CSC384: Intro to Artificial Intelligence
Planning-III

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

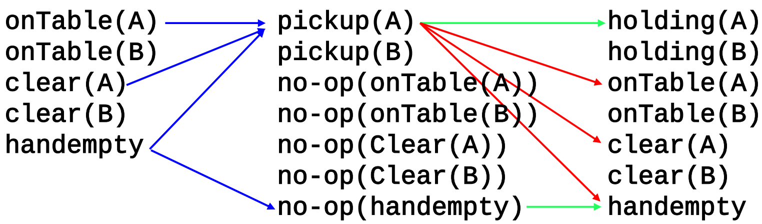
    $\{a^1_1, a^2_1, \ldots\} \rightarrow \{a^1_2, a^2_2, \ldots\} \rightarrow \{a^1_3, a^2_3, \ldots\} \rightarrow \ldots$

    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.

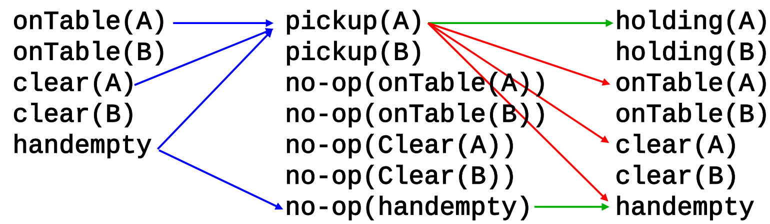
Possible actions

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

GraphPlan Level Graph



Level S_0 contains all facts true in the initial state.

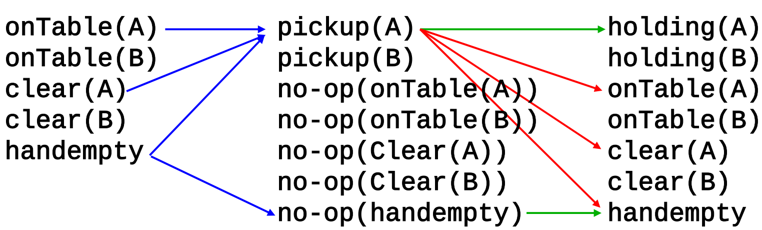
Level A_0 contains all actions whose preconditions are true in S_0. Included in the set of actions are no-ops. One no-op for every ground atomic fact. The precondition of the no-op is its fact, its add effect is its fact.

…

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 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

```
onTable(A)      pickup(A)              holding(A)
onTable(B)      pickup(B)              holding(B)
clear(A)        no-op(onTable(A))      onTable(A)
clear(B)        no-op(onTable(B))      onTable(B)
handempty       no-op(Clear(A))        clear(A)
                no-op(Clear(B))        clear(B)
                no-op(handempty)       handempty
```

- 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

```
onTable(A)      pickup(A)              holding(A)
onTable(B)      pickup(B)              holding(B)
clear(A)        no-op(onTable(A))      onTable(A)
clear(B)        no-op(onTable(B))      onTable(B)
handempty       no-op(Clear(A))        clear(A)
                no-op(Clear(B))        clear(B)
                no-op(handempty)       handempty
```

- 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.

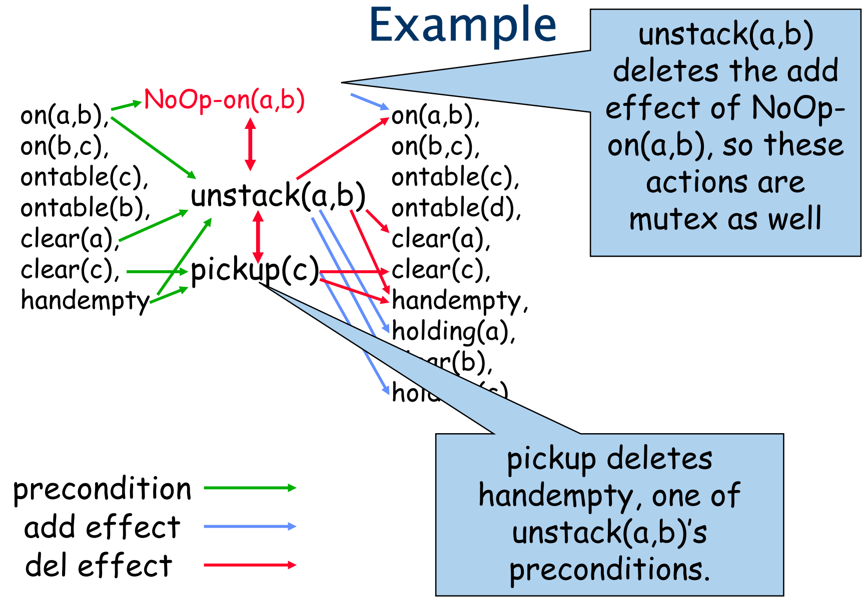
1. Two actions are mutex if either action deletes a precondition or add effect of another
2. Two propositions *p* and *q* are mutex if all actions adding *p* are mutex of all actions adding *q*
3. Two actions are mutex if two of their preconditions are mutex

