Knowledge representation and reasoning
Lecture 1: Introduction

Natasha Alechina
natasha.alechina@nottingham.ac.uk

Definition of knowledge-based systems and knowledge bases

- Knowledge-based systems are systems for which intentional stance is grounded by design in symbolic representation
- The symbolic representation of knowledge is called a knowledge base.

Examples of knowledge-based systems

- Various expert systems
  - MYCIN (1970s, Stanford University)
  - XCON (1978, Carnegie Mellon University)
- Perhaps most famous knowledge base: CYC (1980s, Douglas Lenat, Cycorp, Austin, Texas)
- Ontologies
  - Snomed CT `http://snomed.dataline.co.uk/`
  - Gene ontology `http://www.geneontology.org/`
- Google Knowledge Graph
- (Parts of) IBM Watson

Long standing split in AI

- explicit symbolic representation (symbolic AI); this module is symbolic AI
- non-symbolic AI: behaviour robotics, (deep) learning
- machine learning saves us from having to hand-code explicit reasoning rules and statements about the world; some tasks which humans can easily do are unlikely to every be formalisable
- on the down side, hard to say exactly what a machine-learned program will do next
- are there things that machine learning/collecting information from the web will never be able to do?

Winograd Schema Challenge

- Hector Levesque (2013): Winograd Schema Challenge (to replace Turing test)
- requires understanding the meaning of language vs exploiting statistical correlations
- Examples:
  - The trophy would not fit in the brown suitcase because it was too big. What was too big?
    1. the trophy
    2. the suitcase
  - Joan made sure to thank Susan for all the help she had given. Who had given the help?
    1. Joan
    2. Susan

Winograd schema challenge

- there is a competition running since 2016 called Winograd Schema Challenge sponsored by Nuance Communications
- there was a successful individual project at the School by George Hallam in 2013/14 to answer some types of Winograd schema questions
- represented knowledge about fitting things in containers etc.
- used first order reasoning (resolution) to produce answers

Handout for G53KRR lecture on resolution .

1. Reducing a first order sentence to clausal normal form.

1. eliminate $\supset$ and $\equiv$ using

$$(\alpha \supset \beta) \equiv (\neg\alpha \vee \beta)$$

$$(\alpha \equiv \beta) \equiv ((\alpha \supset \beta) \wedge (\beta \supset \alpha))$$

2. move $\neg$ inward so that it appears only in front of an atom, using

$$\neg\neg\alpha \equiv \alpha$$

$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$$

$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$$

$$\neg\forall x\alpha \equiv \exists x\neg\alpha$$

$$\neg\exists x\alpha \equiv \forall x\neg\alpha$$

3. ensure that each quantifier has a distinct variable by renaming:

$$\forall x\alpha \equiv \forall y\alpha(x/y)$$

$$\exists x\alpha \equiv \exists y\alpha(x/y)$$

where y does not occur in $\forall x\alpha$ and $\exists x\alpha$ and $\alpha(x/y)$ means α with all occurrences of x replaced by y.

4. eliminate existentials using Skolemisation:

if $\exists x\alpha$ is not in the scope of any universal quantifiers, then we replace $\exists x\alpha$ with $\alpha(a)$ where a is a new constant called a Skolem constant. (It should be different for every existential quantifier).

if $\exists x\alpha$ is in the scope of universal quantifiers $\forall x_1, \ldots, \forall x_n$ (and these are all universal quantifiers it is in the scope of):

$$\forall x_1(\ldots\forall x_2\ldots\forall x_n(\ldots\exists x\alpha)\ldots)$$

then replace $\exists x\alpha$ by $\alpha(x/f(x_1, \ldots, x_n))$ where f is a Skolem function (again use a different Skolem function for every existential quantifier).

Example: $\exists x_1\exists x_2\forall y\exists z P(x_1, x_2, y, z)$ becomes $\forall y P(c_1, c_2, y, f(y))$.

5. Move universals outside the scope of $\wedge$ and $\vee$ using the following equivalences (provided x is not free in α):

$$(\alpha \wedge \forall x\beta) \equiv \forall x(\alpha \wedge \beta)$$

$$(\alpha \vee \forall x\beta) \equiv \forall x(\alpha \vee \beta)$$

6. We now got $\forall x_1 \ldots \forall x_n \alpha$ where α does not contain quantifiers. Reduce α to CNF as before using distributivity:

$$\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

7. Collect terms:

$$(\alpha \vee \alpha) \equiv \alpha$$

$$(\alpha \wedge \alpha) \equiv \alpha$$

8. Reduce to clausal form by dropping universal quantifiers and conjunctions and making disjunctions into clauses (lists of literals):

$$\forall x((P(x) \vee \neg R(a, f(b, x))) \wedge Q(x, y))$$

becomes

$$[P(x), \neg R(a, f(b, x))], [Q(x, y)]$$

9. Actually need also to do *factoring*: if a clause contains two literals ρ_1 and ρ_2 which unify in both directions (there is a substitution θ such that $\rho_1\theta = \rho_2$ and θ' such that $\rho_2\theta' = \rho_1$) then replace them with a single literal. For example, $[P(x), P(y)]$ becomes $[P(x)]$.

2. Substitution and unification.

A substitution θ is a finite set of pairs $\{x_1/t_1, \ldots, x_n/t_n\}$ where x_i are distinct variables and t_i are arbitrary terms (could all be the same variable y, or $f(x, y, b)$ or whatever). If ρ is a literal then $\rho\theta$ is a literal which results from simultaneously substituting each x_i in ρ by t_i. Same for clauses: if c is a clause the $c\theta$ is the result of applying the substitution to all literals in c.

For example, if $\theta = \{x/a, y/g(x, b, z)\}$ then

$$[P(x), \neg R(a, f(b, x))]\theta = [P(a), \neg R(a, f(b, a))]$$

$$[Q(x, y)]\theta = [Q(a, g(x, b, z))]$$

θ unifies (is a unifier for) two literals ρ_1 and ρ_2 if $\rho_1\theta = \rho_2\theta$. For example, $P(x, f(x))$ and $P(y, f(a))$ are unified by $\theta = \{x/a, y/a\}$.

3. General rule of resolution:

$$\frac{c_1 \cup \{\rho_1\} \quad c_2 \cup \{\neg\rho_2\}}{(c_1 \cup c_2)\theta}$$

where θ unifies ρ_1 and ρ_2: $\rho_1\theta = \rho_2\theta$. We also assume that we renamed all variables in $c_1 \cup \{\rho_1\}$ and $c_2 \cup \{\neg\rho_2\}$ so that each clause has its distinct variables.

Example:

$$\frac{[\neg Man(x), Mortal(x)] \quad [Man(socrates)]}{[Mortal(socrates)]}$$

using $\theta = \{x/socrates\}$. (So $[Mortal(x)]\theta = [Mortal(socrates)]$.)

G53KRR: unification, resolution with equality

A unifier of two literals ρ_1 and ρ_2 is a substitution θ such that $\rho_1\theta = \rho_2\theta$.

There are many possible unifiers, and some of them are too specific. For example, $P(x, y)$ and $P(a, z)$ can be unified by $\theta_1 = x/a, y/z$ and by $\theta_2 = x/a, y/b, z/b$ (the second one is too specific).

A most general unifier (mgu) for ρ_1 and ρ_2 is a unifier θ such that for any other unifier θ', there is a further substitution $\theta*$ such that $\theta' = \theta\theta*$. Basically, θ' is obtained from θ by doing some extra substitutions.

An mgu for ρ_1 and ρ_2 (assuming ρ_1 and ρ_2 do not have common variables to start with) can be computed as follows:

1. start with $\theta = \{\ \}$
2. exit if $\rho_1\theta = \rho_2\theta$
3. set DS to be the pair of terms at the first place where $\rho_1\theta$ and $\rho_2\theta$ disagree
4. find a variable v in DS and a term t in DS not containing v; if none exist, fail
5. otherwise set θ to $\theta\{v/t\}$ and go to step 2.

If we have to deal with a set of clauses containing equality, we need to add to KB the following axioms:

reflexivity $\forall x(x = x)$

symmetry $\forall x \forall y(x = y \supset y = x)$

transitivity $\forall x \forall y \forall z(x = y \wedge y = z \supset x = z)$

substitution for functions for every function f of arity n in the set of clauses,
$\forall x_1 \forall y_1 \ldots \forall x_n \forall y_n(x_1 = y_1 \wedge \ldots \wedge x_n = y_n \supset f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$

substitution for predicates for every predicate P of arity n in the set of clauses,

$$\forall x_1 \forall y_1 \ldots \forall x_n \forall y_n(x_1 = y_1 \wedge \ldots \wedge x_n = y_n \supset P(x_1, \ldots, x_n) \supset P(y_1, \ldots, y_n))$$

With these axioms, resolution is sound and complete for refutation for first-order logic with equality.

Example: $\forall x(x = a \vee x = b)$, $\exists x P(x)$, $\neg P(a)$, $\neg P(b)$ should be inconsistent. Clauses:

C1 $[x = a, x = b]$

C2 $[P(c)]$

C3 $[\neg P(a)]$

C4 $[\neg P(b)]$

We will also use the following instance of substitution for predicates:

C5 $[\neg(x_1 = y_1), \neg P(x_1), P(y_1)]$

Proof:

(1) $[\neg(c = y_1), P(y_1)]$ from C5 and C2, x_1/c

(2) $[\neg(c = a)]$ from C3 and (1), y_1/a

(3) $[c = b]$ from (2) and C1, x/c

(4) $[P(b)]$ from (3) and (1), y_1/b

(5) [] from (4) and C4.

G53KRR Handout on Horn clauses, SLD resolution, backward chaining

Horn clauses A Horn clause is a clause which contains at most one positive literal:

$$[\neg Child(x), \neg Female(x), Girl(x)]$$

$$[\neg Girl(x)]$$

Positive and negative Horn clauses

- Horn clauses which do contain a positive literal are called positive or definite clauses.
- Horn clauses which do not contain a positive literal are called negative clauses (or sometimes *goals*)

Note that positive Horn clauses are equivalent to FOL sentences of the form:

$$\forall x_1 \ldots \forall x_n(\rho_1 \wedge \ldots \wedge \rho_n \supset \rho)$$

For example,

$$\forall x(Child(x) \wedge Female(x) \supset Girl(x))$$

$$Female(a)$$

SLD resolution SLD stands for **S**elected **L**iterals, **L**inear pattern, **D**efinite Clauses.

An SLD derivation of a clause c from a set of clauses S is a sequence

$$c_1, \ldots, c_n$$

such that

- $c_n = c$,
- $c_1 \in S$
- and each c_{i+1} is a resolvent of c_i and some clause from S.

Note that apart from c_1 all clauses in an SLD derivation are negative clauses (because resolution always 'eats up' the only positive literal in the clause from S). If S is a set of Horn clauses and from S by resolution we can derive [ ], then we can always derive [ ] from S using SLD resolution. SLD resolution is complete for refutation for Horn clauses, but not for arbitrary clauses.

Example The process of deriving an empty clause can be seen as 'eliminating' negative literals using positive clauses in the KB. For example,

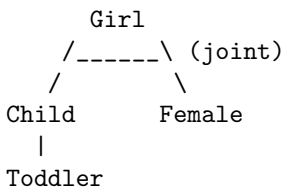
$$Child \wedge Female \supset Girl$$

$$Toddler \supset Child$$

$$Toddler$$

$$Female$$

and we want to derive $Girl$, so add a clause $\neg Girl$. First we have one goal (negative clause) $Girl$; we eliminate it against $[\neg Child, \neg Female, Girl]$ and get two new subgoals: $Child$, $Female$. Goal tree:



SLD resolution derivation:

$c_1 = \neg Girl$

$c_2 = [\neg Child, \neg Female]$ (by resolution from c_1 and $[\neg Child, \neg Female, Girl]$)

$c_3 = [\neg Toddler, \neg Female]$ (by resolution from c_2 and $[\neg Toddler, Child]$)

$c_4 = [\neg Female]$ (by resolution from c_3 and $[Toddler]$)

$c_5 = [\,]$ ((by resolution from c_4 and $[Female]$).

