning with sensing, etc.

Graphplan

- Operates in two phases.
  - Phase I. Guess a “concurrent” plan length k, then build a leveled graph with k alternating layers.
  - Phase II. Search this leveled graph for a plan. If no plan is found, return to phase I and build a bigger leveled graph with k+1 alternating layers. The final plan, if found, consists of a sequence of sets of actions

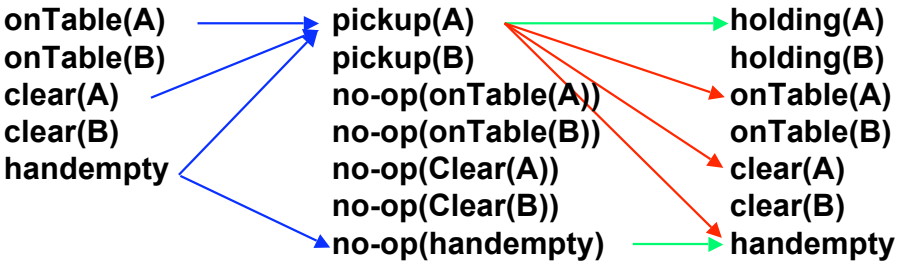
    $\{a^1_1, a^2_1, \ldots\} \rightarrow \{a^1_2, a^2_2, \ldots\} \rightarrow \{a^1_3, a^2_3, \ldots\} \rightarrow \ldots$

    The plan is “concurrent” in the sense that at stage I, all actions in the i-th set are executed in parallel.

Graphplan

- The leveled graph alternates between levels containing propositional nodes and levels containing action nodes. (Similar to the reachability graph).
- Three types of edges: precondition-edges, add-edges, and delete-edges.

GraphPlan Level Graph



Initial state

Only the propositions true in the initial state.

Possible actions

Only the actions whose preconditions are in the previous level.

Also have no-ops for capturing non-changes.

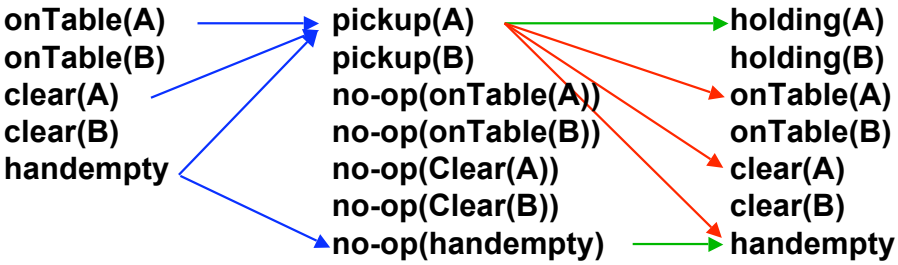
All propositions added by actions in previous level

Precondition

Delete

Add

GraphPlan Level Graph



Level S_0 contains all facts true in the initial state.

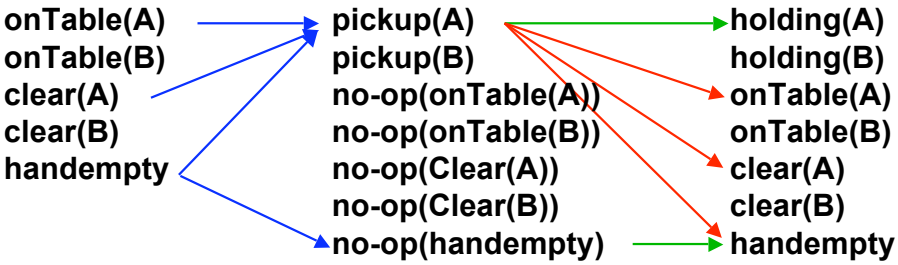
Level A_0 contains all actions whose preconditions are true in S_0. Included in the set of actions are no-ops. One no-op for every ground atomic fact. The precondition of the no-op is its fact, its add effect is its fact.

…

Level S_i contains all facts that are added by actions at level A_{i-1}

Level A_i contains all actions whose preconditions are true in S_i

GraphPlan Mutexes.



