Lecture 9: First-order Logic and Planning

- First-order logic
- Inference in first-order logic
- Expressing planning problems: PDDL and STRIPS language and PDDL
- State-space planning
  - Forward planners
  - Goal regression
- Plan-space planning
- What to do if plans fail

Recall: Propositional Logic

- The good: Propositional logic is very simple!

  Just facts (literals), usual logical connectives, inference is simple
- The bad: Propositional logic is very simple!
  - We cannot express things in a compact way
  - Knowledge base may need to have many similar facts and sentences
- E.g. in the wumpus world, we want to be able to express how a move works *for all squares* in one sentence

First-Order Logic (FOL)

- A key element of FOL are *predicates*, which are used to describe objects, properties, and relationships between objects

  E.g. *On(x,y)*
- A *quantified statement* is a statement that applies to a class of objects

  E.g. $\forall x\ On(x, Table) \rightarrow Fruit(x)$
  - This means that there is only fruit on the table
  - The first element is called a *quantifier*, *x* is a *variable* and *Table* is a *constant*
  - *On* is a *predicate*
- The use of *quantifiers* allows FOL to handle *infinite domains*, while propositional logic can only handle finite domains.

Syntax of FOL: Basic elements

Constants	*Wumpus, 2, CS424, ...*
Predicates	*At*, $>$,...
Functions	$\log$, $\exp$,...
Variables	*x, y, ...*
Connectives	$\wedge\ \vee\ \neg\ \rightarrow \leftrightarrow$
Equality	$=$
Quantifiers	$\forall\ \exists$

Atomic sentences

Atomic sentence $=$ *predicate*(*term*$_1$,...,*term*$_n$)
or *term*$_1$ $=$ *term*$_2$

Term $=$ *function*(*term*$_1$,...,*term*$_n$)
or *constant* or *variable*

E.g., *At(Wumpus,2,1)* is an atomic sentence with one predicate

Complex sentences

Complex sentences are made from atomic sentences using connectives

$$\neg S, \quad S_1 \wedge S_2, \quad S_1 \vee S_2, \quad S_1 \rightarrow S_2, \quad S_1 \leftrightarrow S_2$$

E.g. *At(Wumpus,2,1)* $\rightarrow \neg$ *At(Wumpus,1,2)*
$>(1,2) \vee \leq(1,2)$
$>(1,2) \wedge \neg>(1,2)$

Universal Quantification

- Syntax: $\forall$*variables sentence*
- E.g. Everyone taking AI is smart.
  $\forall x \; Taking(x,AI) \rightarrow Smart(x)$
- Semantics: $\forall x \; S$ is equivalent to the *conjunction of instantiations* of S:

  $$\begin{array}{ll} & Taking(John,AI) \rightarrow Smart(John) \\ \wedge & Taking(Ann,AI) \rightarrow Smart(Ann) \\ \wedge & \ldots \end{array}$$

- Typically, $\rightarrow$ is the main connective with $\forall$.

Example

- What does this statement mean:
  $\forall x \; Taking(x,AI) \wedge Smart(x)$
- Common mistake: using $\wedge$ as the main connective with $\forall$:
  $\forall x \; Taking(x,AI) \wedge Smart(x)$
  means "Everyone is taking AI and everyone is smart"

Existential Quantification

- Syntax: $\exists$*variables sentence*
- Someone taking AI is smart:

  $\exists x$ *Taking(x,AI)* $\wedge$ *Smart(x)*
- Semantics: $\exists x\ S$ is equivalent to the *disjunction of instantiations* of S

$$
\begin{array}{ll}
 & (\mathit{Taking(Ann,AI)} \wedge \mathit{Smart(Ann)}) \\
\vee & (\mathit{Taking(John,AI)} \wedge \mathit{Smart(John)}) \\
\vee & \ldots
\end{array}
$$

- Typically, $\wedge$ is the main connective with $\exists$.

Example

- What does this mean:

  $\exists x$ *Taking(x,AI)* $\rightarrow$ *Smart(x)*
- Common mistake: using $\rightarrow$ as the main connective with $\exists$:

  $\exists x$ *Taking(x,AI)* $\rightarrow$ *Smart(x)* is true if there is anyone who is not taking AI!