Backward chaining for propositional clauses (first order requires unification):
input: a finite set of atomic sentences $q_1, \ldots, q_n$
output: YES if KB entails all of q_i, NO otherwise

```
procedure: SOLVE[q1, ..., qn ]
if n = 0 then return YES
for each clause c in KB do
     if c = [not p1,...,not pm, q1] and SOLVE [p1,...,pm,q2,...,qn]
     then return YES
end for
return NO
```

Backward chaining on an example:

```
SOLVE[Girl]
   c = [Girl, not Child,not Female] call SOLVE [Child, Female]
   c = [not Toddler, Child] call SOLVE[Toddler,Female]
   c = [Toddler] call SOLVE[Female]
   c = [Female] call SOLVE[] return YES
```

PROCEDURAL CONTROL OF REASONING. PROLOG. Procedural control in the backward chaining procedure:

- Depth first (first solve ps rather than qs)
- Left to right (solve in order as they are listed)
- Backward chaining because search from goals to facts in KB

First order case requires unification, but the order is the same. This is the execution strategy of Prolog.

Prolog Prolog is a (logic) programming language where programs consist of facts and rules (Horn clauses):

```
parent(john, tom).
father(X, Y) :- parent(X,Y), male(X).
```

This is the same as:

$$Parent(john, tom)$$

$$\forall x \forall y (Parent(x, y) \land Male(x) \supset Father(x, y)$$

(Note that in Prolog variables are upper case and constants and predicate names are lower case. Every clause ends with a dot, and implications are pointing right to left.)

G53KRR: Forward chaining, production systems

Some issues with backward chaining To finish backward chaining:

1. Backward chaining on a single clause $[\neg q, q]$ and query q does not terminate.

2. It is possible to force backward chaining into an exponential proof search. Consider $KB_n = \{[\neg p_{i-1}, p_i], [\neg p_{i-1}, q_i], [\neg q_{i-1}, p_i], [\neg q_{i-1}, q_i] : 0 < i < n\}$. For example, for $n = 2$:

$$\{[\neg p_0, p_1], [\neg p_0, q_1], [\neg q_0, p_1], [\neg q_0, q_1]\}$$

Althoough KB_n has $4n - 4$ clauses, for any i, SOLVE(p_i) will take 2^i steps before failing (because there are two ways to prove p_i: prove p_{i-1} or q_{i-1}, and for each of those there are two ways to prove it, etc.)

Forward chaining (for propositional Horn clauses):

input: a finite list of atomic sentences, $q_1, \ldots, q_n$

output: YES if KB entails all of q_i, NO otherwise

1. if all goals q_i are marked as solved, return YES
2. check if there is a clause $[p, \neg p_1, \ldots, \neg p_m]$ in KB, such that all of $p_1, \ldots, p_m$ are marked as solved and p is not marked as solved
3. if there is such a clause, then mark p as solved and go to step 1.
4. otherwise, return NO.

Another way to look at it:

input: a finite list of atomic sentences, $q_1, \ldots, q_n$

output: YES if KB entails all of q_i, NO otherwise

1. if all goals q_i are in KB, return YES
2. check if there is a clause $p_1 \wedge \ldots \wedge p_m \supset p$ in KB, such that all of $p_1, \ldots, p_m$ are in KB and p is not in KB
3. if there is such a clause, then add p to KB and go to step 1.
4. otherwise, return NO.

For first-order case, similar, but need to unify the pattern in the body of the rule $(P_1(\bar{x}) \wedge \ldots \wedge P_m(\bar{x}))$ with unit clauses $P_i(\bar{a})$ in KB first, and then apply the same substitution to $P(\bar{x})$.

For first-order case we do the inference from

$$\forall x_1 \forall x_2 (Parent(x_1, x_2) \wedge Male(x_1) \supset Father(x_1, x_2)),\ Parent(bob, chris),\ Male(bob)$$

to $Father(bob, chris)$.

Production rule systems are forward-chaining reasoning systems which use production rules. Usually production rules are more complex than just Horn clauses (similar to how Prolog has more than just Horn clauses, also negation as failure etc.) but we will look at just Horn clauses. So this is different from the version in the textbook (much simpler).

We just assume that the knowledge base consists of: Working Memory WM which is a finite set of ground atoms (facts, or Working Memory Elements), and a finite set of production rules (universal Horn clauses).

The simplest way of reasoning would be to chain forward as above, but this may flood WM with a lot of irrelevant facts. Instead most production rule systems compute a *conflict set*: the set of all possible *rule instances* applicable for the current state of working memory. A rule instance is a substitution which makes a pattern in some rule match the working memory elements, together with the rule itself. A *conflict resolution strategy* is used to determine which of the rule instances in the conflict set will actually be fired (which conclusions added). Most conflict resolution strategies pick a single rule instance based on one or more of the following criteria: specificity of the rules (which rule has a more specific pattern in the body); or the order in which rules appear in the program; or the order in which facts were added to working memory (for example, depth first where rule instances involving more recent facts are preferred) etc.

Exercise For the following production system, trace the results, assuming that the conflict resolution strategy is: an instance of most important applicable rule is selected. If there are more than one such instances, the instance is selected randomly. The order of rule importance is: R3 more important than R1, R1 is more important than R2.

F1 $animal(tiger)$

F2 $animal(cat)$

F3 $large(tiger)$

F4 $eatsMeat(tiger)$

F5 $eatsMeat(cat)$

R1 $\forall x(animal(x) \wedge large(x) \wedge eatsMeat(x) \supset dangerous(x))$

R2 $\forall x(animal(x) \supset breathesOxygen(x))$

R3 $\forall x(dangerous(x) \supset runAwayNow)$

G53KRR handout on description logic ALC

OWL Web Ontology Language is a W3C standard. It extends most description logics and has slightly different terminology (based on RDF rather than description logic semantics. A subset of OWL, OWL DL, is based on description logic).

Reading:

The Description Logic Handbook. Franz Baader, Diego Calvanese, Deborah L.McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. Cambridge University Press, 2003. ISBN 0-521-78176-0.

A good on-line course: `http://www.inf.unibz.it/~franconi/dl/course/`

Basic idea description logic talks about relationships between *concepts* (noun phrases). There are many different description logics, ALC below is one of them. Most extend ALC with things like counting, transitive roles, inverse roles. Some are subsets of ALC, such as EL which only contains $\exists$, $\sqcup$ and $\sqsubseteq$. EL is mostly what Snomed CT is written in.

Precise definition of the syntax of ALC :

Logical symbols (apart from brackets etc.):

- concept-forming operators: $\forall$, $\exists$, $\sqcup$, $\sqcap$, $\neg$
- connectives: $\sqsubseteq$, $\doteq$

Non-logical symbols:

- Atomic concepts: $Person$, $Thing$,.... Correspond to unary predicates in FOL.
- Roles: age, $employer$, $child$, arm, ... Correspond to binary predicates in FOL.
- Constants: $john, mary, roomA7$,.... Correspond to constants (0-ary functional symbols) in FOL.

Concepts:

- atomic concept is a concept
- if r is a role and C is a concept, then $\forall r.C$ is a concept (e.g. $\forall.child.Girl$ describes someone all of whose children are girls)
- if r is a role and C is a concept, then $\exists r.C$ is a concept (e.g. $\exists.child.Girl$ describes someone who has a daughter)
- if C is a concept, then $\neg C$ (not C) is a concept
- if C_1 and C_2 are concepts then $C_1 \sqcap C_2$ (C_1 and C_2) is a concept
- if C_1 and C_2 are concepts then $C_1 \sqcup C_2$ (C_1 or C_2) is a concept

Sentences:

- if C_1 and C_2 are concepts then $C_1 \sqsubseteq C_2$ is a sentence (all C_1s are C_2s, C_1 is *subsumed* by C_2)
- if C_1 and C_2 are concepts then $C_1 \doteq C_2$ is a sentence (C_1 is equivalent to C_2)
- if a is a constant and C a concept then $C(a)$ is a sentence (the individual denoted by a satisfies the description expressed by C)
- if a, b are constants and r a role then $r(a, b)$ is a sentence (the individuals denoted by a and b are connected by the role r)

A description logic knowledge base is a set of description logic sentences.

TBox and ABox A description logic knowledge base is usually split into terminological part or TBox which describes general relationships between concepts, e.g. $Surgeon \sqsubseteq Doctor$, and assertions about individuals or ABox (e.g. $Doctor(mary)$).

Interpretations for description logic same as for FOL: a set of individuals D and an interpretation mapping I such that

- for a constant a, $I(a) \in D$
- for an atomic concept A, $I(A) \subseteq D$
- for a role r, $I(r) \subseteq D \times D$
- $I(\forall r.C]) = \{x \in D : \text{for any } y, \text{ if } (x, y) \in I(r), \text{ then } y \in I(C)\}$. Same as

$$\forall y(R(x, y) \supset B(y))$$

- $I(\exists r.C]) = \{x \in D : \text{there is a } y \text{ such that } (x, y) \in I(r) \text{ and } y \in I(C)\}$. Same as

$$\exists y(R(x, y) \wedge C(y))$$

- $I(\neg C) = D \setminus I(C)$
- $I(C_1 \sqcap C_2]) = I(C_1) \cap I(C_2)$. Same as

$$C_1(x) \wedge C_2(x)$$

- $I(C_1 \sqcup C_2]) = I(C_1) \cup I(C_2)$. Same as

$$C_1(x) \vee C_2(x)$$

Finally, for sentences:

- $(D, I) \models C(a)$ iff $I(a) \in I(C)$. Same as $C(a)$
- $(D, I) \models r(a, b)$ iff $(I(a), I(b)) \in I(r)$. Same as $R(a, b)$
- $(D, I) \models C_1 \sqsubseteq C_2$ iff $I(C_1) \subseteq I(C_2)$. Same as

$$\forall x(C_1(x) \supset C_2(x))$$

- $(D, I) \models C_1 \doteq C_2$ iff $I(C_1) = I(C_2)$. Same as

$$\forall x(C_1(x) \equiv C_2(x))$$

Reasoning Entailment is defined exactly like in FOL: a set of sentences Γ entails a sentence ϕ (in symbols $\Gamma \models \phi$) if and only if ϕ is true in every interpretation where all of the sentences in Γ are true.

Since ALC is a fragment of first order logic, reasoning in it is more efficient (it is decidable whether a sentence is satisfiable, or whether a finite set of sentences entails another sentence). This holds for many other description logics, although not all of them.

G53KRR handout on defaults.

Defaults or default rules as opposed to normal or categorical rules are ways of drawing conclusions which are justified unless there is some explicit reason to believe otherwise. So normal rule will say 'if x is a natural number then it is greater or equal to 0' and this is really true without exceptions for all natural numbers. A default rule would say 'if x is a bird then it can fly (unless there are good reasons to believe otherwise)'. In other words, if all we know about x is that it is a bird, then it is reasonable to conclude that it can fly. Later however we may discover that it is a special kind of bird which does not fly.

Non-monotonicity In classical reasoning, entailment is *monotonic*: if $KB_1 \models \phi$ and $KB_1 \subseteq KB_2$, then $KB_2 \models \phi$. In other words, if ϕ is entailed by KB_1 and we add more sentences to KB_1, ϕ will still be entailed by the resulting knowledge base; the larger the knowledge base, the more consequences it has: if $KB_1 \subseteq KB_2$ then $Consequences(KB_1) \subseteq Consequences(KB_2)$.

Default reasoning is *nonmonotonic.* If we have $KB_1 =$ { 'Birds normally can fly', 'Tweety is a bird' } then we can derive by default that Tweety can fly. However, if we learn more about Tweety, for example that it is a penguin, then 'Tweety can fly' no longer follows even by default.

The question is, how to make this work formally (define what are valid default consequences)? In these two lectures, consider three approaches: closed-world assumption, circumscription, default logic.