In addition to the facts/actions. GraphPlan also computes and adds mutexes to the graph.

Mutexes are edges between two labels, indicating that these two labels cannot be true at the same time.

Mutexes are added as we construct each layer, and in fact alter the set of labels the eventually appear in a layer.

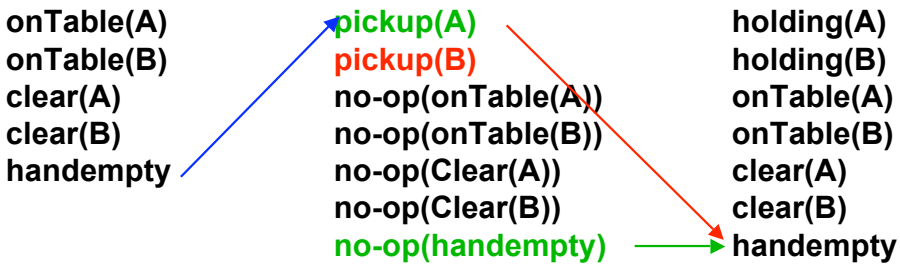
Mutexes

- A mutex between two actions a_1 and a_2 in the same layer A_i, means that a_1 and a_2 cannot be executed simultaneously (in parallel) at the i^{th} step of a concurrent plan.
- A mutex between two facts F_1 and F_2 in the same state layer S_i, means that F_1 and F_2 cannot be be simultaneously true after i stages of parallel action execution.

Mutexes

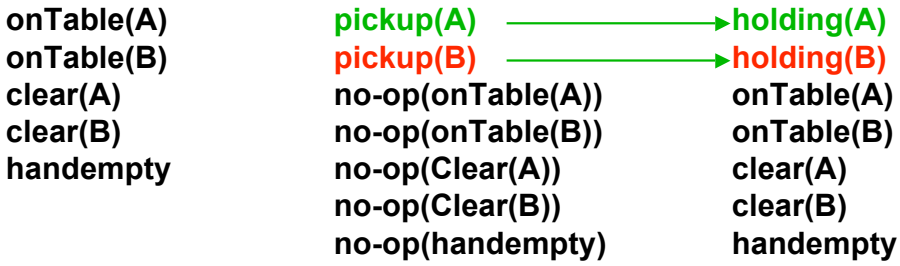
- It is not possible to compute all mutexes.
  - This is as hard as solving the planning problem, and we want to perform mutex computation as a precursor to solving a planning instance.
- However, we can quickly compute a subset of the set of all mutexes. Although incomplete these mutexes are still very useful.
  - This is what GraphPlan does.

Mutexes



- Two actions are mutex if either action deletes a precondition or add effect of another.
- Note no-ops participate in mutexes.
  - Intuitively these actions have to be sequenced—they can't be executed in parallel

Mutexes



- Two propositions *p* and *q* are mutex if all actions adding *p* are mutex of all actions adding *q.*
  - Must look at all pairs of actions that add *p* and *q*.
  - Intuitively, can't achieve p and q together at this stage because we can't concurrently execute achieving actions for them at the previous stage.

Mutexes

holding(A)	→	putdown(A)
holding(B)	→	putdown(B)
onTable(A)		no-op(onTable(A))
onTable(B)		no-op(onTable(B))
clear(A)		no-op(Clear(A))
clear(B)		no-op(Clear(B))
handempty		no-op(handempty)

- Two actions are mutex if two of their preconditions are mutex.
  - Intuitively, we can't execute these two actions concurrently at this stage because their preconditions can't simultaneously hold at the previous stage.

How Mutexes affect the level graph.