Properties of quantifiers

- $\forall x \forall y$ is the same as $\forall y \forall x$
- $\exists x \exists y$ is the same as $\exists y \exists x$
- *$\exists x \forall y$ is not the same as $\forall y \exists x$*

  $\exists x \forall y Loves(x, y)$
  "There is a person who loves everyone in the world"

  $\forall y \exists x Loves(x, y)$
  "Everyone in the world is loved by at least one person"

Example

Let x and y be real numbers.

- $\forall x \exists y \;\; x > y$
- $\exists x \forall y \;\; x > y$

What does each sentence mean? Are they valid, satisfiable or unsatisfiable?

Abellard-Eloise Games

- Abellard handles the universal quantifiers $\forall$
- Eloise handles the existential quantifiers $\exists$
- Abellard is trying to choose values to make the sentence inside the quantifiers false
- Eloise is trying to make the statement inside the quantifiers true
- They take turns as specified by the order of the quantifiers, left to right
- The sentence is valid if and only if Eloise has a winning strategy

Quantifier Duality

Each quantifier can be expressed using the other quantifier and negation:

- $\forall x$ *Likes(x,IceCream)* is equivalent to $\neg\exists x\ \neg$*Likes(x,IceCream)*
- $\exists x$ *Likes(x,Broccoli)* is equivalent to $\neg\forall x\ \neg$*Likes(x,Broccoli)*

Fun with Sentences

- Brothers are siblings

$$\forall x \forall y Brother(x, y) \rightarrow Sibling(x, y)$$

- "Sibling" is reflexive

$$\forall x \forall y Sibling(x, y) \leftrightarrow Sibling(y, x)$$

- One's mother is one's female parent

$$\forall x \forall y Mother(x, y) \leftrightarrow (Female(x) \wedge Parent(x, y))$$

- A first cousin is a child of a parent's sibling

$$\forall x \forall y FirstCousin(x, y) \leftrightarrow \exists p \exists ps Parent(p, x) \wedge$$
$$Sibling(ps, p) \wedge Parent(ps, y)$$

Equality

- *term*$_1$ = *term*$_2$ is true under a given interpretation if and only if *term*$_1$ and *term*$_2$ refer to the same object
- Example:
  Obj1=*Obj2* is satisfiable
  $2 = 2$ is valid
- Example: definition of the sibling predicate:

$$\forall x \forall y Sibling(x, y) \leftrightarrow [\neg(x = y) \wedge$$
$$\exists m \exists f \neg(m = f) \wedge Parent(m, x) \wedge Parent(f, x) \wedge$$
$$Parent(m, y) \wedge Parent(f, y)]$$

Proofs

The proof process can be viewed as a *search* in which the operators are *inference rules*:

- *Modus Ponens (MP)*

$$\frac{\alpha, \quad \alpha \rightarrow \beta}{\beta} \qquad \frac{\textit{Takes(Joe,AI)} \quad \textit{Takes(Joe,AI)} \rightarrow \textit{Cool(Joe)}}{\textit{Cool(Joe)}}$$

- *And-Introduction (AI)*

$$\frac{\alpha \quad \beta}{\alpha \wedge \beta} \qquad \frac{\textit{Cool(Joe)} \quad \textit{CSMajor(Joe)}}{\textit{Cool(Joe)} \wedge \textit{CSMajor(Joe)}}$$

- *Universal Elimination (UE)*: τ must be a ground term i.e. a term with no variables

$$\frac{\forall x \alpha}{\alpha\{x/\tau\}} \qquad \frac{\forall x \ \textit{Takes(x,AI)} \rightarrow \textit{Cool(x)}}{\textit{Takes(Pat,AI)} \rightarrow \textit{Cool(Pat)}}$$

Example Proof

Bob is a buffalo	1. Buffalo(Bob)
Pat is a pig	2. Pig(Pat)
Buffaloes outrun pigs	3. $\forall$x$\forall$y Buffalo(x) $\wedge$ Pig(y) $\rightarrow$ Faster(x,y)
AI 1 & 2	4. Buffalo(Bob) $\wedge$ Pig(Pat)
UE 3, x/Bob, y/Pat	5. Buffalo(Bob) $\wedge$ Pig(Pat) $\rightarrow$ Faster(Bob,Pat)
MP 4 & 5	6. Faster(Bob,Pat)

