Planning

- This material is covered in R&N 2nd edition chapters 10.3, 11.1, 11.2, 11.4
- This material is covered in R&N 3rd edition chapter 10

plan n.
1. A scheme, program, or method worked out beforehand for the accomplishment of an objective: a plan of attack.
2. A proposed or tentative project or course of action: had no plans for the evening.
3. A systematic arrangement of elements or important parts; a configuration or outline: a seating plan; the plan of a story.
4. A drawing or diagram made to scale showing the structure or arrangement of something.
5. In perspective rendering, one of several imaginary planes perpendicular to the line of vision between the viewer and the object being depicted.
6. A program or policy stipulating a service or benefit: a pension plan.

Synonyms: blueprint, design, project, scheme, strategy

A portion of a manufacturing process plan

[a representation] of future behavior … usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents.

- Austin Tate

[MIT Encyclopedia of the Cognitive Sciences, 1999]
Modes of Planning

- Mixed Initiative Planning
- Automated Plan Generation

Why Planning

- Intelligent agents must operate in the world. They are not simply passive reasoners (Knowledge Representation, reasoning under uncertainty) or problem solvers (Search), they must also act on the world.
- We want intelligent agents to act in “intelligent ways”. Taking purposeful actions, predicting the expected effect of such actions, composing actions together to achieve complex goals.

Why Planning

E.g. if we have a robot we want the robot to decide what to do; how to act to achieve our goals

A Planning Problem

- How to change the world to suit our needs
- Critical issue: we need to reason about what the world will be like after doing a few actions, not just what it is like now

GOAL: Steven has coffee
CURRENTLY: robot in mailroom, has no coffee, coffee not made, Steven in office, etc.
TO DO: goto lounge, make coffee,...
Example Planning Applications

Autonomous Agents for Space Exploration
- Autonomous planning, scheduling, control
- NASA: JPL and Ames
- Remote Agent Experiment (RAX)
- Deep Space 1
- Mars Exploration Rover (MER)

Other Autonomous Systems
- Sheet-metal bending machines - Amada Corp
- Software to plan the sequence of bends
  [Gupta and Bourne, J. Manufacturing Sci. and Engr., 1999]

Manufacturing Automation
Games

E.g., Bridge Baron - Great Game Products

- 1997 world champion of computer bridge [Smith, Nau, Throop, AI Magazine, 1998]
- 2004: 2nd place

Other Applications (cont.)

- Scheduling with Action Choices & Resource Requirements
  - Problems in supply chain management
  - HSTS (Hubble Space Telescope scheduler)
  - Workflow management

Air Traffic Control

- Route aircraft between runways and terminals. Crafts must be kept safely separated. Safe distance depends on craft and mode of transport. Minimize taxi and wait time.

Character Animation

- Generate step-by-step character behaviour from high-level spec

Plan-based Interfaces

- E.g. NLP to database interfaces
- Plan recognition, Activity Recognition

Other Applications (cont.)

- Web Service Composition
  - Compose web services, and monitor their execution
  - Many of the web standards have a lot of connections to plan representation languages
    - BPEL; BPEL-4WS allow workflow specifications
    - OWL-S allows process specifications

- Grid Services/Scientific Workflow Management

- Genome Rearrangement
  - The relationship between different organisms can be measured by the number of "evolution events" (rearrangements) that separate their genomes
  - Find shortest (or most likely) sequence of rearrangements between a pair of genomes

Planning

- Reasoning about what the world will be like after doing a few actions is similar to what we have already examined.

- Only now we want to reason about dynamic environments.
  - in(robby,Room1), lightOn(Room1) are true: will they be true after robby performs the action turnOffLights?
  - in(robby,Room1) is true: what does robby need to do to make in(robby,Room3) true?

- Reasoning about the effects of actions, and computing what actions can achieve certain effects is at the heart of decision making.
Planning under Uncertainty

- One of the major complexities in planning that we will deal with later is planning under uncertainty.
- Our knowledge of the world is probabilistic.
- Sensing is subject to noise (especially in robots).
- Actions and effectors are also subject to error (uncertainty in their effects).

Planning

For now we restrict our attention to the deterministic case.

We will examine:

- determining the effects of actions.
- finding sequences of actions that can achieve a desired set of effects.
  - This will in some ways be a lot like search, but we will see that representation also plays an important role.

