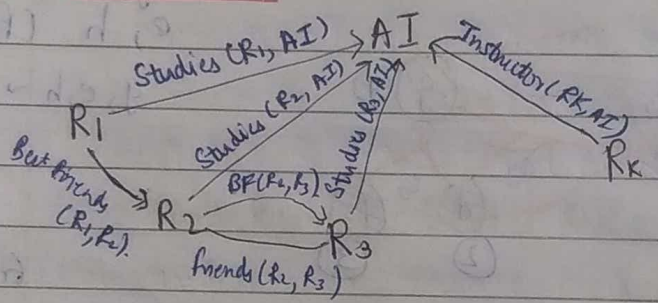
~~FORWARD CHAINING~~ ~~BACKWARD CHAINING~~
~~Data Driven.~~ ~~Goal Driven.~~

LECTURE - 8

FIRST ORDER LOGIC

What we studied in the previous lecture is called zeroth order logic.
* In First Order Logic, there are objects & relationship b/w the objects.

ONTOLOGICAL COMMITMENT



* Some Relations can be true & some can be false

eg → Friends (R_2, R_3) → T
Friends (R_1, R_3) → F

Functions ⇒ R_2 = BEST FRIND (R_1). One to One Mapping
R_3 = BEST FRIEND (R_2)

ARITY ⇒ No. of Arguments the function will take.
eg ⇒ Male (R_1) ⇒ (Arity = 1) , Friends (R_1, R_2) ⇒ (Arity = 2)

Ontological Commitment ⇒ Commitment that the different object they share to each other.

Temporal Logic : Relations change with time.

Higher Order Logic : When you define relations of the first order logic themselves as objects.

Epistemological Commitments : In this, there is a probability about the relations.

ASSUMPTIONS . 1. All relations are tuples.
eg. $(R_1, R_2) \neq (R_2, R_1)$ Order Matters.

2. We will deal with total functions. That means for every possible input there will be an output.

eg → Studies (AI, R_1)
Studies (ML, R_1).
¬Studies (Robotics, R_1).
¬Studies (AI, "ML") =?

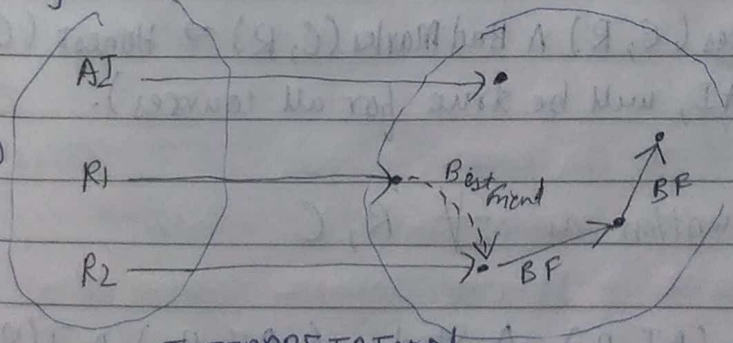
We know that ML is totally irrelevent arguments as it is not a roll no. But in AI, There is no type of object. OBJECT IS AN OBJECT. Thatswhy There is no error in Studies (AI, ML). So we explicitly assign Them false if it doesn't make any sense.

3. There may be objects which do not have a name.
eg → Studies (BestFriend (R_3), AI).
Studies (BestFriend (BestFriend (R_3)), AI).
Friends (BestFriends (R_1), Instructor (Robotics))

We didn't tell AI about BestFriend of (R_3) & Instructor (Robotics). So AI has to take care of all these issues as well.

9n Friends (BestFriend (R_3), Instructor (Robotics)).
↳ Relation (Predicate Symbol)
T ↙ ↘ F

NOTE ⇒ Objects are unknown ⇒ could be ∞.



INTERPRETATION

Terms / objects ⇒ R_1, R_2, R_3
Atomic Sentence ⇒ Smallest unit that can be true / false.
Complex Sentences ⇒ Using terms & Atomic sentences, we will make in Complex Sentences.

eg → Studies(A1, R1) ∧ BadMarks(AI, R1) ∧ student(R1) ∧ Course(AI) ⇒ Honest(AI, R1)

* If we have to write the above complex statement for all the roll numbers in one go, then we will do. →

$\forall R$, Studies(AI, R) ∧ BadMarks(AI, R) ∧ Student(R) ∧ Course(AI) ⇒ Honest(AI, R)

CONVENTION ⇒ CAPITALS → Variables. R → Variable Here
LOWER CASE → Constant.