1. Two actions are mutex if either action deletes a precondition or add effect of another
2. Two propositions *p* and *q* are mutex if all actions adding *p* are mutex of all actions adding *q*
3. Two actions are mutex if two of their preconditions are mutex

- We compute mutexes as we add levels.
  - S_0 is set of facts true in initial state. (Contains no mutexes).
  - A_0 is set of actions whose preconditions are true in S_0.
    - Mark as mutex any action pair where one deletes a precondition or add effect of the other.
  - S_1 is set of facts added by actions at level A_0.
    - Mark as mutex any pair of facts p and q if all actions adding p are mutex with all actions adding q.
  - A_1 is set of actions whose preconditions are not mutex at S_1.
    - Mark as mutex any action pair with preconditions that are mutex in S_1, or where one deleted a precondition or add effect of the other.

How Mutexes affect the level graph.

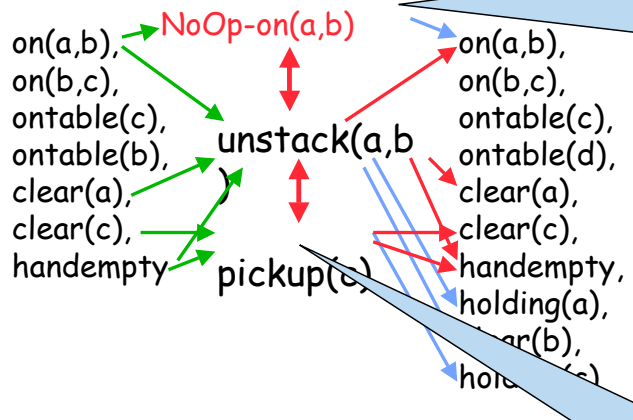
1. Two actions are mutex if either action deletes a precondition or add effect of another
2. Two propositions *p* and *q* are mutex if all actions adding *p* are mutex of all actions adding *q*
3. Two actions are mutex if two of their preconditions are mutex

-
  - …
  - S_i is set of facts added by actions in level A_{i-1}
    - Mark as mutex all facts satisfying 2 (where we look at the action mutexes of A_{i-1} is set of facts true in initial state. (Contains no mutexes).
  - A_i is set of actions whose preconditions are true and non-mutex at S_i.
    - Mark as mutex any action pair satisfying 1 or 2.

How Mutexes affect the level graph.

- Hence, mutexes will prune actions and facts from levels of the graph.
- They also record useful information about impossible combinations.

Example

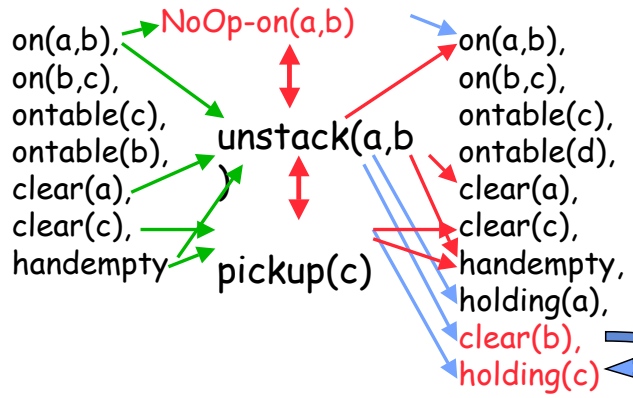


unstack(a,b) deletes the add effect of NoOp-on(a,b), so these actions are mutex as well

pickup deletes handempty, one of unstack(a,b)'s preconditions.