What we’ll cover

- Representing and reasoning about actions (including planning) in the situation calculus, a (mostly) first-order logical language for reasoning about action and change
- Planning in STRIPS
- Planning as Search
- GraphPlan

Situation Calculus

- First we look at how to model dynamic worlds within first-order logic.
- The situation calculus is an important formalism developed for this purpose.
- Situation Calculus is a first-order language.
- Include in the domain of individuals a special set of objects called situations. Of these $S_0$ is a special distinguished constant which denotes the “initial” situation.
### Situations

Situations are the history of actions from $s_0$. You can think of them as indexing “states” of the world, but two different situations can have the same state. (E.g., “scratch, eat” may lead to the same state of the world as “eat, scratch.”) When dealing with dynamic environments, the world has different properties at different points in time.

- $\text{in(robby,room1, s_0)}$, $\neg\text{in(robby,room3,s_0)}$  
  $\neg\text{in(robby,room3,s_1)}$, $\text{in(robby,room1,s_1)}$

Different things are true in situation $s_1$ than in the initial situation $s_0$.

Contrast this with the previous kinds of knowledge we examined.

### Fluents

Previously, we were encoding a property of a term as a relation in first-order logic. The distinction here is that properties that change from situation to situation (called fluents) take an extra situation argument.

- $\text{clear(b)} \rightarrow \text{clear(b,s)}$
  - “clear(b)” is no longer statically true, it is true contingent on what situation we are talking about
Actions

- Actions are also part of language
  - A set of “primitive” action objects in the (semantic) domain of individuals.
  - In the syntax they are represented as functions mapping objects to primitive action objects.

Examples:
- pickup(X) function mapping blocks to actions
  - pickup(c) = “the primitive action object corresponding to ‘picking up block c’
- stack(X,Y)
  - stack(a,b) = “the primitive action object corresponding to ‘stacking a on top of b’

Actions applied to situation → new situation

- Remember that actions are terms in the language.
- In order to talk about the situation that results from executing an action in a particular situation, there is a “generic” action application function do(A,S).
  - do maps a primitive action A and a situation S to a new situation.
  - The new situation is the situation that results from applying A to S.

Example:
- do(pickup(c), s_0) = the new situation that is the result of applying action “pickup(c)” to the initial situation s_0.

What do Actions do?

- Actions affect the situation by changing what is true.
  - on(c,a,s_0); clear(a,do(pickup(c),s_0))
- We want to represent the effects of actions, this is done in the situation calculus with two components:
  - Action Precondition Axioms
  - Action Effect Axioms

Specifying the effects of actions

- Action preconditions. Certain things must hold for actions to have a predictable effect.
  - pickup(c) this action is only applicable to situations S when “clear(c,S) ∧ handempty(S)” is true.
- Action effects. Actions make certain things true and certain things false.
  - holding(c, do(pickup(c), S))
  - ∀ X.¬handempty(do(pickup(X),S))
Specifying the effects of actions

- Action effects are conditional on their precondition being true.

\[
\forall S,X. \\
\text{ontable}(X,S) \land \text{clear}(X,S) \land \text{handempty}(S) \\
\implies \text{holding}(X, \text{do}(\text{pickup}(X),S)) \\
\land \neg \text{handempty}(\text{do}(\text{pickup}(X),S)) \\
\land \neg \text{ontable}(X, \text{do}(\text{pickup}(X,S))) \\
\land \neg \text{clear}(X, \text{do}(\text{pickup}(X,S))).
\]

Green indicates a situation term

Plan Generation

There are many ways to generate plans. Here we show how to do it by representing actions in the situation calculus (as you have just seen) and generating a plan via deductive plan synthesis.

This is not the approach taken by state-of-the-art planners, as we will see later, but it is where the field started and is still used for more complex tasks in reasoning about action and change.

...so for now, back to resolution!

Reasoning with the Situation Calculus.

1. \text{clear}(c,s_0)
2. \text{on}(c,a,s_0)
3. \text{clear}(b,s_0)
4. \text{ontable}(a,s_0)
5. \text{ontable}(b,s_0)
6. \text{handempty}(s_0)

Query:

\[
\exists Z. \text{holding}(b,Z)
\]

Does there exist a situation in which we are holding b? And if so what is the name of that situation.