Closed-world reasoning *Closed-world assumption* (CWA): if an atomic sentence is not in the knowledge base, it is assumed to be false. (Like negation as failure in production rule systems: if a fact is not in the working memory, then its negation matches/is assumed to be true.) The corresponding entailment $\models_{CWA}$:

$$KB \models_{CWA} \phi \;\Leftrightarrow\; KB^+ \models \phi$$

where $KB^+ = KB \cup \{\neg p : p \text{ is atomic and } KB \not\models p\}$.

If $KB = \{Bird(t)\}$ then $\neg Penguin(t)$ follows under CWA.

Problems: if $KB = \{p \vee q\}$, KB^+ is inconsistent since it contains $p \vee q$, $\neg p$ and $\neg q$, so everything follows from it.

Generalised CWA is a fix for this:

$$KB \models_{GCWA} \phi \;\Leftrightarrow\; KB^* \models \phi$$

where $KB^* = KB \cup \{\neg p : p$ is atomic and for all collections of atoms $q_1, \ldots, q_n$, if $KB \models p \vee q_1 \vee \ldots \vee q_n$, then $KB \models q_1 \vee \ldots \vee q_n\}$.

CWA with domain closure: only explicitly named individuals are assumed to exist:

$$KB \models_{CD} \phi \;\Leftrightarrow\; KB^\diamond \models \phi$$

where $KB^\diamond = KB^+ \cup \forall x(x = c_1 \vee \ldots \vee x = c_n)$ where $c_1, \ldots, c_n$ are all the constant symbols appearing in KB.

Under CWA with domain closure, if there is no fact $P(a)$ in the knowledge base then it entails by default $\neg\exists x P(x)$.

Unique name assumption: $c \neq c'$ for any two distinct constants c, c'.

Reasoning under CWA is constructive and reasonably efficient.

Circumscription (John McCarthy). This is a generalisation of CWA: for default entailment, consider not all models of KB but only those where the set of exceptions is made as small as possible. Namely, consider a predicate Ab (for abnormal) and the formulation of a default rule as

$$\forall x(Bird(x) \wedge \neg Ab(x) \supset Flies(x))$$

and say that a conclusion follows by default if it is entailed on all interpretations where the extension of Ab is as small as possible. (This is called *circumscribing* Ab and the approach is called *circumscription*.) We need one Ab for every default rule, because a bird which is abnormal with respect to flying may be normal with respect to having two legs etc.

Let A be the set of Ab predicates we want to minimise. Let $M_1 = (D, I_1)$ and $M_2 = (D, I_2)$ be two interpretations over the same domain such that every constant and function are interpreted the same way.

$$M_1 \leq M_2 \;\Leftrightarrow\; \forall Ab \in A\ (I_1(Ab) \subseteq I_2(Ab))$$

$M_1 < M_2$ if $M_1 \leq M_2$ but not $M_2 \leq M_1$. (There are strictly fewer abnormal things in M_1).

Minimal entailment: $KB \models_{\leq} \phi$ iff for all interpretations M which make KB true, either $M \models \phi$ or M is not minimal (exists M' such that $M' < M$ and $M' \models KB$).

Example:

$$KB = \{Bird(chilly), Bird(tweety), (tweety \neq chilly), \neg Flies(chilly), \forall x(Bird(x) \wedge \neg Ab(x) \supset Flies(x))\}$$

$KB \not\models Flies(tweety)$ but $KB \models_{\leq} Flies(tweety)$. This knowledge base has a unique minimal extension for Ab, $I(Ab) = \{chilly\}$. This is not always the case. For example, if instead of $\neg Flies(chilly)$ it had $\neg Flies(chilly) \vee \neg Flies(tweety)$ there would be two minimal extensions of Ab: one where Tweety is abnormal and another where Chilly is abnormal.

Different from CWA: cannot replicate this effect by adding a fixed set of negated atomic sentences to KB.

Circumscription has constructive reasoning procedures but complexity is high.

Default logic Roy Reiter. A *default rule* consists of a *prerequisite* α, *justification* β, *conclusion* γ and says 'if α holds and it is consistent to believe β, then believe γ': $\dfrac{\alpha : \beta}{\gamma}$

For example:

$$\frac{Bird(x) : Flies(x)}{Flies(x)}$$

Default rules where justification and conclusion are the same are called *normal default rules* and are writted $Bird(x) \Rightarrow Flies(x)$.

Given a *default theory* $KB = \{F, D\}$, where F is a finite set of first order sentences and D is a finite set of default rules, what is the set of reasonable beliefs (extension of the default theory)?

E is an *extension* of (F, D) iff for every sentence π,

$$\pi \in E \;\Leftrightarrow\; F \cup \{\gamma \mid \frac{\alpha : \beta}{\gamma} \in D, \alpha \in E, \neg\beta \notin E\} \models \pi$$

where $\{\gamma \mid \frac{\alpha:\beta}{\gamma} \in D, \alpha \in E, \neg\beta \notin E\}$ is a set of *applicable assumptions* for this extension.

Here is a non-deterministic procedure to produce an extension E of a theory (F, D):

0. $E = F$

Repeat until no change to E:

1. Close E under logical consequence: $E = E \cup \{\alpha : E \models \alpha\}$
2. Choose $\frac{\alpha:\beta}{\gamma}$ in D such that for some substitution θ, $\alpha\theta \in E$ and $\neg\beta\theta \notin E$.

Set $E = E \cup \{\gamma\theta\}$

Go to 1.

Example: $F = \{Bird(tweety), Bird(chilly), \neg Flies(chilly)\}$, $D = \{Bird(x) \Rightarrow Flies(x)\}$. The only applicable assumption is $Flies(tweety)$ (prerequisite in F hence in E, negation of justification not in E). For Chilly, negation of justification is in E, and for no other object o we can have $Bird(o)$ in E. So E contains the consequences of $F \cup \{Flies(tweety)\}$. E is the only extension, and we can say that $Flies(tweety)$ is a default consequence of (F, D).

A theory may have multiple extensions:

Facts: $F = \{Republican(dick), Quaker(dick)\}$

Default rules: $Republican(x) \Rightarrow \neg Pacifist(x)$, $Quaker(x) \Rightarrow Pacifist(x)$.

Two extensions: E_1 containing $Pacifist(dick)$, E_2 containing $\neg Pacifist(dick)$. Can be forced to make only one extension using a non-normal default rule:

$$\frac{Quaker(x) : Pacifist(x) \wedge \neg MemberOfPoliticalParty(x)}{Pacifist(x)}$$

and a rule $\forall x(Republican(x) \supset MemberOfPoliticalParty(x))$.

skeptical reasoner will only believe sentences which belong to all extensions of the default theory; *credulous* reasoner will choose an arbitrary extension.

G53KRR handout on Bayesian networks.

Bayesian approach (subjective probability) The basic idea is that we assign degrees of belief (subjective probabilities) to statements like "Tweety can fly" or "Patient a has disease b". Both sentences are either true or false in the real world, so standard objective statistical probabilities don't apply. However we can base our degree of belief on statistical information: namely, if 95% of birds can fly we may believe with degree 95% that a particular bird Tweety can fly. This will be an a priori degree of belief, before we know anything else about Tweety. Once we discover other facts about Tweety, our belief will be based on the conditional probability that Tweety flies given other facts (that it is a penguin for example).

Probability axioms and useful rules. Given a universal set U of all possible event occurrences (here we assume it is finite), an event a is a subset of U. For example, if U is a large set of medical histories, a is a subset of them where the patient has been diagnosed with flu. A probability function Pr is a function from events to numbers in $[0, 1]$ satisfying the folowing postulates:

1. $Pr(U) = 1$
2. If $a_1, \ldots, a_n$ are disjoint events, $Pr(a_1 \cup \ldots \cup a_n) = Pr(a_1) + \ldots + Pr(a_n)$.

Some consequences:

- $Pr(\bar{a}) = 1 - Pr(a)$ ($\bar{a}$ is the complement of a)
- $Pr(\emptyset) = 0$
- $Pr(a \cup b) = Pr(a) + Pr(b) - Pr(a \cap b)$

Conditional probability:

$$Pr(a \mid b) = \frac{Pr(a \wedge b)}{Pr(b)}$$

Conditionaly independent events: $Pr(a \mid b) = Pr(a)$.
a and b are conditionally independent given c: $Pr(a \mid b \cap c) = Pr(a \mid c)$.
Conditional version of the negation rule $Pr(\bar{a} \mid b) = 1 - Pr(a \mid b)$.

Bayes' rule

$$Pr(a \mid b) = \frac{Pr(a) \cdot Pr(b \mid a)}{Pr(b)}$$

If a is a disease and b a symptom, and we want to know the probability that someone has the disease given they have the symptom; it is easier to find the a priori probability of the disease and what is the probability that a patient who has the disease will display the symptom, and the a priori probability of the symptom.

Probabilities of sentences (basic Bayesian approach) Suppose there are n propositional variables of interest: $p_1, \ldots, p_n$ (corresponding to sentences like "Tweety flies" or "John has flu"). There are 2^n possible states of the world (truth assignments to those variables). J is a *joint probability distribution* if for every assignment I, $J(I)$ is a number between 0 and 1 and $\Sigma\ J(I) = 1$ (the probability that one of the assignments corresponds to reality is 1). The probability of a sentence α is the sum of probabilities of the worlds where α is true:

$$Pr(\alpha) = \Sigma_{I \models \alpha} J(I)$$

We can now find the probability of any sentence. Unfortunately, this requires us to keep 2^n numbers (probability of each assignment).

Let us represent an assignment to $\{p_1, \ldots, p_n\}$ as $\langle P_1, \ldots, P_n \rangle$ where P_i is p_i if p_i is assigned true and $\neg p_i$ otherwise. For example, an assignment which assigns true to p_1 and false to p_2 can be represented as $\langle p_1, \neg p_2 \rangle$.

$$J(\langle P_1, \ldots, P_n \rangle) = Pr(P_1 \wedge \ldots \wedge P_n)$$

By the chain rule (which follows from the definition of conditional probability),

$$Pr(P_1 \wedge \ldots \wedge P_n) = Pr(P_1) \cdot Pr(P_2|P_1) \cdots Pr(P_n|P_1 \wedge \ldots \wedge P_{n-1})$$

If all the variables were conditionally independent of each other:

$$Pr(P_i|P_1 \wedge \ldots \wedge P_{i-1}) = Pr(P_i)$$

then we could compute probabilities of each interpretation from n numbers. But normally we cannot assume that all variables are conditionally independent.

Belief networks The idea is to represent explicitly which variables *are* conditionally dependent on each other. The network is a directed acyclic graph. The nodes in the network are variables p_i and there is an arc from p_i to p_j if p_j is conditionally dependent on p_i (its probability given p_i is different from its prior probability).

If there is an arc from p_i to p_j we call p_i a *parent* of p_j in the network.

Each propositional variable in the belief network is conditionally independent from non-parent variables given its parent variables:

$$Pr(P_i \mid P_1 \wedge \ldots \wedge P_{i-1}) = Pr(P_i \mid parents(P_i))$$

where $parents(P_i)$ is the conjunction of literals which correspond to parents of p_i in the network.

Exercise Do exercise 2 after Chapter 12:

Consider the following example: *Metastatic cancer is a possible cause of a brain tumor and is also an explanation for an increased total serum calcium. In turn, either of these could cause a patient to fall into occasional coma. Severe headache could also be explained by a brain tumor.*

(a) Represent these causal links in a belief network. Let a stand for 'metastatic cancer', b for 'increased total serum calcium', c for 'brain tumor', d for 'occasional coma', and e for 'severe headaches'.

(b) Give an example of an independence assumption that is implicit in this network.

(c) Suppose the following probabilities are given: $Pr(a) = 0.2, Pr(b|a) = 0.8, Pr(b|\neg a) = 0.2, Pr(c|a) = 0.2, Pr(c|\neg a) = 0.05, Pr(e|c) = 0.8, Pr(e|\neg c) = 0.6, Pr(d|b \wedge c) = 0.8, Pr(d|b \wedge \neg c) = 0.8, Pr(d|\neg b \wedge c) = 0.8, Pr(d|\neg b \wedge \neg c) = 0.05$ and assume that it is also given that some patient is suffering from severe headaches but has not fallen into a coma. Calculate joint probabilities for the eight remaining possibilities (that is, according to whether a, b, and c are true or false).