Search with Primitive Inference Rules

Operators are inference rules

States are sets of sentences

Goal test checks state to see if it contains query sentence

AI, UE, MP is a common inference pattern
Problem: branching factor huge, especially for UE
Idea: find a substitution that makes the rule premise match some known facts
$\Rightarrow$ a single, more powerful inference rule

Unification

- A *substitution* σ unifies atomic sentences p and q if $p\sigma = q\sigma$

p	*q*	σ
Knows(John,x)	*Knows(John,Jane)*	*x/Jane*
Knows(John,x)	*Knows(y,Mary)*	*y/John,x/Mary*
Knows(John,x)	*Knows(y,Mother(y))*	*y/John,x/Mother(John)*

- Idea: Unify rule premises with known facts, apply unifier to conclusion
- E.g., if we know *q* and the rule: *Knows(John,x)* $\rightarrow$ *Likes(John,x)*, we conclude:
  - *Likes(John,Jane)*
  - *Likes(John,Mary)*
  - *Likes(John,Mother(John))*

Generalized Modus Ponens (GMP)

$$\frac{p_1{}',\ \ p_2{}',\ \ldots,\ p_n{}',\ \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{q\sigma} \quad \text{where } p_i{}'\sigma = p_i\sigma \text{ for all } i$$

$$\begin{aligned}
\text{E.g. } p_1{}' &= \quad \mathit{Faster(Bob,Pat)} \\
p_2{}' &= \quad \mathit{Faster(Pat,Steve)} \\
p_1 \wedge p_2 \rightarrow q &= \quad \mathit{Faster(x,y)} \wedge \mathit{Faster(y,z)} \rightarrow \mathit{Faster(x,z)} \\
\sigma &= \quad x/\mathit{Bob}, y/\mathit{Pat}, z/\mathit{Steve} \\
q\sigma &= \quad \mathit{Faster(Bob,Steve)}
\end{aligned}$$

GMP is used with KB of *definite clauses* (*exactly* 1 positive literal):

- A single atomic sentence
- Or a clause of the form: (conjunction of atomic sentences) $\Rightarrow$ (atomic sentence)

All variables assumed universally quantified.

Completeness in FOL

- Procedure i is complete if and only if

$$\mathit{KB} \vdash_i \alpha \quad \text{whenever} \quad \mathit{KB} \models \alpha$$

- GMP is *complete for KBs of universally quantified definite clauses*, but incomplete for general first-order logic
- Example:

$$\begin{aligned}
\mathit{PhD(x)} &\rightarrow \mathit{HighlyQualified(x)} \\
\neg\, \mathit{PhD(x)} &\rightarrow \mathit{EarlyEarnings(x)} \\
\mathit{HighlyQualified(x)} &\rightarrow \mathit{Rich(x)} \\
\mathit{EarlyEarnings(x)} &\rightarrow \mathit{Rich(x)}
\end{aligned}$$

should be able to infer *Rich(Doina)*. But the second sentence has two positive literals, so Modus ponens will not work.

Resolution

- Entailment in first-order logic is only *semi-decidable*: we can find a proof of α if *KB* $\models \alpha$ but cannot always prove that *KB* $\not\models \alpha$

  Cf. Halting Problem (see COMP-330): proof procedure may be about to terminate with success or failure, or may go on for ever
- However, there is a sound an complete inference procedure, called *resolution*

Resolution

- Resolution is a sound and complete inference method for first-order logic.
- Resolution is a *refutation* procedure: to prove that *KB* $\models \alpha$, we show that *KB* $\wedge \neg\alpha$ is unsatisfiable
- The knowledge base and $\neg\alpha$ are expressed in universally quantified, conjunctive normal form
- Like in propositional logic, the resolution inference rule combines two clauses to make a new one:
- Inference continues until an empty clause is derived (contradiction)

Resolution inference rule

Basic propositional version:

$$\frac{\alpha \vee \beta, \quad \neg\beta \vee \gamma}{\alpha \vee \gamma} \qquad \text{or equivalently} \qquad \frac{\neg\alpha \rightarrow \beta, \quad \beta \rightarrow \gamma}{\neg\alpha \rightarrow \gamma}$$