Resolution

- Convert "pickup" action axiom into clause form:

\[
\forall S,Y. \\
\text{ontable}(Y,S) \land \text{clear}(Y,S) \land \text{handempty}(S) \\
\implies \text{holding}(Y, \text{do}(\text{pickup}(Y),S)) \\
\land \neg \text{handempty}(\text{do}(\text{pickup}(Y),S)) \\
\land \neg \text{ontable}(Y, \text{do}(\text{pickup}(Y,S))) \\
\land \neg \text{clear}(Y, \text{do}(\text{pickup}(Y,S))).
\]
### Resolution

12. R[8d, 7] \(Y=b, Z=\text{do}(\text{pickup}(b), S))\)
\(\neg\text{ontable}(b, S), \neg\text{clear}(b, S), \neg\text{handempty}(S), \text{ans}(\text{do}(\text{pickup}(b), S))\)

13. R[12a,5] \(S=s_0\)
\(\neg\text{clear}(b, s_0), \neg\text{handempty}(s_0), \text{ans}(\text{do}(\text{pickup}(b), s_0))\)

14. R[13a,3] {}\n\(\neg\text{handempty}(s_0), \text{ans}(\text{do}(\text{pickup}(b), s_0))\)

15. R[14a,6] {}\n\text{ans}(\text{do}(\text{pickup}(b), s_0))

### The answer?

- \text{ans}(\text{do}(\text{pickup}(b), s_0))

- This says that a situation in which you are holding b is called “\text{do}(\text{pickup}(b), s_0)”

- This tells you what actions to execute to achieve “holding(b)”.

### Two types of reasoning.

- In general we can answer questions of the following form:
  1. Predicting the effects of a given sequence of action
    E.g., \(\text{on}(b, c, \text{do}(\text{stack}(b, c), \text{do}(\text{pickup}(b), s_0)))\)
  2. Computing a sequence of actions that achieve a goal conditions E.g.,
    \(\exists S. \text{on}(b, c, S) \land \text{on}(c, a, S)\)

### The Frame Problem

- Unfortunately, logical reasoning won’t immediately yield the answer to these kinds of questions.

- e.g., query: \(\text{on}(c, a, \text{do}(\text{pickup}(b), s_0))\)?
  - is c still on a after we pickup b?
  - Intuitively it should be
  - Can logical reasoning reach this conclusion given the representation of actions that we have proposed thus far?
The Frame Problem

1. clear(c,s0)
2. on(c,a,s0)
3. clear(b,s0)
4. ontable(a,s0)
5. ontable(b,s0)
6. handempty(s0)
8. (¬ontable(Y,S), ¬clear(Y,S), ¬handempty(S),
   holding(Y,do(pickup(Y,S))))
9. (¬ontable(Y,S), ¬clear(Y,S), ¬handempty(S),
   ¬handempty(do(pickup(X),S))))
10. (¬ontable(Y,S), ¬clear(Y,S), ¬handempty(S),
    ¬ontable(Y,do(pickup(Y,S))))
11. (¬ontable(Y,S), ¬clear(Y,S), ¬handempty(S),
    ¬clear(Y,do(pickup(Y,S))))
12. ¬on(c,a,do(pickup(b),s0)) {QUERY}

Nothing can resolve with 12!

Logical Consequence

• Remember that resolution only computes logical consequences.
• We stated the effects of pickup(b), but did not state that it doesn’t affect on(c,a).
• Hence there are models in which on(c,a) no longer holds after pickup(b) (as well as models where it does hold).
• The problem is that representing the non-effects of actions is very tedious and in general is not possible.
  • Think of all of the things that pickup(b) does not affect!

The Frame Problem

• Finding an effective way of specifying the non-effects of actions, without having to explicitly write them all down is the frame problem.
• Good solutions have been proposed, and the situation calculus has been a powerful way of dealing with dynamic worlds:
  • Logic-based high level robotic programming languages

Computation Problems

• Although the situation calculus is a very powerful representation. It is not always efficient enough to use to compute sequences of actions.
• The problem of computing a sequence of actions to achieve a goal is “planning”
• Next we will study some less expressive representations that support more efficient planning.
Simplifying the Planning Problem

- Assume complete information about the initial state through the closed-world assumption (CWA).
- Assume a finite domain of objects
- Assume that action effects are restricted to making (conjunctions of) atomic formulae true or false. No conditional effects, no disjunctive effects, etc.
- Assume action preconditions are restricted to conjunctions of ground atoms.

Closed World Assumption (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
  a) if a ground atomic fact is not in our list of “known” facts, its negation must be true.
  b) 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.