precondition
add effect
del effect

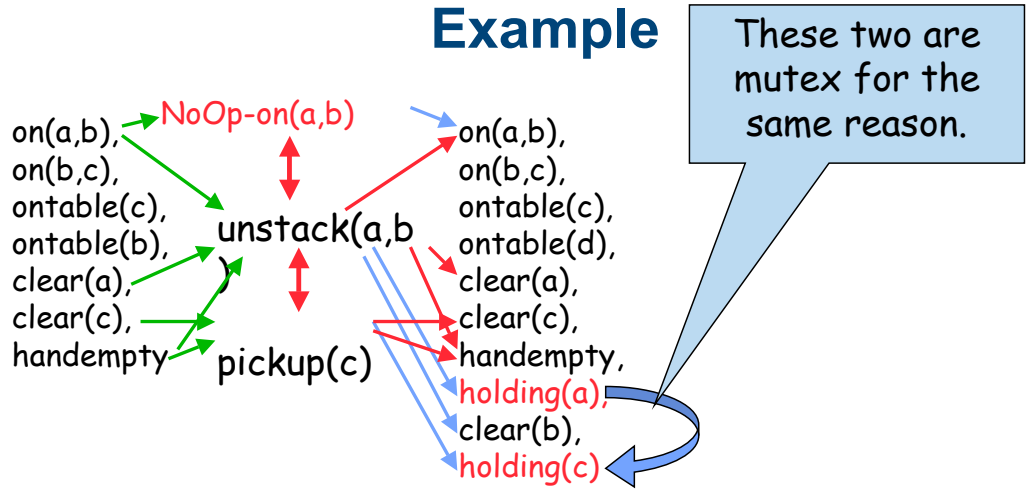
Example



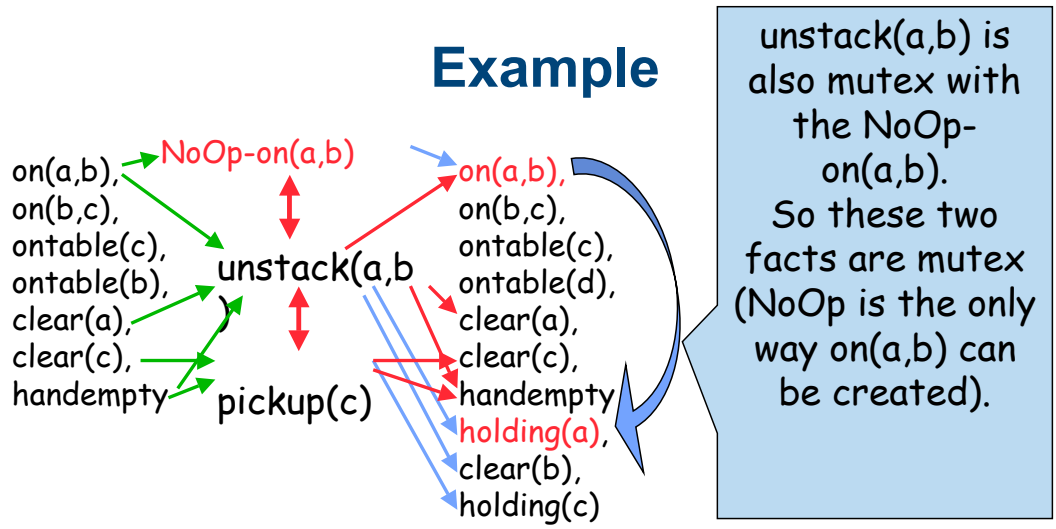
unstack(a,b) is the only action that adds clear(b), and this is mutex with pickup(c), which is the only way of adding holding(c).