(d) According to the numbers given, the a priori probability that the patient has metastatic cancer is 0.2. Given that the patient is suffering from severe headaches but has not fallen into a coma, are we now more or less inclined to believe that the patient has cancer? Explain.

G53KRR exercise on Bayesian networks.

This is exercise 2 after Chapter 12 in Brachman and Levesque's book:

Consider the following example: *Metastatic cancer is a possible cause of a brain tumor and is also an explanation for an increased total serum calcium. In turn, either of these could cause a patient to fall into occasional coma. Severe headache could also be explained by a brain tumor.*

(a) Represent these causal links in a belief network. Let a stand for 'metastatic cancer', b for 'increased total serum calcium', c for 'brain tumor', d for 'occasional coma', and e for 'severe headaches'.

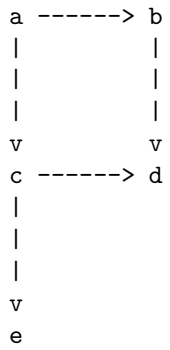
(b) Give an example of an independence assumption that is implicit in this network.

(c) Suppose the following probabilities are given: $Pr(a) = 0.2, Pr(b|a) = 0.8, Pr(b|\neg a) = 0.2, Pr(c|a) = 0.2, Pr(c|\neg a) = 0.05, Pr(e|c) = 0.8, Pr(e|\neg c) = 0.6, Pr(d|b \wedge c) = 0.8, Pr(d|b \wedge \neg c) = 0.8, Pr(d|\neg b \wedge c) = 0.8, Pr(d|\neg b \wedge \neg c) = 0.05$ and assume that it is also given that some patient is suffering from severe headaches but has not fallen into a coma. Calculate joint probabilities for the eight remaining possibilities (that is, according to whether a, b, and c are true or false).

(d) According to the numbers given, the a priori probability that the patient has metastatic cancer is 0.2. Given that the patient is suffering from severe headaches but has not fallen into a coma, are we now more or less inclined to believe that the patient has cancer? Explain.

Answers :

(a) Sorry for an ascii drawing. The main thing here is that arcs go from cause (e.g. brain tumor) to effect (e.g. headache). Other layouts like the one I did on the board are OK too.



(b) Examples are:

- $Pr(c \mid a \wedge b) = Pr(c \mid a)$, $Pr(c \mid \neg a \wedge b) = Pr(c \mid \neg a)$ etc.
- $Pr(d \mid a \wedge b \wedge c) = Pr(d \mid b \wedge c)$
- $Pr(e \mid a \wedge b \wedge c \wedge d) = Pr(e \mid c)$

(c) I spell out the computation of the probability of the first conjunction in more detail, after that I will skip the chain rule and use the negation rule without mentioning it.

1. $Pr(a \wedge b \wedge c \wedge \neg d \wedge e) =$ (using the normal chain rule)
   $Pr(a) \cdot Pr(b \mid a) \cdot Pr(c \mid a \wedge b) \cdot Pr(\neg d \mid a \wedge b \wedge c) \cdot Pr(e \mid a \wedge b \wedge c \wedge \neg d) =$ (substituting conditional probabilities using independence assumptions of the network)
   $Pr(a) \cdot Pr(b \mid a) \cdot Pr(c \mid a) \cdot Pr(\neg d \mid b \wedge c) \cdot Pr(e \mid c) =$
   (using the negation rule $Pr(\neg d \mid b \wedge c) = 1 - Pr(d \mid b \wedge c)$)
   $Pr(a) \cdot Pr(b \mid a) \cdot Pr(c \mid a) \cdot (1 - Pr(d \mid b \wedge c)) \cdot Pr(e \mid c) =$
   $0.2 \cdot 0.8 \cdot 0.2 \cdot 0.2 \cdot 0.8 = 0.00512$
2. $Pr(a \wedge b \wedge \neg c \wedge \neg d \wedge e) =$
   $Pr(a) \cdot Pr(b \mid a) \cdot (1 - Pr(c \mid a)) \cdot (1 - Pr(d \mid b \wedge \neg c)) \cdot Pr(e \mid \neg c) =$
   $0.2 \cdot 0.8 \cdot 0.8 \cdot 0.2 \cdot 0.6 = 0.01536$
3. $Pr(a \wedge \neg b \wedge c \wedge \neg d \wedge e) =$
   $Pr(a) \cdot (1 - Pr(b \mid a)) \cdot Pr(c \mid a) \cdot (1 - Pr(d \mid \neg b \wedge c)) \cdot Pr(e \mid c) =$
   $0.2 \cdot 0.2 \cdot 0.2 \cdot 0.2 \cdot 0.8 = 0.00128$
4. $Pr(a \wedge \neg b \wedge \neg c \wedge \neg d \wedge e) =$
   $Pr(a) \cdot (1 - Pr(b \mid a)) \cdot (1 - Pr(c \mid a)) \cdot (1 - Pr(d \mid \neg b \wedge \neg c)) \cdot Pr(e \mid \neg c) =$
   $0.2 \cdot 0.2 \cdot 0.8 \cdot 0.95 \cdot 0.6 = 0.01824$
5. $Pr(\neg a \wedge b \wedge c \wedge \neg d \wedge e) =$
   $(1 - Pr(a)) \cdot Pr(b \mid \neg a) \cdot Pr(c \mid \neg a) \cdot (1 - Pr(d \mid b \wedge c)) \cdot Pr(e \mid c) =$
   $0.8 \cdot 0.2 \cdot 0.05 \cdot 0.2 \cdot 0.8 = 0.00128$
6. $Pr(\neg a \wedge b \wedge \neg c \wedge \neg d \wedge e) =$
   $(1 - Pr(a)) \cdot Pr(b \mid \neg a) \cdot (1 - Pr(c \mid \neg a)) \cdot (1 - Pr(d \mid b \wedge \neg c)) \cdot Pr(e \mid \neg c) =$
   $0.8 \cdot 0.2 \cdot 0.95 \cdot 0.2 \cdot 0.6 = 0.01824$
7. $Pr(\neg a \wedge \neg b \wedge c \wedge \neg d \wedge e) =$
   $(1 - Pr(a)) \cdot (1 - Pr(b \mid \neg a)) \cdot Pr(c \mid \neg a) \cdot (1 - Pr(d \mid \neg b \wedge c)) \cdot Pr(e \mid c) =$
   $0.8 \cdot 0.8 \cdot 0.05 \cdot 0.2 \cdot 0.8 = 0.00512$
8. $Pr(\neg a \wedge \neg b \wedge \neg c \wedge \neg d \wedge e) =$
   $(1 - Pr(a)) \cdot (1 - Pr(b \mid \neg a)) \cdot (1 - Pr(c \mid \neg a)) \cdot (1 - Pr(d \mid b \wedge \neg c)) \cdot Pr(e \mid \neg c) =$
   $0.8 \cdot 0.8 \cdot 0.95 \cdot 0.95 \cdot 0.6 = 0.34656$

(d) We are asked whether $Pr(a \mid \neg d \wedge e)$ is greater or smaller than $Pr(a)$.

$Pr(a \mid \neg d \wedge e) = Pr(a \wedge \neg d \wedge e)/Pr(\neg d \wedge e)$ (conditional probability definition). We need to compute $Pr(a \wedge \neg d \wedge e)$ and $Pr(\neg d \wedge e)$, and to do that we use the probabilities we computed above. They describe all 8 possible states of the world given that $\neg d$ and e are true, and they are all disjoint. We are using $Pr(X) = Pr(X \wedge Y) + Pr(X \wedge \neg Y)$, or that the probability of the union of disjoint events equals to the sum of probabilities of those events.

So $Pr(a \wedge \neg d \wedge e) = Pr(a \wedge b \wedge c \wedge \neg d \wedge e) + Pr(a \wedge b \wedge \neg c \wedge \neg d \wedge e) + Pr(a \wedge \neg b \wedge c \wedge \neg d \wedge e) + Pr(a \wedge \neg b \wedge \neg c \wedge \neg d \wedge e)$ and $Pr(\neg d \wedge e)$ is the sum of all 8 numbers above.

$Pr(a \wedge \neg d \wedge e) = 0.04$

$Pr(\neg d \wedge e) = 0.04 + 0.00128 + 0.01824 + 0.00512 + 0.34656 = 0.4112$

$Pr(a \mid \neg d \wedge e) = 0.04/0.4112$ which is approximately 0.1. So the probability got smaller.

Knowledge representation and reasoning
Lecture 17: Reasoning about actions

Natasha Alechina
natasha.alechina@nottingham.ac.uk

Situation Calculus

- Situation calculus is a dialect of FOL where situations (static states of the world) and actions are basic terms:
- variables over situations are denoted $s, s_1, s_2, \ldots,$.
- a distinguished initial situation is denoted by a constant S_0.
- actions are terms like $move(x, y, z)$ (move thing z to coordinates x, y) etc. Note that actions are also terms, not formulas: they denote an 'action' and are not true or false.
- a special function *do* takes an action and a situation and returns a new situation: $do(a, s)$ denotes a new situation which results from performing an action a in a situation s.

Fluents

- Predicates and functions whose values vary from situation to situation are called fluents.
- Last argument in a fluent is a situation:
- $\neg Holding(r, x, s) \wedge Holding(r, x, do(pickup(r, x), s))$ (a robot r is not holding x in s but is holding it in the situation resulting from s by picking x up).
- A distinguished fluent *Poss* says which actions are possible in a given situation:
  $Poss(pickup(r, blockA), S_0)$

Preconditions of actions

- A precondition is a condition which makes an action possible.
- It can be expressed by a formula, sometimes called precondition axiom.
- For example:

$$\forall r \forall x \forall s\ (Poss(pickup(r, x), s) \equiv \forall z(\neg Holding(r, z, s) \wedge \neg Heavy(x) \wedge NextTo(r, x, s)))$$

(a robot can pick x up if it is not holding anything else, x is not heavy, and the robot is next to it).

Postconditions or effects of actions

- A postcondition or effect of an action is a change resulting from executing the action.
- Formulas expressing postconditions are sometimes called effect axioms.
- For example, $\forall x \forall s \forall r(Fragile(x) \supset Broken(x, do(drop(r, x), s)))$
- Effect axioms for fluents which become true as a result of an action are called positive, and those where the fluent becomes false are called negative.

Frame axioms

- Classical planning assumption: actions are deterministic, and the world changes only as a result of clearly specified actions.
- For every action, we can also say which fluents it *does not* affect.
- The formulas which specify which properties are not changed as a result of an action are called frame axioms.
- For example,
  $\forall x \forall y \forall s \forall r$
  $(\neg Broken(x, s) \wedge (x \neq y \vee \neg Fragile(x)) \supset$
  $\neg Broken(x, do(drop(r, y), s)))$

Frame axioms continued

- Frame axioms do not logically follow from precondition and effect axioms.
- They are called frame axioms because they limit or frame the effects of actions.

Why do we need frame axioms 1

A typical kind of task in reasoning about actions is to check whether

- a certain sequence of actions $a_1, \ldots, a_n$ will succeed (bring about some desired state of the world)
- a certain sequence of actions is possible

In both cases, some relevant information about S_0 is given (which fluents hold in S_0).

Why do we need frame axioms 2

- The precondition and effects of actions are used to determine which fluents will be true in $do(a_n, do(a_{n-1}, \ldots do(a_1, S_0)\ldots)$.
- Some fluent may be a precondition of some action a_i which is true in S_0 and is unchanged by $a_1, \ldots, a_{i-1}$.
- However we cannot derive that it is unchanged from just the precondition and effect axioms for $a_1, \ldots, a_{i-1}$: need to also have explicit frame axioms.

Frame problem

- Frame problem is the problem of representing frame conditions coincisely (*not* with an axiom for each pair of action and fluent!).