“CW KB” or “CW-KB” = Closed-world knowledge base
“CWA” = Closed World Assumption

Querying A CW KB

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

if F is atomic
    return(F ∈ KB)

if F = F₁ ∧ F₂
    return(Query(F₁) && Query(F₂))

if F = F₁ ∨ F₂
    return(Query(F₁) || Query(F₂))

if F = ¬F₁
    return(! Query(F₁))

if F = F₁ → F₂
    return(!Query(F₁) || Query(F₂))

if F = ∃X.F₁
    for each constant c ∈ KB
        if (Query(F₁{X=c}))
            return(true)
        return(false).

if F = ∀X.F₁
    for each constant c ∈ KB
        if (!Query(F₁{X=c}))
            return(false)
        return(true).
Querying A CW KB

Guarded quantification (for efficiency).

if F = ∀X.F₁
  for each constant c ∈ KB
  if (!Query(F₁{X=c}))
    return(false)
  return(true).

E.g., consider checking
∀X: apple(x) → sweet(x)
we already know that the formula is true for all "non-apples"

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.

Querying A CW KB

Guarded quantification (for efficiency).

∀X:[p(X)] F₁ ↔ ∀X: p(X) → F₁
  for each constant c s.t. p(c)
  if (!Query(F₁{X=c}))
    return(false)
  return(true).

∃X:[p(X)] F₁ ↔ ∃X: p(X) ∧ F₁
  for each constant c s.t. p(c)
  if (Query(F₁{X=c}))
    return(true)
  return(false).

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,s₀) ∧ ¬ on(a,b,s₁)
• 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**

\[
\begin{align*}
\text{pre} &= \{\text{handempty, clear(b), ontable(b)}\} \\
\text{add} &= \{\text{holding(b)}\} \\
\text{del} &= \{\text{handempty, clear(b), ontable(b)}\}
\end{align*}
\]

\[
\begin{align*}
\text{KB} &= \{\text{handempty, clear(c), clear(b), on(c,a), ontable(a), ontable(b)}\} \\
\text{KB} &= \{\text{holding(b)}, clear(c), on(c,a), ontable(a)}\}
\end{align*}
\]
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)\}

• 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
```

• 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), ...

Elevator Control

Figure 1: A Micronic® keypad allows passengers to choose 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 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 to determine a sequence of actions that when applied to the initial CW-KB yield an updated CW-KB which satisfies the goal.

This is known as the classical planning task.

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.

```
move(b,c)  B  C  A
move(c,b)  A  C  B
move(c,table)  A  B  C
move(a,b)  B  A  C
```
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 one technique:
  **Relaxed Plan heuristics used with heuristic search.**

  The heuristics are domain independent. As such they are part of a class of so-called domain-independent heuristic search for planning.

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 $A_i$.
  (again, remember we’re ignoring the delete lists!!)

Recall: 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).
Example

\[ \text{on}(a,b), \quad \text{on}(b,c), \quad \text{on}(c,d), \quad \text{clear}(a), \quad \text{clear}(b), \quad \text{handempty} \]

\[ \text{unstack}(a,b), \quad \text{pickup}(d) \]

\[ S_0 \quad A_0 \]

Example

\[ \text{on}(a,b), \quad \text{on}(b,c), \quad \text{on}(c,d), \quad \text{clear}(a), \quad \text{clear}(b), \quad \text{handempty}, \quad \text{clear}(d), \quad \text{holding}(a), \quad \text{clear}(b), \quad \text{holding}(d) \]

\[ S_1 \quad A_1 \]

Reachability

- 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!
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

---

Example

```plaintext
legend: [pre]act[add]

$S_0 = \{f_1, f_2, f_3\}$
$A_0 = \{[f_1]a_1[f_4], [f_2]a_2[f_5]\}$
$S_1 = \{f_1, f_2, f_3, f_4, f_5\}$
$A_1 = \{[f_2, f_4, f_5]a_3[f_6]\}$
$S_2 = \{f_1, f_2, f_3, f_4, f_5, f_6\}$

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

We split $G$ into $G_p$ and $G_N$:

```
CountActs(G, S_2)
```

$G_p = \{f_5, f_1\}$ /already in $S_1$
$G_N = \{f_6\}$ /New in $S_2$

$A = \{a_3\}$ /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($G_1, S_1$)
```

CountActs($G_1, S_1$):

```plaintext
Now, we are at level $S_1$

$S_0 = \{f_1, f_2, f_3\}$
$A_0 = \{[f_1]a_1[f_4], [f_2]a_2[f_5]\}$
$S_1 = \{f_1, f_2, f_3, f_4, f_5\}$
$A_1 = \{[f_2, f_4, f_5]a_3[f_6]\}$
$S_2 = \{f_1, f_2, f_3, f_4, f_5, f_6\}$