→ So this R can be replaced by any object. eg → Bestfriend(R), AI.

∀ → for all (Quantifier). → Universal Quantifier
∃ → there exist (Quantifier). → Existential Quantifier

eg ∃R ¬Honest(AI, R) ∧ Studies(AI, R)

* NOTE → Dont underline in answer sheets. Go according to the convention.

* ∀ & ∃, under the duality are duals of each other.

¬(∀R ——) → ∃R ¬——
¬(∃Q ——) → ∀Q ——.

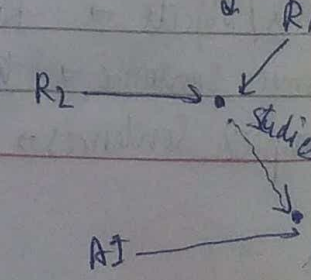
Consider the previous example, we can also write it as :-

∀R ∀C Studies(C, R) ∧ BadMarks(C, R) ⇒ Honest(C, R).

C → Courses. (True for AI, will be true for all courses).

∀R, ∀C can be written as ∀ R, C.

eg → ∃R1, R2 Studies(AI, R1) ∧ Studies(AI, R2) ∧ ¬(R1 = R2)
(Atleast 2 students will study AI).



eg → ∀C (∃R Studies(C, R))

∀R ∃C Studies(C, R))

* Logical programming is declarative.

eg⇒ $\text{Mother}(c) = m \iff \text{Female}(m) \wedge \text{Parent}(m, c)$.
$\forall c, m$

$\forall h, w$ $\text{Husband}(h, w) \iff \text{male}(h) \wedge \text{Spouse}(h, w)$.
$\forall g, c$ $\text{Grandparent}(g, c) \iff \exists p\ \text{Parent}(p, c) \wedge \text{Parent}(g, p)$.
$\forall s_1, s_2$ $\text{Siblings}(s_1, s_2) \iff \exists p\ \text{Parent}(p, s_1) \wedge \text{Parent}(p, s_2) \wedge \neg(s_1 = s_2)$.

SETS

- $\forall S\ \text{Set}(S) \iff (S = \{\}) \vee (\exists x, s_2\ \text{Set}(s_2) \wedge S = \{x \mid s_2\})$
  (Defn of set)
- $\neg\exists x, s\ \{x \mid s\} = \{\}$ (defn of empty set).
- $\forall S\ x \in S \iff \exists y, s_2\ S = \{y \mid s_2\} \wedge (x = y \vee x \in s_2)$
  (Defn of membership function)
- $\forall s_1, s_2\ s_1 \subseteq s_2 \Rightarrow \forall x\ (x \in s_1 \Rightarrow x \in s_2)$
  (Defn of subset)
- $\forall s_1, s_2\ s_1 = s_2 \iff s_1 \subseteq s_2 \wedge s_2 \subseteq s_1$ (Equality)
- $\forall x, s_1, s_2\ x \in (s_1 \cup s_2) \iff x \in s_1 \vee x \in s_2$. (Union)
- $\forall x, s_1, s_2\ x \in (s_1 \cap s_2) \iff x \in s_1 \wedge x \in s_2$ (Intersection)

eg⇒ $\neg\text{Honest}(X, AI)$ & I expect all X that satisfies this relation or output → it will return the binding list of x values / references of all x values which satisfies this cond^n.

Problems with this :-
1. There can be ∞ objects.
2. 2 or more than 2 objects can point to the same reference. (Not unique).

ASSUMPTIONS → ① Uniqueness Assumption → Every Object will be unique
② Domain Closure → No. of objects = No. of names.
③ Closed World Assumption → Ans can be true or false.

eg → ¬ Honest (R 10, AI).

In Open world, the answer to this statement is I don't know. But in closed world, the answer to this statement will be either true or false. If you cannot prove a statement honest that means it is dishonest. Vice versa is also true.

★ We can convert First Order Logic into zeroth order logic.

• UNIVERSAL QUANTIFIER (∀).

eg → α = ∀ R, C BadMarks (C, R) ∩ Studies (C, R) ⇒ Honest (C, R).