Full first-order version:

$$\frac{\begin{array}{c} p_1 \vee \ldots \; p_j \; \ldots \vee p_m, \\ q_1 \vee \ldots \; q_k \; \ldots \vee q_n \end{array}}{(p_1 \vee \ldots \; p_{j-1} \vee p_{j+1} \; \ldots p_m \vee q_1 \ldots \; q_{k-1} \vee q_{k+1} \; \ldots \vee q_n)\sigma}$$

where $p_j\sigma = \neg q_k\sigma$

Conjunctive Normal Form (CNF)

- Literal = (possibly negated) atomic sentence, e.g., ¬ *Rich(Me)*
- Clause = disjunction of literals, e.g., ¬ *Rich(Me)* ∨ *Unhappy(Me)*
- The knowledge base is a big conjunction of clauses.
- Example:

$$\frac{\begin{array}{l} \neg Rich(x) \vee Unhappy(x) \\ Rich(Me) \end{array}}{Unhappy(Me)}$$

with $\sigma = \{x/Me\}$

Converting a Knowledge Base to CNF

1. Replace $P \rightarrow Q$ by $\neg P \vee Q$
2. Move $\neg$ inwards, e.g., $\neg \forall x\, P$ becomes $\exists x\, \neg P$
3. Standardize variables apart, e.g., $\forall x\, P \vee \exists x\, Q$ becomes $\forall x\, P \vee \exists y\, Q$
4. Move quantifiers left in order, e.g., $\forall x\, P \vee \exists x\, Q$ becomes $\forall x \exists y\, P \vee Q$
5. Eliminate $\exists$ by Skolemization (next slide)
6. Drop universal quantifiers
7. Distribute $\wedge$ over $\vee$, e.g., $(P \wedge Q) \vee R$ becomes $(P \vee R) \wedge (Q \vee R)$

Skolemization

- We want to get rid of existentially quantified variables: $\exists$*Rich(x)* becomes *Rich(G1)* where *G1* is a new *Skolem constant*.
- Example: $\exists k \frac{d}{dy}(k^y) = k^y$ becomes $\frac{d}{dy}(e^y) = e^y$
- It gets more tricky when $\exists$ is inside $\forall$
- E.g.: "Everyone has a heart"
  $\forall$x *Person(x)* $\rightarrow$ $\exists$y *Heart(y)* $\wedge$ *Has(x,y)*

  How should we replace *y* here?
- Incorrect: $\forall$x *Person(x)* $\rightarrow$ *Heart(H1)* $\wedge$ *Has(x,H1)*
- Correct: $\forall$x *Person(x)* $\rightarrow$ *Heart(H(x))* $\wedge$ *Has(x,H(x))*
  where *H* is a new symbol called a *Skolem function*
- Skolem functions have as arguments all enclosing universally quantified variables

Resolution Proof

To prove α:

1. Negate it
2. Convert to CNF
3. Add the result to the knowledge base
4. Infer a contradiction (empty clause)

Example: to prove *Rich(Doina)*, add $\neg$ *Rich(Doina)* to the CNF KB
$\neg$ *PhD(x)* $\vee$ *HighlyQualified(x)*
PhD(x) $\vee$ *EarlyEarnings(x)*
$\neg$ *HighlyQualified(x)* $\vee$ *Rich(x)*
$\neg$ *EarlyEarnings(x)* $\vee$ *Rich(x)*

Resolution Strategies

Heuristics that impose a sensible order on the resolutions we attempt:

- *Unit resolution*: prefer to perform resolution if one clause is just a literal - yields shorter sentences
- *Set of support*: identify a subset of the *KB* (hopefully small); every resolution will take a clause from the set and resolve it with another sentence, then add the result to the set of support
  - *Can make inference incomplete!*
- *Input resolution*: always combine a sentence from the query or *KB* with another sentence
  - Modus ponens is a kind of input resolution
  - Not compete in general

More resolution strategies

- *Linear resolution*: resolve P and Q if P is in the original KB or is an ancestor of Q in the proof tree.
- *Subsumption*: eliminate all sentences more specific than a sentence already in the KB
- *Demodulation and paramodulation*: special extra inference rules to allow treatment of equality

Applications of First-Order Logic

- Prolog: a logic programming languages
- Production systems
- Semantic nets
- Automated theorem proving
- *Planning*

STRIPS

- Developed at Stanford in early 1970s (Stanford Research Institute Planning System), for the first "intelligent" robot
- *Domain*: a set of typed objects; usually represented as propositions
- *States* are represented as first-order predicates over objects
  - *Closed-world assumption:* everything not stated is false; the only objects in the world are the ones defined
- *Operators/Actions* defined in terms of:
  - *Preconditions:* when can the action be applied?
  - *Effects:* what happens after the action?