precondition
add effect
del effect

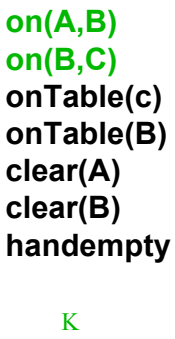
Example



Example

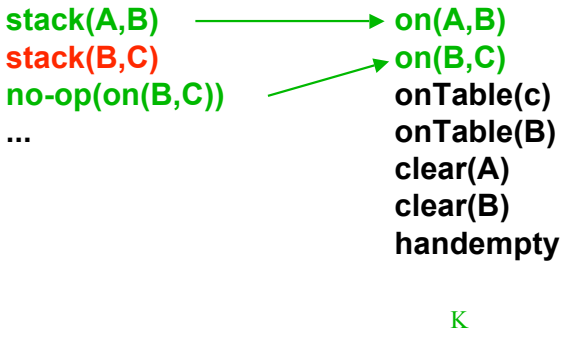


Phase II. Searching the Graphplan



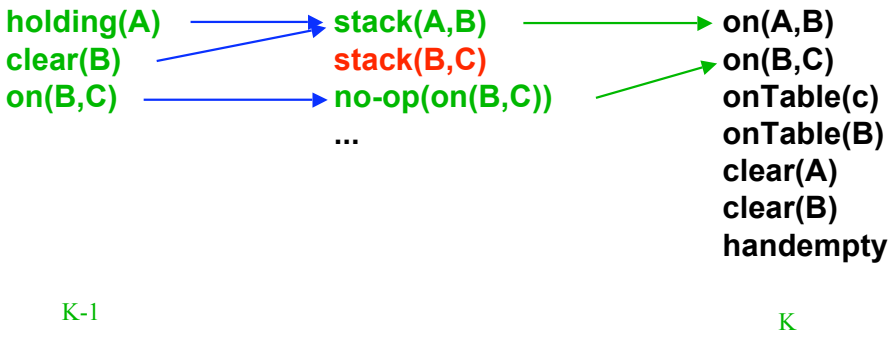
- Build the graph to level k, such that every member of the goal is present at level k, and no two are mutex. Why?

Searching the Graphplan



- Find a non-mutex collection of actions that add all of the facts in the goal.

Searching the Graphplan



- The preconditions of these actions at level K-1 become the new goal at level K-1.
- Recursively try to solve this new goal. If this fails, backtrack and try a different set of actions for solving the goal at level k.

Phase II-Search

- Solve(G,K)
  - forall sets of actions A={a_i} such that
    no pair (a_i, a_j) ∈ A is mutex
    the actions in A suffice to add all facts in G
    - Let P = union of preconditions of actions in A
    - If Solve(P,K-1)
      Report PLAN FOUND
  - At end of forall. Exhausted all possible action sets A
    Report NOPLAN

This is a depth first search.

Graph Plan Algorithm

- Phase I. build leveled graph.
- Phase II. Search leveled graph.
  - Phase I: While last state level does not contain all goal facts with no pair being mutex
    - add new state/action level to graph
    - if last state/Action level = previous state/action level (including all MUTEXES) graph has leveled off) report NO PLAN.
  - Phase II: Starting at last state level search backwards in graph for plan. Try all ways of moving goal back to initial state.
    - If successful report PLAN FOUND.
    - Else goto Phase I.

Dinner Date Example

- Initial State
  {dirty, cleanHands, quiet}
- Goal
  {dinner, present, clean}
- Actions
  - Cook: Pre: {cleanHands}
    Add: {dinner}
  - Wrap: Pre: {quiet}
    Add: {present}
  - Tidy: Pre: {}
    Add: {clean}
    Del: {cleanHands, dirty}
  - Vac: Pre: {}
    Add: {clean}
    Del: {quite, dirty}

Dinner example: rule1 action mutex

Legend: NO: No-Op, **C**:clean,**D**: Dinner, **H**: cleanHands, **P**:Present, **Q**:quiet, **R**: diRty

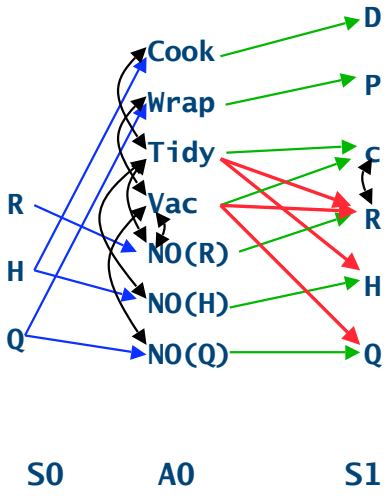
- **Actions (including all No-OP actions)**
  - Cook: Pre: {H} Add: {D} Del: {}
  - Wrap: Pre: {Q} Add: {P} Del: {}
  - Tidy: Pre: {} Add: {C} Del: {H,R}
  - Vac: Pre: {} Add: {C} Del: {Q, R}
  - NO(C): Pre: {C} Add: {C} Del: {}
  - NO(D): Pre: {D} Add: {D} Del: {}
  - NO(H): Pre: {H} Add: {H} Del: {}
  - NO(P): Pre: {P} Add: {P} Del: {}
  - NO(Q): Pre: {Q} Add: {Q} Del: {}
  - NO(R): Pre: {R} Add: {R} Del: {}