Substitution ({C = AI, R = R1}, α)

BadMarks (AI, R1) ∧ Studies (AI, R1) ⇒ Honest (AI, R1).

UNIVERSAL INSTANTIATION

In General, $\frac{\forall x, \alpha}{Subs(\{x/g\}, \alpha\})}$ g → ground rules (variable free, not ∀, ∃).

→ No. of possibilities in universal instantiation is Infinite even though no. of objects are finite.

R → R1, R2, ..., R100, BestFriend (R1), BestFriend (BestFriend (R1)) ....

• EXISTENTIAL QUANTIFIER (∃)

$\frac{\exists x, \alpha}{subs(\{x/z\}, \alpha)}$ z → constant & it should not have occured anywhere else.

eg ⇒ ∃ R ¬ Honest (A1, R).

Let R = R? → ¬ Honest (A1, R?).

So we remove ∃, but R? can be any student.

But in this also, there is problem of infinite instantiation (fofofo(x))

⇒ So what we will do is we only consider Relations / Proposition / Functions till level 1. That means we will not consider BestFriend (BestFriend (R1)) becoz it is level 2. This makes our instantiation finite.

* But if we didn't get any result by trying every object in level 1, then we will try level 2. and so on. So its like ⇒ iterative deepening.

Generalised "Modus Ponens"

$\forall R, C \quad p_1 = \text{BadMarks}(C,R) \wedge p_2 = \text{Studies}(C,R) \wedge p_3 = \text{Student}(R) \wedge p_4 = \text{Course}(C)$

$\Rightarrow q = \text{Honest}(C,R)$.

Substitution $(\theta) = \{C/AI, R/R_1\}$.

$p_1 \wedge p_2 \wedge \ldots \Rightarrow q$.

p_1 BadMarks (C,R)	p_1' = BadMarks (AI, R_1)
p_2 Student (C,R)	p_2' = Studies (AI, R_1).
p_3 Student (R)	p_3' = Student (R_1).
p_4 Course (R)	p_4' = Course (AI).
q Honest (C,R)	q' = Honest (AI, R_1).

(All the things are given in Knowledge Base except q').

BadMarks (AI, R_1), Studies (AI, R_1), Student (R_1), Course (AI).

$\{$BadMarks $(C,R) \wedge \ldots) \Rightarrow$ Honest $(C,R)\}$

Honest (AI, R_1).

$p_1', p_2', p_3', \ldots p_n', p_1 \wedge p_2 \wedge p_3 \ldots \wedge p_n \Rightarrow q)$.

Subs $(\theta, q) = q'$

(Generalised MODUS PONENS).

* This is also called ~~LIFTING~~ LIFTING of something which is of the zeroth order to the first order.

UNIFICATION

Ex Studies (AI, BestFriend (R_1)) in our KB & I have a premise ⇒ Studies $(x$, BestFriend $(y))$.

- These two are exactly equal if we apply the substitution $\{x|AI, y|R_1\}$

eg ⇒ Studies (AI, BestFriend(y)) . Studies (x, BestFriend (z))

$\{x|AI, y|R_1, z|R_1\}$

$\{x|AI, y|R_{10}, z|R_{10}\}$.

MOST GENERAL UNIFIER ⇒ $\{x|AI, y|z\}$

→ This substitution list is called as unifier because after the substitution two things unified to become one i.e. they become exactly the same.

Some failed examples are

Studies (AI, x), Studies (Robotics, y)

Studies (AI, x), Friends (AI, x)

Studies (x, R_{10}), Studies (AI, x). → Fails

Studies (x, R_{10}), Studies (AI, x_2) → Doesn't fail

$\{x|AI, x_2|R_{10}\}$

Defn of Unification ⇒ Checking whether 2 things by substitution can be made equal or not.

⇒ Unification could be used for whatever we saw in zeroth order logic to make the premise of the rule true, to check if we have the goal satisfied by some substitution.

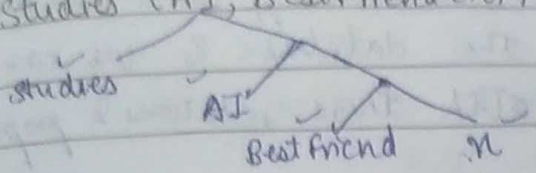
⇒ By this we can uplift anything from zeroth order logic to first order logic.