  No explicit description of how the action should be executed
- *Goals*: conjunctions of literals

STRIPS representations

- States are represented as conjunctions

  In(Robot,room) $\wedge$ $\neg$ In(Charger, r) $\wedge$ ...
- Goals are represented as conjunctions:

  (implicit $\exists$ r) In(Robot, r) $\wedge$ In(Charger, r)
- Actions (operators):
  - Name: Go(here, there)
  - Preconditions: expressed as conjunctions
    At(Robot, here) $\wedge$ Path(here, there)
  - Postconditions (effects): expressed as conjunctions
    At(Robot, there) $\wedge$ $\neg$ At(Robot, here)
- Variables can only be instantiated with objects of the correct type

STRIPS Operator Representation

- Operators have a name, preconditions and postconditions or effects
- Preconditions are conjunctions of *positive literals*
- Postconditions/effects are represented in terms of:
  - *Add-list*: list of propositions that become true after the action
  - *Delete-list*: list of propositions that become false after the action

Semantics

- If the precondition is false in a world state, the action does not change anything (since it cannot be applied)
- If the precondition is true:
  - Delete the items in the Delete-list
  - Add the items in the Add-list.

  Order of operations is important here!

This is a very restricted language, which means we can do efficient inference.

Example: Buying Action

- Action: *Buy(x)* (where *x* is a good)
- Precondition: *At(s), Sells(s,x,p), HaveMoney(p)* (where *s* is a store, *p* is the price)
- Effect:
  - Add-list: *Have(x)*
  - Delete-list: *HaveMoney(p)*
- *Note that many important details are abstracted away!*
- Additional propositions can be added to show that now the store has the money, the stock has decreased etc.

Example: Move Action

- Action: *Move(object, from, to)*
- Preconditions: *At(object, from), Clear(to), Clear(object)*
- Effects:
  - Add-list: *At(object, to), Clear(from)*
  - Delete-list: *At(object, from), Clear(to)*

Pros and cons of STRIPS

- Pros:
  - Since it is restricted, inference can be done efficiently
  - All operators can be viewed as simple deletions and additions of propositions to the knowledge base
- Cons:
  - Assumes that a small number of propositions will change for each action (otherwise operators are hard to write down, and reasoning becomes expensive)
  - Limited language (preconditions and effects are expressed as conjunctions, implicit quantifiers), so not applicable to all domains of interest.

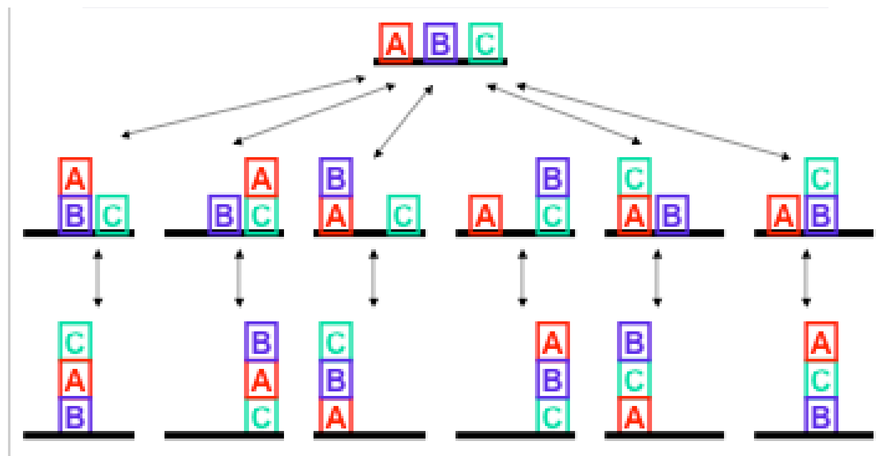
Example: Blocks World

Initial state = On(A,table) ∧ On(B,table) ∧ On(C,table) ∧ Clear(A) ∧ Clear(B) ∧ Clear(C)
Goal state = On(A,B) ∧On (B,C)