Solution to the frame problem

- For each fluent $F(\bar{x}, s)$ (where $\bar{x}$ are all the free variables of the fluent) we collect together all positive effect axioms.
- For example, if $Broken(x, s)$ has two positive effect axioms:
  $\forall x \forall s\ (Fragile(x) \supset Broken(x, do(drop(x), s)))$

  $\forall x \forall s\ (Broken(x, do(break(x), s)))$
- and together they can be written as:
  $\forall x \forall a \forall s((Fragile(x) \wedge a = drop(x)) \vee (a = break(x))$
  $\supset Broken(x, do(a, s)))$
- In general, have an expression
  $\forall \bar{x} \forall a \forall s(\Pi_F(\bar{x}, a, s) \supset F(\bar{x}, do(a, s)))$

Solution to the frame problem continued

- Same for the negative effect axioms:
  $\forall \bar{x} \forall a \forall s(N_F(\bar{x}, a, s) \supset \neg F(\bar{x}, do(a, s)))$
- For example:
  $\forall \bar{x} \forall a \forall s(a = fix(x) \supset \neg Broken(x, do(a, s)))$

Solution to the frame problem continued

- Once we have a single formula Π_F for all actions which make $F(x, s)$ true and a single formula N_F for all actions which make F false, we can write explanation closure axioms:
  $\forall \bar{x} \forall a \forall s(\neg F(\bar{x}, s) \wedge F(\bar{x}, do(a, s)) \supset \Pi_F(\bar{x}, a, s))$
  $\forall \bar{x} \forall a \forall s(F(\bar{x}, s) \wedge \neg F(\bar{x}, do(a, s)) \supset N_F(\bar{x}, a, s))$
- They *replace all frame axioms* by saying that
  F only becomes true if Π_F holds (only certain actions in certain circumstances make F true)
  F only becomes false if N_F is true
- Π_F and N_F are short, and explanation axioms entail all the frame axioms (under the assumptions of deterministic actions and only change as a result of actions).

Successor state axioms

- If some additional assumptions hold, namely:
  1. no action has both a positive and negative effect on a fluent F,
  2. action terms can only be equal if they have the same action name applied to the same arguments

  then explanation closure axioms can be combined into a successor state axiom for a fluent:
  $\forall \bar{x} \forall a \forall s(F(\bar{x}, do(a, s)) \equiv \Pi_F(\bar{x}, a, s) \vee (F(\bar{x}, s) \wedge \neg N_F(\bar{x}, a, s)))$
- Under those assumptions, all that is needed to solve the frame problem and describe the actions and fluents completely are: precondition axioms and successor state axioms

Summary: Describing actions I

- "Effect" axiom—describe changes due to action
  $\forall s\ (AtGold(s) \supset Holding(Gold, do(Grab, s)))$
- "Frame" axiom—describe *non-changes* due to action
  $\forall s\ (HaveArrow(s) \supset HaveArrow(do(Grab, s)))$
- Frame problem: find an elegant way to handle non-change
  (a) representation—avoid frame axioms
  (b) inference—avoid repeated "copy-overs" to keep track of state
- Qualification problem: true descriptions of real actions require endless caveats—what if gold is slippery or nailed down or . . .
- Ramification problem: real actions have many secondary consequences—what about the dust on the gold, wear and tear on gloves, . . .

Describing actions II

- Successor-state axioms solve the representational frame problem
- Each axiom is about a *predicate* rather than about an action:

  P true afterwards $\equiv$ an action made P true or P true already and no action made P false

- For holding the gold: $\forall a \forall s Holding(Gold, do(a, s)) \equiv$
  $[(a = Grab \wedge AtGold(s)) \vee (Holding(Gold, s) \wedge a \neq Release)]$

Making plans in situation calculus

- Initial condition in KB:
  $At(Agent, [1,1], S_0)$
  $At(Gold, [1,2], S_0)$
- Query: $KB \models (?)\exists s\ Holding(Gold, s))$ i.e., in what situation will I be holding the gold?
- Answer: $s/do(Grab, do(Forward, S_0))$ i.e., go forward and then grab the gold

Making plans: A better way

- Represent plans as action sequences $[a_1, a_2, \ldots, a_n]$
- $doPlan(p, s)$ is the result of executing p in s
- Then the query $(KB, \exists p\ (Holding(Gold, doPlan(p, S_0))))$ has the solution $p/[Forward, Grab]$
- Definition of *doPlan* in terms of *do*: $\forall s doPlan([\,], s) = s\ \forall a \forall p \forall s(doPlan([a|p], s) = doPlan(p, do(a, s))$
- Planning systems are special-purpose reasoners designed to do this type of inference more efficiently than a general-purpose reasoner

Summary

- Situation calculus provides conventions for describing actions and change in FOL
- can formulate planning as inference on a situation calculus KB
- Next lecture: Planning.
  Brachman and Levesque, Chapter 15.
  Russell and Norvig, 3rd ed., Chapter 10.1-10.2. (or other editions, Classical Planning).

Knowledge representation and reasoning
Lecture 18: Planning

Natasha Alechina
natasha.alechina@nottingham.ac.uk

What is Planning

- Planning is a reasoning problem:
- what actions (plan) to perform in order to make some condition (goal) true
- Planning is central to AI as the study of intelligent behaviour achieved through computational means
- In general, in real world, it is a very difficult problem

STRIPS operators

- STRIPS planning language (Fikes and Nilsson, 1971, derives from work at SRI International on robot called Shakey)
- goals are conjunctions of atoms (positive literals). We will often replace conjunctions of literals with sets of literals (meaning, all of the literals in the set are true)
- actions descriptions (action schemas) assume finite preconditions and effects of fixed form:
  - Precondition: conjunction of positive literals (we will write it as a set)
  - Effect: conjunction of literals (or a set of literals)

ACTION: $buy(x)$
PRECONDITION: $At(p), Sells(p, x)$
EFFECT: $Have(x)$

PDDL

- Planning Domain Definition Language
- Less restrictive than STRIPS
- Preconditions and goals can contain negative literals
- Other levels in PDDL also allow action durations, resource requirements etc. (not in this lecture)

Add and Delete lists

- Given an action schema

  ACTION: a
  PRECONDITION: some literals
  EFFECT: $E_1, \ldots, E_m$

- $Add(a) = \{E \mid E$ is a positive literal in EFFECT$\}$ (positive effects of a)

- $Del(a) = \{P \mid E = \neg P$ where $E \in$ EFFECT$\}$ (atoms appearing with negation in the effect of a)

Planning domain

- Planning domain is described by giving a list of fluents and action schemas
- Fluents are predicates, have no situation argument
- States are sets of ground fluents; fluents which are not mentioned in a state description are false (closed world assumption)
- a is possible in s if precondition of a is true in s
- the state resulting from executing a in s,

$$do(s, a) = (s - Del(a)) \cup Add(a)$$

Example (slightly modified)

ACTION: $buy(x)$
PRECONDITION: $At(p), Sells(p, x), Have(Money)$
EFFECT: $Have(x), \neg Have(Money)$

- $Del(buy(Jaguar)) = \{Have(Money)\}$
- $Add(buy(Jaguar)) = \{Have(Jaguar)\}$
- If $s =$ $\{At(JDealer), Sells(JDealer, Jaguar), Blue(Sky), Have(Money)\}$,
- $buy(Jaguar)$ is possible in s
- $do(s, buy(Jaguar) = (s - \{Have(Money)\}) \cup \{Have(Jaguar)\} =$ $\{At(JDealer), Sells(JDealer, Jaguar), Blue(Sky), Have(Jaguar)\}$

Planning problem

- Planning problem = planning domain + objects + initial state + goal
- Goal is a conjunction of literals: $Have(Jaguar) \wedge \neg At(Jail)$
- Can solve planning problem using search

State and goal description

- State descriptions are always ground (no variables)
- Goal description may have variables: $At(x) \wedge Have(y)$
- A property with a variable such as $At(x)$ is satisfied at a state if there is a way of substituting an object for x so that the resulting formula is true in the state
- An atomic ground formula $At(Home)$ is true iff it is in the state description
- A negation of a ground atom $\neg At(G)$ is true iff the atom $At(G)$ is not in the state description.

Forward and regression planning

- Usual search: forward search from the initial state to a goal state
- Nothing prevents us from searching from a goal state back to the initial state
- Sometimes given the branching factor it is more efficient to search backward
- Motivating example: imagine trying to figure out how to get to some small place with few traffic connections from somewhere with a lot of traffic connections

Simple example of forward planning

Planning domain:

- Fluents : $At(x)$ (at place x), $Sells(x, y)$ (shop x sells y), $Have(x)$ (have x)
- Two action schemas:

  ACTION: $buy(x)$
  PRECONDITION: $At(p), Sells(p, x), Have(Money)$
  EFFECT: $Have(x), \neg Have(Money)$

  ACTION: $go(x, y)$
  PRECONDITION: $At(x), x \neq y$
  EFFECT: $At(y), \neg At(x)$

Simple example of forward planning 2

- Planning problem: planning domain above plus
- Objects: *Money*, *J* (for Jaguar), *Home*, *G* (for Garage)
- Initial state: $At(Home), Have(Money), Sells(G, J)$
- Goal state: $Have(J)$

Simple example of forward planning 3

$$s_1 = \{At(Home), Have(Money), Sells(G, J)\}$$

buy(x) *not available for any* x *(don't have* $Sells(Home, x)$*)*

$$\downarrow go(Home, G)$$

$$s_2 = \{At(G), Have(Money), Sells(G, J)\}$$

$go(G, Home)$ *also available*

$$\downarrow buy(J)$$

$$s_3 = \{At(G), Have(J), Sells(G, J)\}$$

Heuristics for forward planning

- Similar to search, cf A^*: performance improves by orders of magnitude if a good heuristic is used
- Number of fluents in the goal which will be satisfied by the next action
- add more edges to the graph (make more actions possible), and use solutions to the resulting problem as a heuristic. (There are often more efficient algorithms to solve the relaxed problem.) Examples: remove (some) preconditions, ignore delete lists
- For example, in 8 puzzle assume tiles can move to occupied spaces = Manhattan distance heuristic
- abstract the problem (make the search space smaller)

Backward (regression) planning

- Also called relevant-states search
- Start at the goal state(s) and do regression (go back).
- To be precise, there we start with a ground goal description g which describes a set of states (all those where $Have(J)$ holds but $Have(Money)$ may or may not hold, for example).

Backward (regression) planning 2

- Given a goal description g and a ground action a, the regression from g over a gives a state description g':
- $g' = (g - Add(a)) \cup \{Precondition(a)\}$
- For example, if the goal is $Have(J)$, and a is $buy(J)$
- $g' = (\{Have(J)\} - \{Have(J)\}) \cup$
  $\{At(p), Sells(p, J), Have(Money)\} =$
  $\{At(p), Sells(p, J), Have(Money)\}$
- note that g' is partially uninstantiated (p is a free variable). In our example, there is only one match for p, namely G, but in general there may be several.

Backward (regression) planning 3

- Which actions to regress over?
- Relevant actions: have an effect which is in the set of goal elements and no effect which negates an element of the goal.
- For example, $buy(J)$ is a relevant action.
- Search backwards from g, remembering the actions and checking whether we reached an expression applicable to the initial state.

Simple example of backward planning

$$Have(J)$$

$$\uparrow buy(J)$$

$$At(x), Have(Money), Sells(x, J)$$

Does not match the initial state yet

$$\uparrow go(y, x)$$

$$At(y), Have(Money), Sells(x, J)$$

Matches the initial state with $y/Home$ and x/G

Comparison of forward and backward planning

- Usually lots of actions available for forward planning
- Easier to find heuristics for forward planning
- Backward planning considers a lot fewer actions/relevant states than forward search, but uses sets of states (g, g') - hard to come up with good heuristics.

Use of logic and deduction

- In situation calculus, planning *is* deduction
- In 'normal' planning, we only need to check whether a state description entails some property, for example, $s \models P(A, B) \wedge \neg Q(A)$
- In simple cases, like in this lecture, this just involves checking that $P(A, B)$ is in the list of properties s has, and $Q(A)$ is not (closed world assumption: if $Q(A)$ is not listed, then $\neg Q(A)$ must be true)
- However, often planning domains are described using additional axioms, and then checking $s \models P(A, B)$ may involve more complex reasoning (whether $P(A, B)$ follows from the description of s and the axioms).