⇒ Suppose you have to unify x with best friend(x).

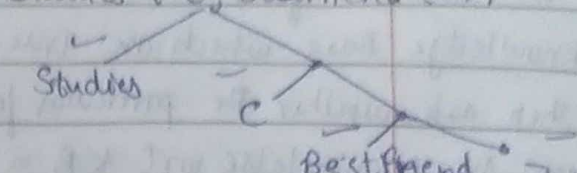
$x \neq$ bestfriend (x) → This is called Occur check.

We check if the variable that we have is also occuring in the relation or not.

eg → Studies (AI, BestFriend(x))



Studies (C, BestFriend(z))



{ ~~AI/C~~ C | AI , x/z }

PSEUDOCODE

Unify (x, y, θ (All the substitution made))

x, y : Constant, Variable, List, Compound (Relation).

~~if x = y, return θ~~

if θ = failure, return failure

if x = y, return θ // if x & y are constants

if var(x) return UNIFYVAR (x, y, θ).

if var(y) return UNIFYVAR (y, x, θ).

if list(x) & list(y).

return Unify (Rest(x), rest(y), unify (first(x), first(y), θ).

if compound(x) & compound(y).

return UNIFY (Args(x), Args(y), unify (OP(x), OP(y), θ)

return failure.

Unify (x, y, θ) x → var, y → can be anything,

if {x | oldvalue} ∈ θ then unify (y, oldvalue, θ).

if {y | z} ∈ θ then unify (y, z, θ).

if OCCUR-CHECK (x, y) return failure

return (Add x/y to θ)

eg → Friends (y, x) , Friends (BestFriend(x), y).

y | BestFriend(x) , x/y.

x | BestFriend(x) → Doesn't pass occur check.

eg →

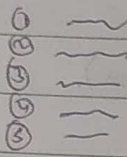
x	y	x	3
5	3	8	z
x/5	y/3	x/8	

Not possible

ZEROTH ORDER LOGIC

① We tell the facts to the ~~~~ knowledge Base which are true & then ask whether the particular fact can be true or false wrt KB

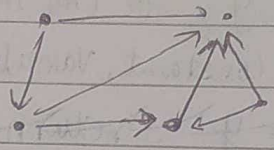
② Stored as an array of facts.



FIRST ORDER LOGIC

① We are storing the relations in the database & we can fetch those relations & propositions

② Stored as a relational databases.



FORWARD CHAINING

$\forall$ - Neglected.

We will take care about it in substitution.

* Generalised modus ponens was forward chaining. We looked at generic rule & we applied substitution, such that the rule is satisfied The premise is satisfied in the KB & we derive the consequent.

```
Forward Chaining (KB, α).
  while true
    new ← { }.
    For all rules R(p1 ∧ p2 ∧ ... pn ⇒ q).
      Standardize-var-names(R)  // Renaming of Rules
      For all θ s.t. subs(θ, p1 ∧ p2 ∧ ... ∧ pn) = subs(θ, p1' ∧ p2' ∧ ... ∧ pn')
                                  which is true in KB.
        q' = subs(θ, q).  // Generalized Modus Ponens
        if φ = UNIFY(q', α)
          return φ
      new ← new ∪ q'
    if new = { } then return false.
    add new to KB.
```

if ?(q' ~~unifies~~ unifies with some s in new or KB)

Assumption → Input is a datalog, means there are no functions ⇒ no infinite objects

* That means we have relations $P(-,-,-,-,\ldots,-)$
arity = n.

$\Theta \rightarrow$ objects.

$K\Theta^n \rightarrow$ ~~# objects~~ $\Rightarrow$ finite (K → no. of predications)

(Total no. of propositions possible)

→ exponential in terms of arity, but n will be small.

* If not datalog, then no. of objects will be infinite.

BACKWARD CHAINING

In this, unlike zeroth order, we have to return substitution list # θ

→ So suppose one branch returns θ. Now it will check for the another branches if θ is feasible or not. If its feasible, then then it will return θ to the root (source). This will be happening in parallel.

Generator.