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

We split $G$ into $G_p$ and $G_N$:

```
CountActs(G, S_2)
```

$G_p = \{f_5, f_1\}$ /already in $S_0$
$G_N = \{f_4, f_5\}$ /New in $S_1$

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

//the new goal: $G_p \cup$ Pre(A)
$G_1 = \{f_5, f_1, f_2, f_4\}$
Return
$2 + \text{CountActs}(G_2, S_0) = 2 + 0$

So, in total $\text{CountActs}(G, S_2) = 1 + 2 = 3$
```
Using the Heuristic

1. To use CountActions as a heuristic, we build a layered structure from a state S that reaches the goal.
2. Then we CountActions to see how many actions are required in a relaxed plan.
3. We use this count as our heuristic estimate of the distance of S to the goal.
4. 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.

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 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 action 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.

- 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.

onTable(A)
onTable(B)
clear(A)
clear(B)
handempty

pickup(A)
pickup(B)
no-op(onTable(A))
no-op(onTable(B))
no-op(Clear(A))
no-op(Clear(B))
no-op(handempty)

holding(A)
holding(B)
onTable(A)
onTable(B)
clear(A)
clear(B)
handempty

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 true and 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.

…

\( S_i \) 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

- \text{on}(a,b), \text{on}(b,c), \text{ontable}(c), \text{ontable}(b), \text{clear}(a), \text{clear}(c), \text{handempty} 
- \text{unstack}(a,b)
- \text{pickup}(c)
- \text{on}(a,b), \text{on}(b,c), \text{ontable}(c), \text{ontable}(d), \text{clear}(a), \text{clear}(c), \text{handempty}, \text{holding}(a), \text{clear}(b), \text{holding}(c)

precondition
add effect
del effect

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).

These two are mutex for the same reason.
Example

\[\text{on}(a,b), \quad \text{on}(b,c), \quad \text{onTable}(c), \quad \text{onTable}(b), \quad \text{clear}(a), \quad \text{clear}(c), \quad \text{handempty}\]

\[\text{unstack}(a,b), \quad \text{unstack}(b,c), \quad \text{handempty}\]

\[\text{NoOp} \rightarrow \text{on}(a,b)\]

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

\[\text{on}(A,B)\]
\[\text{on}(B,C)\]
\[\text{onTable}(c)\]
\[\text{clear}(A)\]
\[\text{clear}(B)\]
\[\text{handempty}\]

\[\text{holding}(A)\]
\[\text{clear}(B)\]

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

\[\text{stack}(A,B)\]
\[\text{stack}(B,C)\]
\[\text{no-op}(\text{on}(B,C))\]

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

Searching the Graphplan

\[\text{holding}(A)\]
\[\text{stack}(A,B)\]
\[\text{no-op}(\text{on}(B,C))\]

K-1

- 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)
  - For all 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

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

Legend:
- NO: No-Op
- C: clean
- D: Dinner
- H: cleanHands
- P: Present
- Q: Quiet
- R: dirty

- 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}

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td></td>
</tr>
<tr>
<td>A0</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
</tr>
</tbody>
</table>

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):
    - X{Cook, Wrap, Tidy} mutex Tidy&Cook
    - X{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

At layer A1, first draw rule1 action mutex, then find rule3 action mutex (for this only look at mutex fact at level S1).
At S2, apply rule 2 for fact mutex. 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}
- x{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
  - x{Cook, NO(P), Vac}
  - NewG={H,P} @ S1
  - Find a nonmutex set of act in A0 to get NewG:
    - x{NO(H), Wrap}
  - NewG'={H,Q} @ S0
  - Done!
  - So, the actions are:
    - Wrap, {Cook, Vac}
  - I.e., sequentially either
    - Wrap, Cook, Vac
    - Wrap, Vac, Cook
  - Note that we could still backtrack to S2, try remaining action sets!

Beyond STRIPS

STRIPS operators are not very expressive and as a consequence not as compact as they might be.
ADL (Action Description Language) extends the expressivity of STRIPS
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 \( P \land \neg Q \) 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: \( on(X,Y) \land clear(Z) \)

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

\[
\begin{array}{|c|c|}
\hline
C & A \\
\hline
& B \\
\hline
\end{array}
\]

\[
\xrightarrow{move(c,a,b)}
\]

\[
\begin{array}{|c|c|}
\hline
C & A \\
\hline
& B \\
\hline
\end{array}
\]

move\( (c,a,b) \)

Pre: \( on(c,a) \land clear(b) \)

Effs: \( ADD[on(c,a)] \)
\( DEL[on(c,a)] \)
\( b \neq table \rightarrow DEL[clear(b)] \)
\( a \neq table \rightarrow ADD[clear(a)] \)

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

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.