Action = Move(b,x,y)
Precondition = On(b,x) ∧ Clear(b) ∧ Clear(y)
Effect = On(b,y) ∧ Clear(x) ∧ ¬On(b,x) ∧ ¬Clear(y)

Action = MoveToTable(b,x)
Preconditions = On(b,x) ∧ Clear(b)
Effect = On(b,Table) ∧ Clear(x) ∧ ¬On(b,x)

State Transitions in the Blocks World



Two Basic Approaches to Planning

1. *State-space planning* works at the level of the states and operators
   - Finding a plan is formulated as a *search through state space*, looking for a path from the start state to the goal state(s)
   - Most similar to constructive search
2. *Plan-space planning* works at the level of plans
   - Finding a plan is formulated as a *search through the space of plans*
   - We start with a partial, possibly incorrect plan, then apply changes to it to make it a full, correct plan
   - Most similar to iterative improvement/repair

Plan-Space Planning in the Blocks World

- Start with plan: *Put(A,B), Put(B,C)*
- Plan fails because the precondition of the second action is not satisfied after the first action
- So we can try to add a step, remove a step, or re-order the steps

State-Space Planners

- *Progression planners* reason from the start state, trying to find the operators that can be applied (match preconditions)
- *Regression planners* reason from the goal state, trying to find the actions that will lead to the goal (match effects or post-conditions)

In both cases, the planners work with *sets of states* instead of using individual states, like in straightforward search

Progression (Forward) Planning

1. Determine all operators that are applicable in the start state
2. Ground the operators, by replacing any variables with constants
3. Choose an operator to apply
4. Determine the new content of the knowledge base, based on the operator description
5. Repeat until goal state is reached.

Example: Supermarket Domain

- In the start state we have *At(Home)*, which allows us to apply operators of the type *Go(x,y)*.
- The operator can be instantiated as *Go(Home, HardwareStore)*, *Go(Home,GroceryStore)*, *Go(Home, School)*, ...
- If we choose to apply *Go(Home, HardwareStore)*, we will delete from the KB *At(Home)* and add *At(HardwareStore)*.
- The new proposition enables new actions, e.g. *Buy*

Note that in this case there are <u>*a lot of possible operators*</u> to perform!

Goal Regression

- Introduced in Newell & Simon's General Problem Solver
- Algorithm:
  1. Pick an action that satisfies (some of) the goal propositions
  2. Make a new goal by:
     - Removing the goal conditions satisfied by the action
     - Adding the preconditions of this action
     - Keeping any unsolved goal propositions
  3. Repeat until the goal set is satisfied by the start state

Example: Supermarket Domain

- In the goal state we have *At(Home)* $\wedge$ *Have(Milk)* $\wedge$ *Have(Drill)*
- The action *Buy(Milk)* would allow us to achieve *Have(Milk)*
- To apply this action we need to have the precondition *At(GroceryStore)*, so we add it to the set of propositions we want to achieve
- The goal set becomes: *At(Home)* $\wedge$ *At(GroceryStore)* $\wedge$ *Have(Drill)*
- Next, we may want to achieve *At(HardwareStore)*

Note that in this case the *order* in which we try to achieve these propositions matters!

Variations of Goal Regression

- Using a *stack* of goals - also called *linear planning*

  This is *not complete!* I.e. we may not find a plan even if one exists
- Using a *set* of goals - also called *non-linear planning*

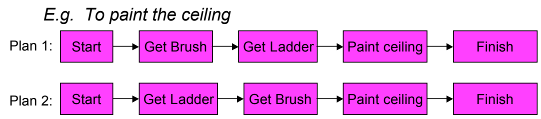
  This is *complete*, but more expensive (need to decide what to work on next)
- Both versions are *sound*: only legal plans will be found