Knowledge representation and reasoning
Lecture 19: Planning 2

Natasha Alechina
natasha.alechina@nottingham.ac.uk

Planning domain

- Fluents or predicates which are used to describe preconditions and effects of actions
- For example, in the blocks world domain:
  - $Block(x)$: x is a block
  - $On(x, y)$: x is on y, where y is either another block or table
  - $Clear(x)$: there is nothing on top of block x
- Action schemas: available actions and their postconditions and effects. For example,

  ACTION: $moveToTable(b, x)$:
  PRECONDITION: $Block(b), Block(x), On(b, x), Clear(b)$
  EFFECT: $On(b, Table), Clear(x), \neg On(b, x)$

Example action schemas for blocks world

ACTION: $move(b, x, y)$:
PRECOND: $Block(b), Block(y), On(b, x), Clear(b), Clear(y), x \neq y$
EFFECT: $On(b, y), Clear(x), \neg On(b, x), \neg Clear(y)$

ACTION: $moveToTable(b, x)$:
PRECONDITION: $Block(b), Block(x), On(b, x), Clear(b)$
EFFECT: $On(b, Table), Clear(x), \neg On(b, x)$

Add and delete lists

- Given an action schema
  ACTION: a
  PRECONDITION: some literals
  EFFECT: $E_1, \ldots, E_m$
- $Add(a) = \{E \mid E$ is a positive literal in EFFECT$\}$ (positive effects of a)
- $Del(a) = \{P \mid E = \neg P$ where $E \in$ EFFECT$\}$ (atoms appearing with negation in the effect of a)

Add and delete list example

ACTION: $move(b, x, y)$:
PRECOND: $Block(b), Block(y), On(b, x), Clear(b), Clear(y), x \neq y$
EFFECT: $On(b, y), Clear(x), \neg On(b, x), \neg Clear(y)$

- $Add(move(b, x, y)) = \{On(b, y), Clear(x)\}$
- $Del(move(b, x, y)) = \{On(b, x), Clear(y)\}$

Planning problem

- Planning domain
- Objects (what can be substituted for variables)
- Initial state: a list of positive ground (no variables) literals, which are properties that hold in the state.
- Using Closed World Assumption, only describe what holds, and all the rest is assumed false
- Goal state: what literals should hold in the goal state. Can contain variables, which are assumed to be existentially quantifed.

Planning problem example

- planning domain as before
- objects: blocks A, B, C
- initial state: $\{Block(A), Block(B), Block(C), Clear(C), Clear(A), On(C, B), On(A, Table), On(B, Table)\}$. In this state, $On(A, Table)$ is true and $\neg On(C, Table)$ is true.
- Example of a non-ground goal: $On(x, B)$ as a goal means something should be on top of B; it is true in the initial state because x can be substituted for C.

Solving the planning problem

- Can be solved using search
- State space is not explicitly given; states are all possible sets of positive ground fluents
- Operators are actions; $do(s, a) = (s - Del(a)) \cup Add(a)$
- Plan = sequence of actions from initial state to a goal state, as in classical search

Forward or progressive planning

Algorithm from the textbook; S is current state, S' next (progressed) state

Input: a state and a goal
Output: a plan or fail.
ProgPlan[S,Goal] =
If Goal is satisfied in S, then return empty plan
For each operator *o* such that *precond*(*o*) is satisfied in S:
Let S' = S + add(o) - del(o)
. Let plan = ProgPlan[S',Goal]
. If plan $\neq$ fail, then return [act(o); plan]
End for
Return fail

Regression planning

Algorithm from the textbook; G is the goal, G' is regressed goal

Input: a state and a goal
Output: a plan or fail.
RegrPlan[S,Goal] =
If Goal is satisfied in S, then return empty plan
For each operator *o* such that $del(o) \cap Goal = \emptyset$
. Let Goal' = Goal + precond(o) - add(o)
. Let plan = RegrPlan[S,Goal']
. If plan $\neq$ fail, then return [plan; act(o)]
End for
Return fail

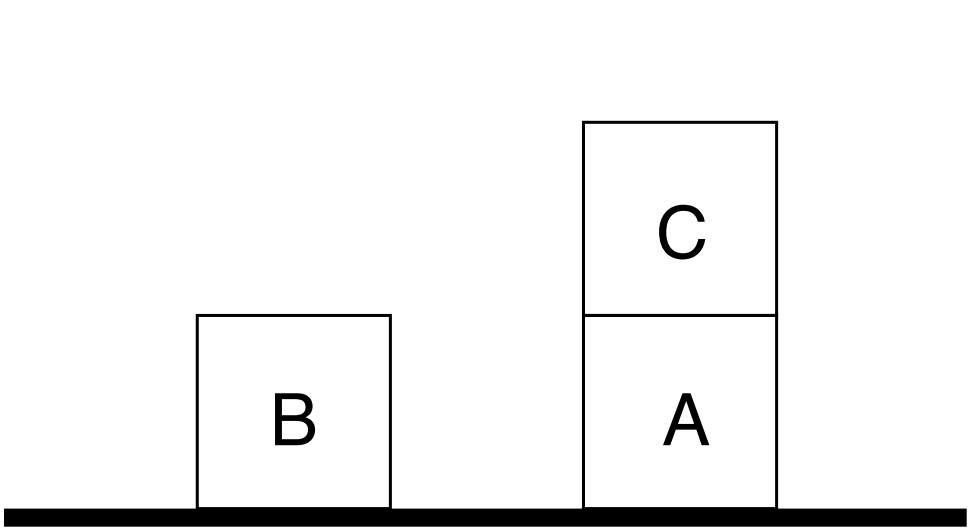
Goal stack planning

- One of the earlier planning algorithms called goal stack planning. It was used by STRIPS.
- We work backwards from the goal, looking for an operator which has one or more of the goal literals as one of its effects and then trying to satisfy the preconditions of the operator.
- The preconditions of the operator become subgoals that must be satisfied. We keep doing this until we reach the initial state.
- Goal stack planning uses a stack to hold goals and actions to satisfy the goals, and a knowledge base to hold the current state, action schemas and domain axioms
- Goal stack is like a node in a search tree; if there is a choice of action, we create branches

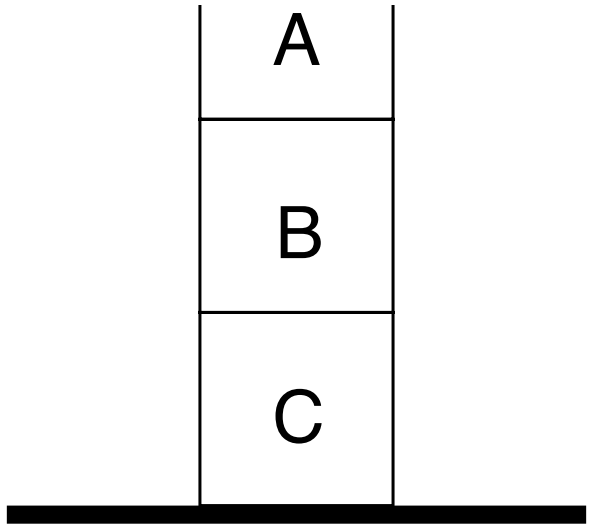
Goal stack planning pseudocode

- Push the original goal on the stack. Repeat until the stack is empty:
- If stack top is a compound goal, push its unsatisfied subgoals on the stack.
- If stack top is a single unsatisfied goal, replace it by an action that makes it satisfied and push the action's precondition on the stack.
- If stack top is an action, pop it from the stack, execute it and change the knowledge base by the action's effects.
- If stack top is a satisfied goal, pop it from the stack.

Goal stack planning example



Start



Goal

Goal stack planning 1

(I do pushing the subgoals at the same step as the compound goal.)
The order of subgoals is up to us; we could have put $On(B, C)$ on top

$$On(A, B)$$
$$On(B, C)$$
$$On(C, Table)$$
$$On(A, B) \wedge On(B, C) \wedge On(C, Table)$$

$S = \{On(C, A), On(A, Table), On(B, Table), Clear(B), Clear(C)\}$
plan = [ ]
The top of the stack is a single unsatisfied goal. We push the action which would achieve it, and its preconditions.

Goal stack planning 2

$On(A, Table)$
$Clear(B)$
$Clear(A)$
$On(A, Table) \wedge Clear(A) \wedge Clear(B)$
$move(A, x, B)$
$On(A, B)$
$On(B, C)$
$On(C, Table)$
$On(A, B) \wedge On(B, C) \wedge On(C, Table)$

$S = \{On(C, A), On(A, Table), On(B, Table), Clear(B), Clear(C)\}$
plan = [ ]
The top of the stack is a satisfied goal. We pop the stack (twice).

Goal stack planning 3

$$Clear(A)$$
$$On(A, Table) \wedge Clear(A) \wedge Clear(B)$$
$$move(A, Table, B)$$
$$On(A, B)$$
$$On(B, C)$$
$$On(C, Table)$$
$$On(A, B) \wedge On(B, C) \wedge On(C, Table)$$

$S = \{On(C, A), On(A, Table), On(B, Table), Clear(B), Clear(C)\}$
plan = [ ]
The top of the stack is an unsatisfied goal. We push the action which would achieve it, and its preconditions.

Goal stack planning 4

$On(C, A)$
$Clear(C)$
$On(C, A) \wedge Clear(C)$
$moveToTable(C, A)$
$Clear(A)$
$On(A, Table) \wedge Clear(A) \wedge Clear(B)$
$move(A, Table, B)$
$On(A, B)$
$On(B, C)$
$On(C, Table)$
$On(A, B) \wedge On(B, C) \wedge On(C, Table)$

$S = \{On(C, A), On(A, Table), On(B, Table), Clear(B), Clear(C)\}$
plan = [ ]
The top of the stack is a satisfied goal. We pop the stack (three times).

Goal stack planning 5

$moveToTable(C, A)$
$Clear(A)$
$On(A, Table) \wedge Clear(A) \wedge Clear(B)$
$move(A, Table, B)$
$On(A, B)$
$On(B, C)$
$On(C, Table)$
$On(A, B) \wedge On(B, C) \wedge On(C, Table)$

$S = \{On(C, A), On(A, Table), On(B, Table), Clear(B), Clear(C)\}$
plan = [ ]
The top of the stack is an action. We execute it, update the state with its effects, and add it to the plan.

Goal stack planning 6

$$Clear(A)$$
$$On(A, Table) \wedge Clear(A) \wedge Clear(B)$$
$$move(A, Table, B)$$
$$On(A, B)$$
$$On(B, C)$$
$$On(C, Table)$$
$$On(A, B) \wedge On(B, C) \wedge On(C, Table)$$

$S =$
$\{On(C, Table), On(A, Table), On(B, Table), Clear(A), Clear(B), Clear(C)\}$
plan = $[moveToTable(C, A)]$
The top of the stack is a satisfied goal. We pop the stack (twice).

Goal stack planning 7

$$move(A, Table, B)$$
$$On(A, B)$$
$$On(B, C)$$
$$On(C, Table)$$
$$On(A, B) \wedge On(B, C) \wedge On(C, Table)$$

$S =$
$\{On(C, Table), On(A, Table), On(B, Table), Clear(A), Clear(B), Clear(C)\}$
plan = $[moveToTable(C, A)]$
The top of the stack is an action. We execute it, update the state with its effects, and add it to the plan.