Backward - Chaining (KB, α)

Backward Chaining - OR (KB, α, {})

Backward - Chaining - OR (KB, α, θ)

for all STANDARDIZE rules & substitution (ϕ) in $(p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)$ such that ϕ = UNIFY(q, α, θ)

if θ' = Backward - Chaining - AND ($p_1 \wedge p_2 \ldots \wedge p_n, \phi$)

yields θ'

Backward - Chaining - AND (KB, α, θ).

if θ = error, return error

if len(α) = 0 yield θ

($p_1 \wedge p_2 \wedge \ldots \wedge p_n$) — first, rest

for all θ' in Backward - Chaining - OR (KB, subs(θ, First(α)), θ)

for all θ'' in Backward - Chaining - AND (KB, rest(α), θ')

yield θ''

* STANDARDIZE means renaming the variables which are reoccuring.

* Forward chaining, making premise true., Backward chaining → Making goal true.

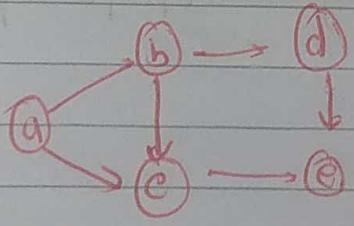
RESOLUTION In Zeroth order ⇒ l_i & m_j are Opposite.
In First Order ⇒ Unify(l_i, $\neg m_j$).
(∀) → Unification, (∃) → something new (replace).
Unlike Zeroth order, there is no assumptions.

LECTURE-9

PROLOG This language uses backward chaining. as backward chaining is goal oriented. Also backward chaining is a fancy way of saying backtracking (Memory Effluent).

ASSUMPTIONS IN PROLOG

1) Two symbols in interpretation will not point to the same object. (UNIQUE NAME ASSUMPTION)
2) ~~Infinite~~ function calling inside a function infinite time is not considered. (DATABASE SEMANTICS ASSUMPTIONS HOLDS)
3) Model is assumed to be closed world. ⇒ That means if I cannot prove any statement to be true, then it will be false.
4) No occur check. That means UNIFY(x, BestFriend(x)) will pass.



Let us consider an example of graph.

edge(a,b). //"." means I am telling my KB a fact
* Statements that ends with "." is TELL
* Statements that doesn't ends with "." is ASK

edge(a,b).
edge(b,c). These are the
edge(a,c). edges of the above graph.
edge(b,d).
edge(c,e).
edge(d,e).

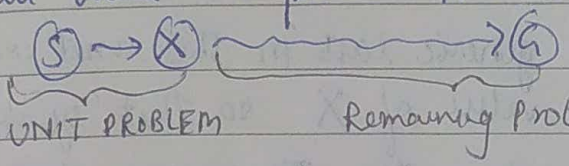
* In Prolog →
Lower Case = Symbol Upper Case = VARIABLE

* edge(a,b) → True, edge(b,c) → True, edge(b,a) → false.
* edge(a,X) → ∃X edge(a,X) → True. (X=b, X=c)
* edge(X,c) → True. (X=b, X=a).
* edge(a,X), edge(X,d) → ∃X edge(a,X) ∧ edge(X,d).
True (X=b)

* GRAPH SEARCH

path (S,G).

We will break the problem into two halves.



path(S,G) :- edge(S,G). // Base Condn.
path(S,G) :- edge(S,X), path(X,G).

This means that

edge(S,G) ∨ ∃X edge(S,X) ∧ path(X,G) ⇒ path(S,G)

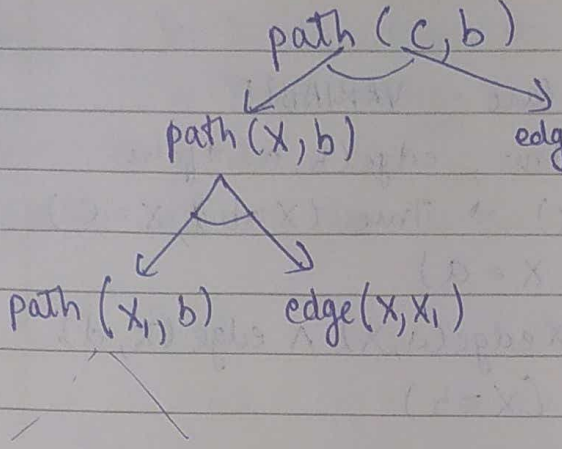
* In prolog, consequent comes first, antecedent comes last.