- Look at those with non-empty Del, and find others that have these Del in their Pre or Add:
- So, *Rule 1 action* mutex are as follows (these are fixed):
  (Tidy,Cook), (Tidy, NO(H)), (Tidy, NO(R)), (Vac, Wrap), (Vac,NO(Q)), (Vac, NO(R))
- Rule 3 action mutex depend on state layer and you have to build the graph.

Dinner Example:

Legend:

- **Arrows: Blue: pre, Green: add, Red: Del, Black: Mutex**
- **D**: Dinner, **C**:clean, **H**: cleanHands, **Q**:quiet, **P**:Present, **R**: diRty
- **Init={R,H,Q} Goal={D,P,C}**



Note:

- At layer S1 all goals are present and no pair forms a mutex
- So, go to phase II and search the graph:
- i.e. Find a set of non-mutex actions that adds all goals {D,P,C}:

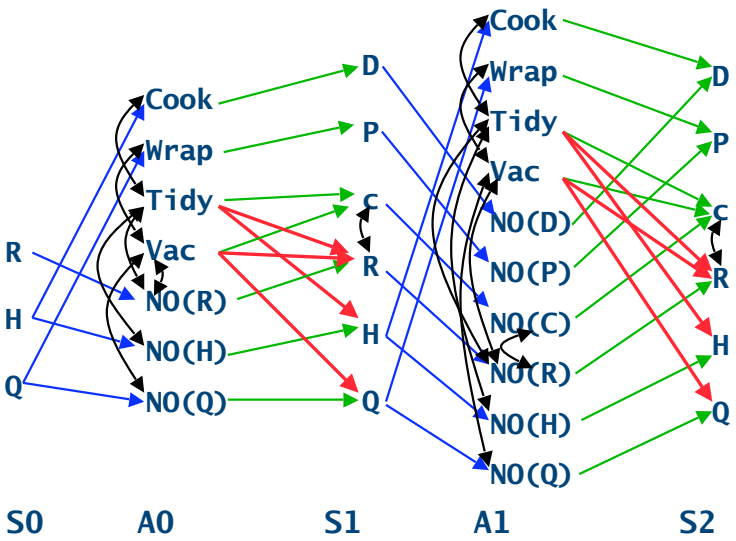
  ✗{Cook, Wrap, Tidy} mutex Tidy&Cook

  ✗{Cook, Wrap, Vac} mutex Vac&Wrap

- No such set exists, nothing to backtrack, so goto phase I and add one more action and state layers

Dinner Example:

- **Arrows: Blue: pre, Green: add, Red: Del, Black: Mutex**
- **D**: Dinner, **C**:clean, **H**: cleanHands, **Q**:quiet, **P**:Present, **R**: diRty
- **Init={R,H,Q} Goal={D,P,C}**
- **Note: first draw rule1 action mutex at layer A1, then find rule3 action mutex (for this only look at mutex fact at level S1). Finally, apply rule 2 for fact mutex at S2.**



Note:At layer S2 all goals are present and no pair forms a mutex, so

- phase II: Find a set of non-mutex actions that adds all goals {D,P,C}:

x{Cook, Wrap, Tidy}

x{Cook, Wrap, Vac}

✓{Cook, Wrap, NO(C)}

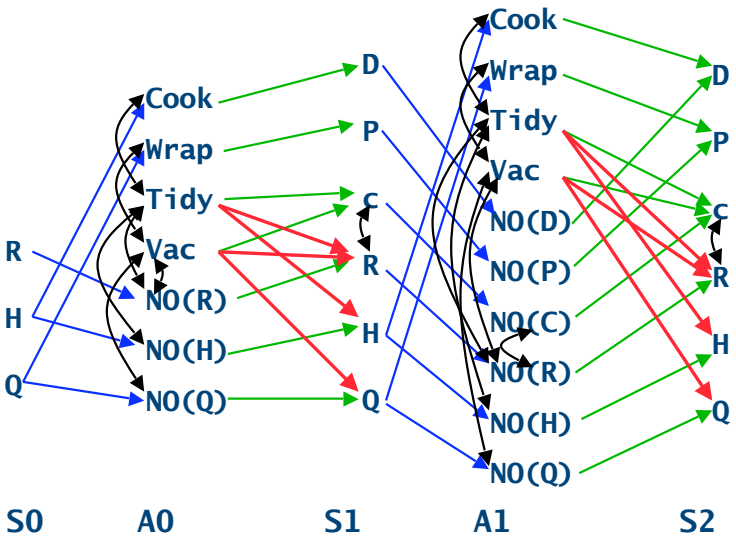
NewG={H,Q,C} @ S1

- **Cannot** find any non-mutex action set in A0
- **Backtrack** to S2, try another action set

✓{Cook, NO(P), Vac}

Dinner Example:

- **Arrows: Blue: pre, Green: add, Red: Del, Black: Mutex**
- **D**: Dinner, **C**:clean, **H**: cleanHands, **Q**:quiet, **P**:Present, **R**: diRty
- **Init={R,H,Q} Goal={D,P,C}**
- **Note: first draw rule1 action mutex at layer A1, then find rule3 action mutex (for this only look at mutex fact at level S1). Finally, apply rule 2 for fact mutex at S2.**



✓{Cook, NO(P), Vac}
NewG={H,P} @ S1

Find a nonmutex set of act in A0 to get NewG:

- {NO(H),Wrap}

- **NewG'={H,Q} @ S0** ✓
- Done!
- **So, the actions are**
- Wrap, {Cook, Vac}
  - Wrap,Cook,Vac
  - Wrap,Vac,Cook
- **Note that we could still backtrack to S2, try remaining action sets!**

ADL Operators.

ADL operators add a number of features to STRIPS.

- Their preconditions can be arbitrary formulas, not just a conjunction of facts.
- They can have conditional and universal effects.
- Open world assumption:
  - States can have negative literals
  - The effect (P∧¬Q) means add P and ¬Q but delete ¬ P and Q.

But they must still specify atomic changes to the knowledge base (add or delete ground atomic facts).

ADL Operators Examples.

move(X,Y,Z)

Pre: on(X,Y) ∧ clear(Z)

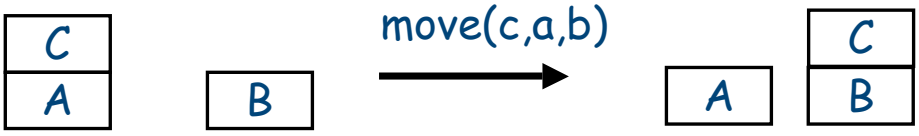
Effs: ADD[on(X,Z)]

DEL[on(X,Y)]

Z ≠ table → DEL[clear(Z)]

Y ≠ table → ADD[clear(Y)]

ADL Operators, example



move(c,a,b)
  Pre: on(c,a) ∧ clear(b)
  Effs: ADD[on(c,b)]
        DEL[on(c,a)]
  b ≠ table → DEL[clear(b)]
  a ≠ table → ADD[clear(a)]

KB = { clear(c), clear(b),
on(c,a),
on(a,table),
on(b,table)}

KB = { on(c,b)
clear(c), clear(a)
on(a,table),
on(b,table)}

ADL Operators Examples.

clearTable()

Pre:

Effs: ∀X. on(X,table) → DEL[on(X,table)]

ADL Operators.

- Arbitrary formulas as preconditions.
  - in a CW-KB we can evaluate whether or not the preconditions hold for an arbitrary precondition.
- They can have conditional and universal effects.
  - Similarly we can evaluate the condition to see if the effect should be applied, and find all bindings for which it should be applied.

Specify atomic changes to the knowledge base.

- CW-KB can be updated just as before.