-
  - …
  - 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.

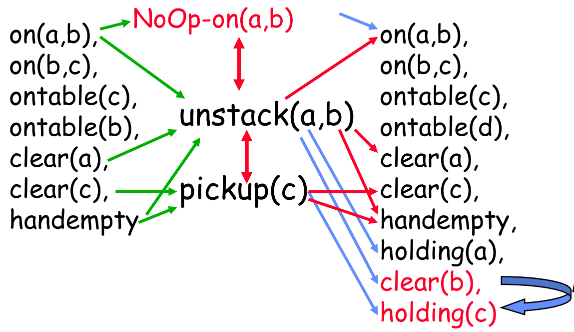
How Mutexes affect the level graph.

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

Example



Example

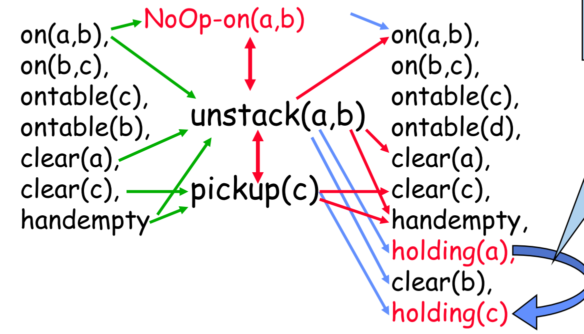


unstack(a,b) is the only action that adds clear(b), and this is mutex with pickup(c), which is the only way of adding holding(c).

precondition
add effect
del effect

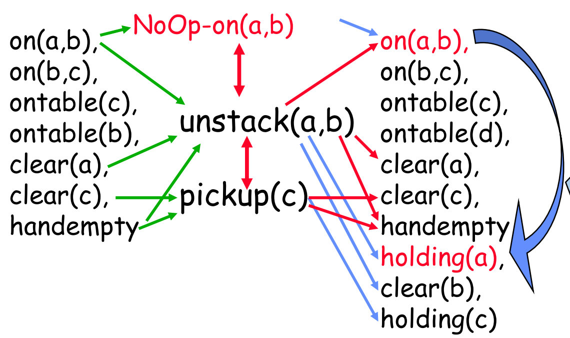
Example

These two are mutex for the same reason.



precondition
add effect
del effect

Example



unstack(a,b) is also mutex with the NoOp-on(a,b). So these two facts are mutex (NoOp is the only way on(a,b) can be created).

precondition
add effect
del effect

Phase II. Searching the Graphplan

on(A,B)
on(B,C)
onTable(c)
onTable(B)
clear(A)
clear(B)
handempty

K

- 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

```
stack(A,B) ----------> on(A,B)
stack(B,C)             on(B,C)
no-op(on(B,C)) ------> onTable(c)
...                    onTable(B)
                       clear(A)
                       clear(B)
                       handempty

                          K
```

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

Searching the Graphplan

```
holding(A) --> stack(A,B) ----------> on(A,B)
clear(B)   --> stack(B,C)             on(B,C)
on(B,C) -----> no-op(on(B,C)) ------> onTable(c)
               ...                    onTable(B)
                                      clear(A)
                                      clear(B)
                                      handempty

   K-1                                   K
```

- 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 $(a_i, a_j) \in A$ 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)