Prodigy Planner

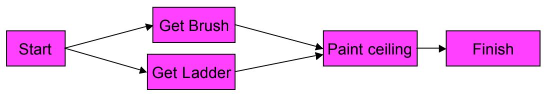
- Do both forward search and goal regression at the same time.
- At each step, choose either an operator to apply or goal to regress
- Uses domain-dependent heuristics to guide the search
- General heuristics (e.g. number of propositions satisfied) do not work well in planning, because subgoals interact

Total vs. Partial Order

- Total order: Plan is always a strict sequence of actions



- Partial order: Plan steps may be unordered



Partial Order Planning

- Search in *plan space* and use *least commitment* whenever possible
- In state space search:
  - Search space is a set of states of the world
  - Actions cause transitions between states
  - Plan is a path through state space
- In plan space search:
  - Search space is a set of *partial plans*
  - *Plan operators* cause transitions
  - Goal is a legal plan
- Can maintain both *state* and *plans* (e.g. Prodigy)

Plan-Space Planners

Plan is defined by $\langle A, O, B, L \rangle$:

- A is a set of actions/operators from the problem domain
- O is a set of ordering constraints of the form $a_i < a_j$

  The constraint specifies that a_i must come before a_j but does not say exactly when
- B is a set of bindings, of the form $v_i = C$, $v_i \neq C$, $v_i = v_j$ or $v_i \neq v_j$, where v_i, v_j are variables and C is a constant
- L is a set of causal links, which records why a certain ordering has to occur:

  $a_i \rightarrow^c a_j$ means that action a_i achieves effect c which is a precondition of a_j

Plan Transformations

- Adding *actions*
- Specifying *orderings*
- *Binding* variables

Constraint satisfaction is used along the way to ensure the consistency of orderings

Discussion of Partial-Order Planning

- Advantages:
  - Plan steps may be unordered (plan will be ordered, or linearized, before execution)
  - Handles concurrent plans
  - Least commitment can lead to shorter search times
  - Sound and complete
  - Typically produces the optimal plan
- Disadvantages:
  - Complex plan operators lead to high cost for generating every action
  - Larger search space, because of concurrent actions
  - Hard to determine what is true in a state

The real world

Things are usually not as expected:

- **Incomplete information**
  - Unknown preconditions, e.g., *Intact(Spare)*
  - Disjunctive effects, e.g., *Inflate(x)* causes *Inflated(x)* according to the knowledge base, but in reality it actually causes *Inflated(x)* $\vee$ *SlowHiss(x)* $\vee$ *Burst(x)* $\vee$ *BrokenPump* $\vee \ldots$
- **Incorrect information**
  - Current state incorrect, e.g., spare NOT intact
  - Missing/incorrect postconditions in operators
- **Qualification problem**: can never finish listing all the required preconditions and possible conditional outcomes of actions

Solutions

- *Conditional (contingency) planning*:
  1. Plans include **observation actions** which obtain information
  2. Sub-plans are created for each contingency (each possible outcome of the observation actions)

  E.g. Check the tire. If it is intact, then we're ok, otherwise there are several possible solutions: inflate, call AAA....

  Expensive because it plans for many unlikely cases
- *Monitoring/Replanning*:
  1. Assume normal states, outcomes
  2. Check progress *during execution*, replan if necessary

  Unanticipated outcomes may lead to failure (e.g., no AAA card)

  In general, some monitoring is unavoidable

Monitoring

- **Execution monitoring**: "failure" means that the preconditions of the *remaining plan* not met
- **Action monitoring**: "failure" means that the preconditions of the *next action* not met (or action itself fails)

In both cases, need to *replan*

Replanning

- Simplest: on failure, replan from scratch
- Better: plan to get back on track by reconnecting to best continuation

  In this case, we can try to reconnect to the plan's next action, or some future action

  The latter is typically more expensive in terms of planning computation (lots of possible places to reconnect!) but usually yields better plans (e.g. if it is very hard to achieve the preconditions of the very next action)

Summary

- Planning is very related to search, but allows the actions/states have more structure
- We typically use logical inference to construct solutions
- State-space vs.plan-space planning
- Least-commitment: we build partial plans, order them only as necessary
- In the real world, it is necessary to consider failure cases - replanning
- Hierarchy and abstraction make planning more efficient
- Many varieties of planners that we have not looked at: case-based planners, MDP planners (we will see this later on) etc.