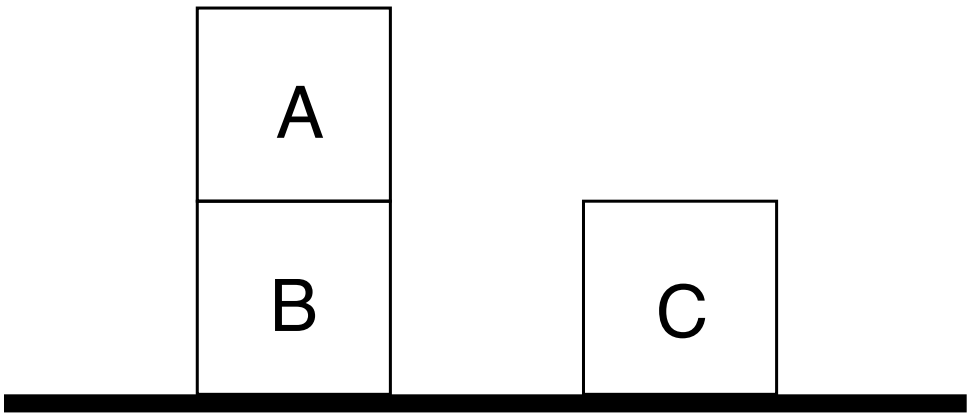
Goal stack planning 8

$On(A, B)$
$On(B, C)$
$On(C, Table)$
$On(A, B) \wedge On(B, C) \wedge On(C, Table)$

$S = \{On(C, Table), On(A, B), On(B, Table), Clear(A), Clear(C)\}$
plan = $[moveToTable(C, A), move(A, Table, B)]$
top of the stack is a satisfied goal

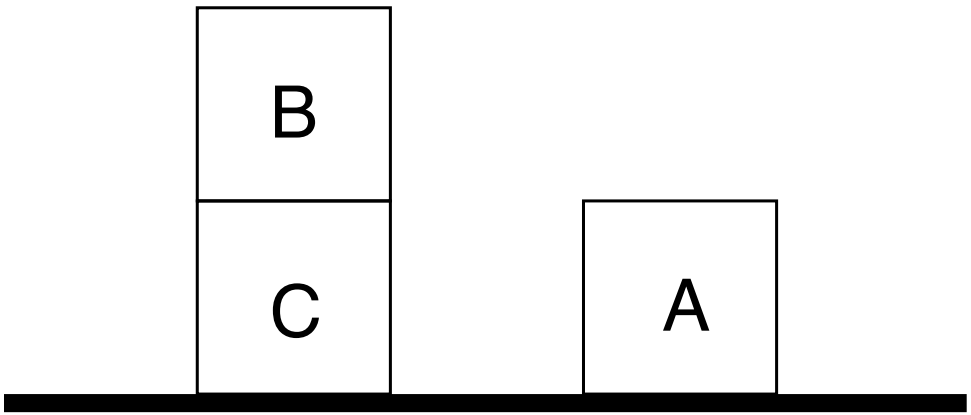
Goal stack planning 9a

the current state is



Goal stack planning 9b

If we follow the same process for the $On(B, C)$ goal, we end up in the state



Goal stack planning 10

Now we finally can move A on top of B, but the resulting plan is redundant:

$$moveToTable(C, A)$$
$$move(A, Table, B)$$
$$moveToTable(A, B)$$
$$move(B, Table, C)$$
$$move(A, Table, B)$$

There are techniques for ‘fixing’ inefficient plans (where something is done and then undone) but it is difficult in general (when it is not straight one after another)

Why this is an instructive example

- Sussman anomaly (called ‘anomaly’ because it seemed to make sense for a conjunctive goals to first achieve one goal and then achieve another goal, and then the complete goal would be achieved) is instructive,
- because it shows that although planning has the advantage over search in that states are factored and we can split the goal into subgoals, it is not straightforward
- achieving one goal ($On(A, B)$) destroys preconditions of an action which is necessary to achieve the other goal ($On(B, C)$), namely $Clear(B)$.
- Such interaction between actions is called clobbering.
- in the next lecture, I will talk about a solution to this problem (partial order planning)

Knowledge representation and reasoning
Lecture 20: Planning 3

Natasha Alechina
natasha.alechina@nottingham.ac.uk

Outline

- ♢ Totally vs partially ordered plans
- ♢ Partial-order planning
- ♢ Examples

Totally vs partially ordered plans

So far we produced a linear sequence of actions (totally ordered plan)

Often it does not matter in which order $some\ of\ the\ actions$ are executed

For problems with independent subproblems often easier to find a **partially ordered plan**: a plan which is a set of actions and a set of constraints $Before(a_i, a_j)$

Partially ordered plans are created by a search through a space of plans (rather than the state space)

Partially ordered plans

Partially ordered collection of steps with

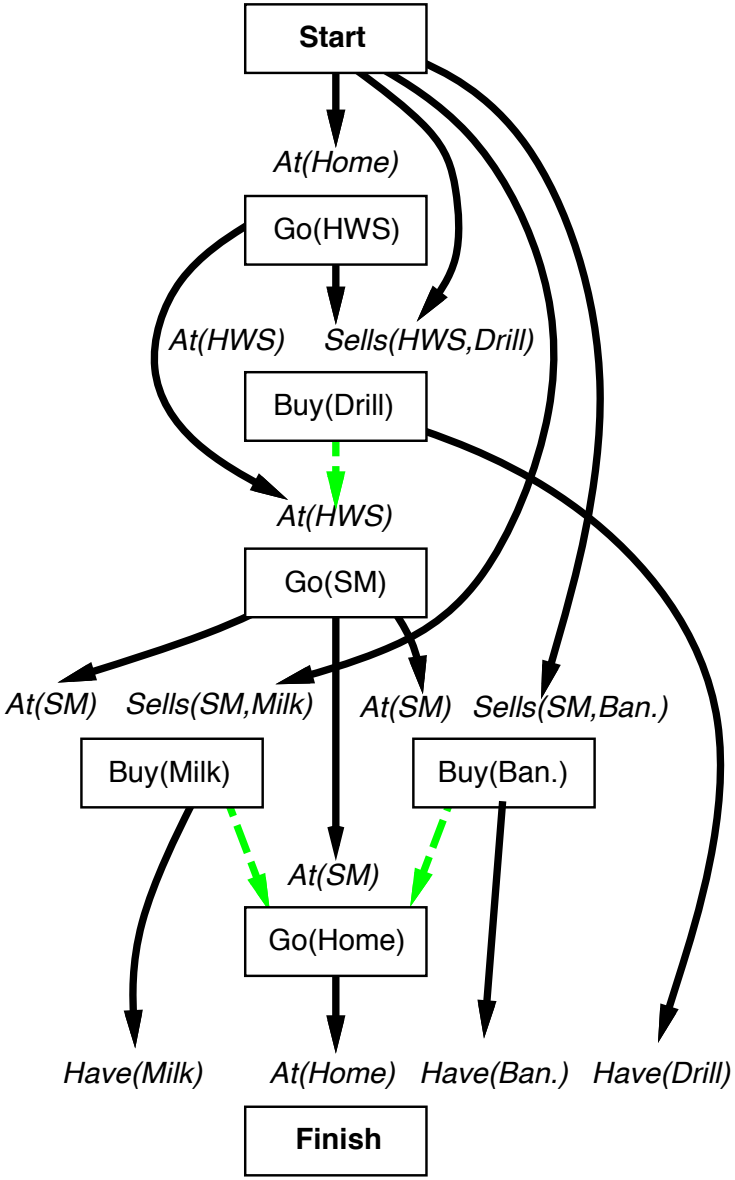
- $Start$ step has the initial state description as its effect
- $Finish$ step has the goal description as its precondition
- causal links from outcome of one step to precondition of another
- temporal ordering between pairs of steps

Open condition = precondition of a step not yet causally linked

A plan is complete iff every precondition is achieved

A precondition is achieved iff it is the effect of an earlier step
and no possibly intervening step undoes it

Example


Start
At(Home)
Go(HWS)
At(HWS)
Sells(HWS,Drill)
Buy(Drill)
At(HWS)
Go(SM)
At(SM)
Sells(SM,Milk)
At(SM)
Sells(SM,Ban.)
Buy(Milk)
Buy(Ban.)
At(SM)
Go(Home)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

Planning process

Operators on partial plans:

- add a link from an existing action to an open condition
- add a step to fulfill an open condition
- order one step wrt another to remove possible conflicts

Gradually move from incomplete/vague plans to complete, correct plans

Backtrack if an open condition is unachievable or
if a conflict is unresolvable

POP algorithm sketch

function POP(*initial, goal, actions*) **returns** *plan*

```
plan ← MAKE-MINIMAL-PLAN(initial, goal)
loop do
    if SOLUTION?(plan) then return plan
    S_need, c ← SELECT-SUBGOAL(plan)
    CHOOSE-ACTION(plan, actions, S_need, c)
    RESOLVE-THREATS(plan)
end
```

---

function SELECT-SUBGOAL(*plan*) **returns** S_{need}, c

pick a plan step S_{need} from STEPS(*plan*)
with a precondition c that has not been achieved

return S_{need}, c

POP algorithm contd.

procedure CHOOSE-ACTION($plan, actions, S_{need}, c$)

choose a step S_{add} from $actions$ or STEPS($plan$) that has c as an effect
if there is no such step **then fail**
add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS($plan$)
add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS($plan$)
if S_{add} is a newly added step from $actions$ **then**
    add S_{add} to STEPS($plan$)
    add $Start \prec S_{add} \prec Finish$ to ORDERINGS($plan$)

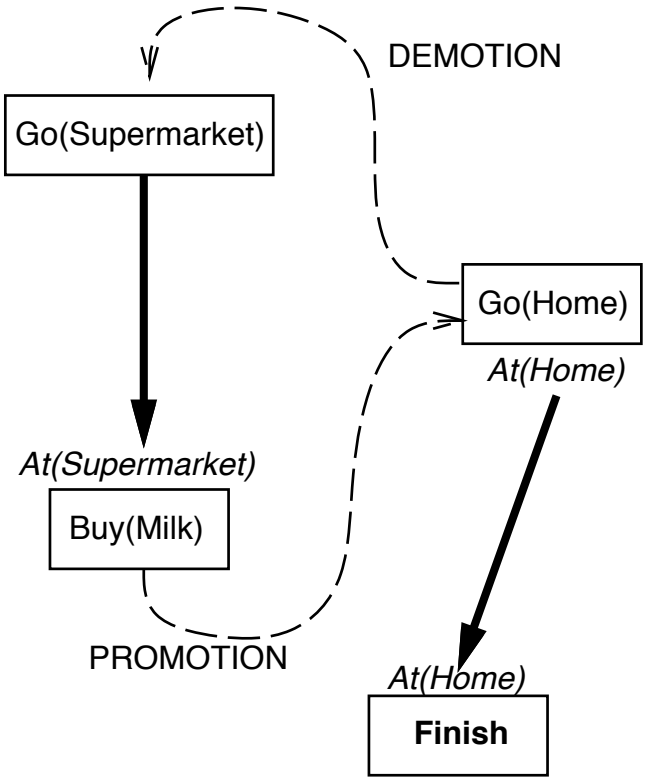
---

procedure RESOLVE-THREATS($plan$)

for each S_{threat} that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS($plan$) **do**
    choose either
        Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS($plan$)
        Promotion: Add $S_j \prec S_{threat}$ to ORDERINGS($plan$)
    if not CONSISTENT($plan$) **then fail**
end

Clobbering and promotion/demotion

A clobberer is a potentially intervening step that destroys the condition achieved by a causal link. E.g., $Go(Home)$ clobbers $At(Supermarket)$:



Demotion: put before $Go(Supermarket)$

Promotion: put after $Buy(Milk)$

Properties of POP

Nondeterministic algorithm: backtracks at choice points on failure:

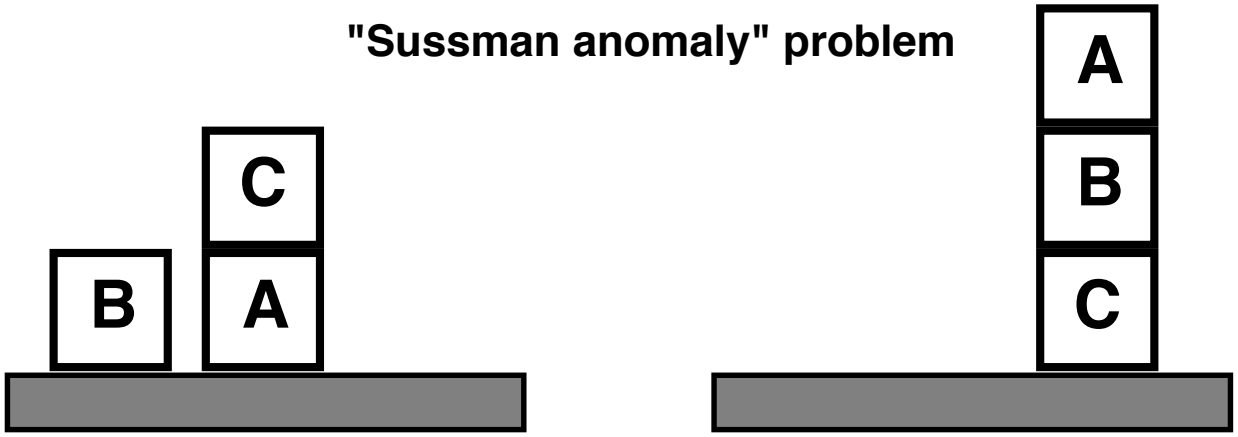
- choice of S_{add} to achieve S_{need}
- choice of demotion or promotion for clobberer
- selection of S_{need} is irrevocable

POP is sound, complete, and systematic (no repetition)

Can be made efficient with good heuristics derived from problem description

Particularly good for problems with many loosely related subgoals

Example: Blocks world



Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y)
Clear(z) On(x,y)

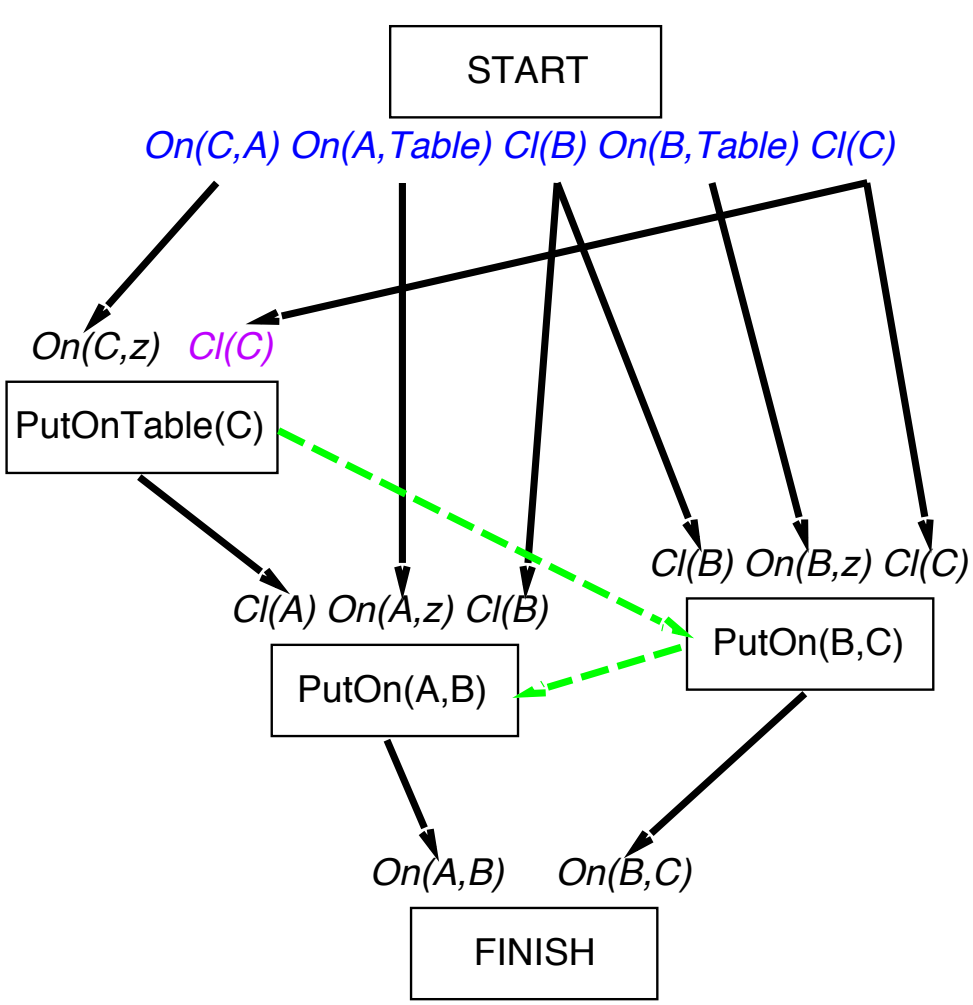
Clear(x) On(x,z)

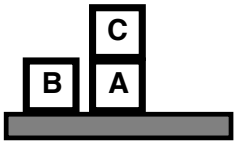
PutOnTable(x)

~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

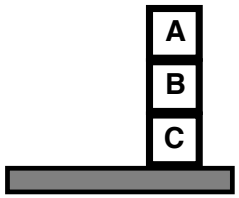
Example contd.





PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)

PutOn(B,C) clobbers Cl(C) => order after PutOnTable(C)



Other planning approaches

- SatPlan: reduction to propositional reasoning, and checking for satisfiability.
- Can check whether a plan of fixed length exists
- Hierarchical planning (planning with abstraction): first find a plan with high-level abstract actions, then refine it
- Planning with time and resources, and scheduling plans

The complexity of classical planning

PlanSAT is the question whether there exists any plan that solves a given planning problem

Bounded PlanSAT is the question whether there exists a plan of length k or less

PlanSAT is about **satisficing** (want any solution, not necessarily the cheapest or the shortest)

Bounded PlanSAT can be used to ask for the **optimal** solution

If in the PDDL language we do not allow functional symbols, both problems are decidable

Complexity of both problems is PSPACE (can be solved by a Turing machine which uses polynomial amount of space)

NP $\subseteq$ PSPACE (PSPACE is even harder than NP)

Some of the top-performing systems

International planning competition winners (from Russell and Norvig, 3rd edition):

Year	Track	Winning systems (approaches)
2008	Optimal	Gamer (symbolic bi-directional search)
2008	Satisficing	LAMA (fast forward search with FF heuristic)
2006	Optimal	SATPlan, MAXPlan (boolean satisfiability)
2006	Satisficing	SGPlan (forward search, partition into independent subproblems
2004	Optimal	SATPlan (boolean satisfiability)
2004	Satisficing	Fast Diagonally Forward (forward search with causal graph)
2002	Automated	LPG (local search, constraint satisfaction)
2002	Hand-coded	TLPLan (temporal action logic with control rules for forward searc
2000	Automated	FF (forward search)
2000	Hand-coded	TalPlanner (temporal action logic with control rules for forward sea
1998	Automated	IPP (planning graphs); HSP (forward search)

What follows for the algorithms

If a planning system based on a particular planning algorithm is very fast it does not mean necessarily that the algorithm is 'better'

Sat solvers are very fast, but one may argue that it is not very practical to propositionalise planning problems

Forward planning with good heuristics can be very fast but again may be not always practically possible

Partial order planners are considered to be very flexible and generally useful, although they don't feature in the winners table...

G53KRR

Development cycle of a knowledge-based system
Knowledge acquisition

Plan of the lecture

Development cycle of a knowledge-based system

- 1. Plan knowledge base (the content of the knowledge base, relevant inputs and outputs, strategy for testing, knowledge dictionary, concepts etc. are identified.)
- 2. Select domain experts and knowledge sources
- 3. Acquire (elicit) knowledge
- 4. Formulate and represent knowledge (knowledge is formulated in the form suitable for inference)
- 5. Implement knowledge base (knowledge is encoded in machine-readable form.)
- 6. Test knowledge base
- depending on the results: continue with knowledge acquisition or go to 7.
- 7. Systems test

Expert systems

- An expert system is a production systems which simulates behaviour of experts
- For example: MYCIN (diagnosis of bacterial diseases, 1970s), XCON (system for configuring VAX computers, 1978)
- Typical example of knowledge-based systems in the 80s

Knowledge acquisition: non-automatic methods

- Interviews with domain experts
- (Extracting knowledge from a human is often called knowledge elicitation)
- Iterative process, hard to get right first time.
- Human experts usually find it very difficult to state all the data relevant for a given problem.

Knowledge acquisition: automatic and semi-automatic methods (for expert systems)

- Programs which compile dependency networks during interviews with experts (MOLE, SALT)
- Programs using learning (META-DENDRAL)

Using dependency networks to acquire knowledge

- MOLE (Elsheman, 1988) works for systems which classify cases as instances of fixed categories, such as a fixed number of possible diagnoses. It builds an inference network similar to belief networks we will see later in the module
- SALT (Marcus and McDermott, 1989) works for open-ended sets of solutions, such as design problems; builds a dependency network and compiles into a set of production rules.

Using learning to acquire knowledge

- Learning decision diagrams from a set of prositive and negative instances of a concept (e.g. when to approve a loan application)
- Learning rules from a set of positive and negative instances META-DENDRAL (Mitchell 1978) learned how to determine structure of complex chemical compounds

Particular technique: decision tables

- A useful way of systematising knowledge preparatory to representing it using production rules
- can be compiled during interviews with experts or reading manuals or example sets

Decision tables

- A decision table has the following structure:

Conditions	Decision rule
Condition stubs	Condition entries
Action stubs	Action entries

- where condition stubs are criteria relevant for a decision, action stubs are possible actions, condition entries are Y,N and - (should be true, should be false, not relevant) and action entries are X (for take this action) or blank.
- A decision rule is represented by a vertical column of condition and action entries.

Example

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Else
cash	Y	Y	Y	N	N	
order $>$ 100	Y	N	N	-	-	
order $\geq$ 50	Y	Y	N	-	-	
order $<$ 50	N	N	Y	-	-	
credit record good	-	-	-	Y	N	
give 20% discount	X					
give 10 % discount		X				
accept order			X	X		
reject order					X	
exception report						X

Example: rules

- Rule 1: **if** cash **and** order $>$ 100 **then** give 20% discount
- Rule 2: **if** cash **and** 50 $\leq$ order $\leq$ 100 **then** give 10 % discount
- Rule 3: **if** cash **and** order $<$ 50 **then** accept order
- Rule 4: **if not** cash **and** credit record good **then** accept order
- Rule 5: **if not** cash **and not** credit record good **then** reject order
- Else generate an exception report.

Semantic web

- Aspiration: turn information available on the web into a huge knowledge base (integrated, readable and usable by machines ...)
- Formats for integration
- Languages for representing knowledge
- Ontology languages (description logics) in the following lecture

Rule ML

- Rule Markup Language (RuleML): specifying Web interchange format for rules
- Motivation comes from various aspects of Semantic Web:
  - Rules marked up for e-commerce (business rules)
  - XML transformation rules
  - Rules used for declarative specification of web services
  - Intelligent agents using rules
- XML-like specification for each ruleset: rule conditions, rule conclusions, direction (backward, forward, bidirectional).

Business rules

- A business rule is a statement that defines or constrains some aspect of the business
- Declarative, easy to modify; the idea is to separate dynamically changing rules which may apply for example only in the sales period from the application source code (for example on-line shop or rental business)
- Rules have a similar spirit to the discount example in the decision table
- Examples: IBM example of business rules: `https://www.ibm.com/support/knowledgecenter/en/SSPJVK/DigitalRecommendations/UserGuide/examplebusinessrules_nextgen.html`

Java Rule Engine API

- Java Rule Engine API (JSR-94) is a lightweight programming interface that constitutes a standard API for acquiring and using a rule engine.
- From the specification: ‘Addresses the community need to reduce the cost associated with incorporating business logic within applications and the community need to reduce the cost associated with implementing platform-level business logic tools and services.’
- **javax.rules** and **javax.rules.admin** packages.

Jess

- Jess is an expert system shell (can fill in your own rules, the engine already exists) written in Java
- Implemented using Rete algorithm (efficient incremental rule matching)
- Can be downloaded for free from `http://herzberg.ca.sandia.gov/jess` for educational use
- Rules can be specified in Jess rule language or XML; rule language is LISP-like:
  (defrule welcome-toddlers
  (person $\{$**age** $<$ **3**$\}$**)**
  $\Longrightarrow$
  (println "Hello, little one!"))
- LHS is a pattern (if a person has age less than 3 years) and RHS is an action (function call, in particular can insert new facts).