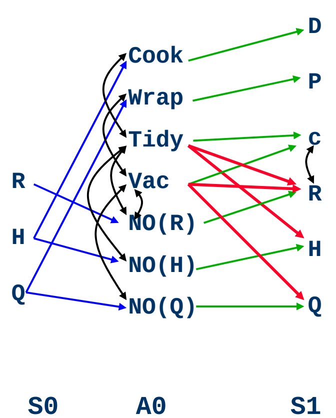
Cook:	Pre: {H}	Add: {D}	Del: {}
Wrap:	Pre: {Q}	Add: {P}	Del: {}
Tidy:	Pre: {}	Add: {C}	Del: {H,R}
Vac:	Pre: {}	Add: {C}	Del: {Q, R}
NO(C):	Pre: {C}	Add: {C}	Del: {}
NO(D):	Pre: {D}	Add: {D}	Del: {}
NO(H):	Pre: {H}	Add: {H}	Del: {}
NO(P):	Pre: {P}	Add: {P}	Del: {}
NO(Q):	Pre: {Q}	Add: {Q}	Del: {}
NO(R):	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:

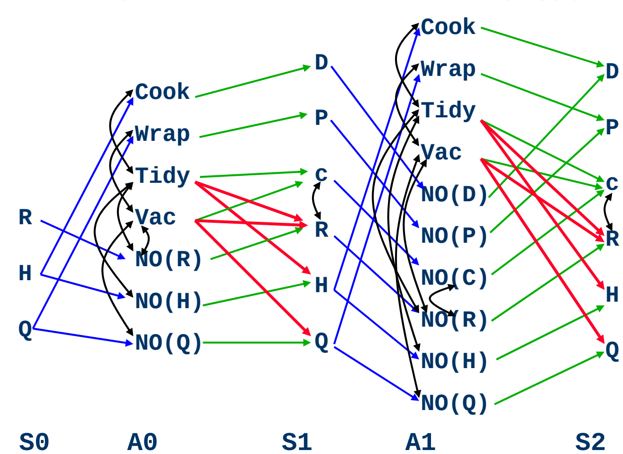
- At layer S1 all goals are present and no pair forms a mutex
- So, go to phase II and search the graph:
- i.e. Find a set of non-mutex actions that adds all goals {D,P,C}:

  ✗{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}:

✗{Cook, Wrap, Tidy}

✗{Cook, Wrap, Vac}

✓{Cook, Wrap, NO(C)}

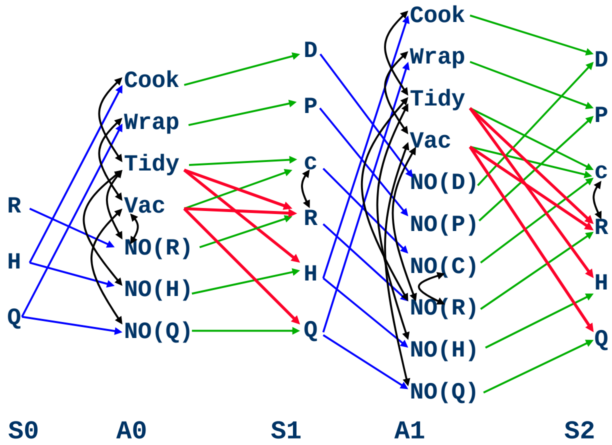
NewG={H,Q,C} @ S1

- **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.

1. Their preconditions can be arbitrary formulas, not just a conjunction of facts.
2. They can have conditional and universal effects.
3. Open world assumption:
   1. States can have negative literals
   2. The effect $(P \wedge \neg Q)$ means add P and $\neg Q$ but delete $\neg$ 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) $\wedge$ clear(Z)

Effs: ADD[on(X,Z)]

DEL[on(X,Y)]

$Z \neq$ table $\rightarrow$ DEL[clear(Z)]

$Y \neq$ table $\rightarrow$ ADD[clear(Y)]

ADL Operators, example

move(c,a,b)
Pre: on(c,a) $\wedge$ clear(b)
Effs: ADD[on(c,b)]
DEL[on(c,a)]
$b \neq$ table $\rightarrow$ DEL[clear(b)]
$a \neq$ table $\rightarrow$ 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: $\forall X.\ on(X,table) \rightarrow DEL[on(X,table)]$

ADL Operators.

1. Arbitrary formulas as preconditions.
   - in a CW-KB we can evaluate whether or not the preconditions hold for an arbitrary precondition.
2. 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.