* path(a,c) → True, path(a,d) → true.
* path(b,a) → false.

→ we know that a∧b = b∧a

path(S,G) :- edge(S,G).
path(S,G) :- path(X,G), edge(S,X). // Reversed it.

path(a,c) → true, but path(c,b). → stack limit exceeded.



* So this is infinite, as we are checking the generic part first.

⇒ So in Prolog, keep specifics firsts & generic last in the and conditions. Because specifics highly limits the value of X so that symbol should be true which automatically reduces the search time.

* Now if we add an edge from ⑥ to ⑤, then there is a cycle. So if we check for path (b, a), it will give error (stack limit exceeded) as we know that there is no path from b to a but prolog will calculate all possible path which is ∞.

* Backtracking in a graph with cycles is ∞.
Solution is ⇒ closed / visited List.

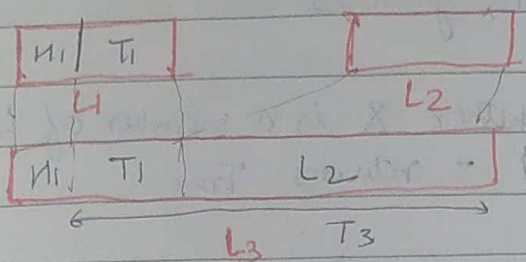
LIST IN PROLOG

* LIST: [1, 2, 3, 4, 5] example of a list.

* APPEND: [L1] ∈ [L2] ⇒ [L1 | L2] (L3)

APP (L1, L2, L3) → This means, does L1 appended to L2 gives me L3?

* In Prolog, your only output can be true / false only.



$L_1 = [H_1 | T_1]$

$L_3 = [H_1 | T_3]$

Head — Tail

APP([H1|T1], L2, [H1|T3]) :-
    APP(T1, L2, T3).
APP([], L2, L2) Base condition which is universally true (a fact).

* app([], L2, L2).
  app([H1|T1], L2, [H1|T3]) :- app(T1, L2, T3). // Code for append.

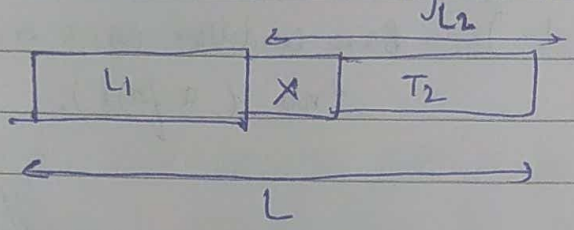
* app([], [1,2,3], L) L = [1,2,3]
* app([6,7], [1,2,3], L) L = [6,7,1,2,3]
* app([1,2], [3,4,5], [1,2,3,4,5]) returns true.
* app(L1, [3,4,5], [1,2,3,4,5]) L1 = [1,2]
* app([1,2], L, [1,2,3,4,5]) L = [3,4,5]
* app(L1, L2, [1,2,3,4,5])
  1. L1 = [], L2 = [1,2,3,4,5]
  2. L1 = [1], L2 = [2,3,4,5]
  3. ⋮
  6. L1 = [1,2,3,4,5], L2 = []
* app(B, [3|A], [1,2,3,4,5])
  B = [1,2], A = [4,5]
* app(B, [3|_], [1,2,3,4,5]) # If you do the underscore, then it will not come as output.
  B = [1,2]
* app(X, [3, AFTER|Y], [1,2,3,4,5])
  X = [1,2], AFTER = 4, Y = [5].

* app(X, [Before, 3, After|Y], [1,2,3,4,5]).
X=[1], Before 2, After = 4, Y=[5].

* MEMBER: This function tells whether X is a member of list L or not.
eg: member(4, [1,2,3,4,5]) - returns True

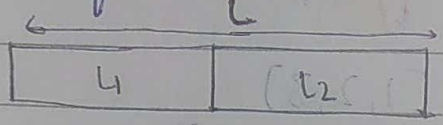
member(X, L) :- app(_, [X|_], L). // code.

• _ ⇒ It also means anonymous names.



mem(X,L) :- app(L1, [X, T2], L)
(But this code gives singleton error)

* PREFIX: This function tells whether L1 is the prefix of L or not.

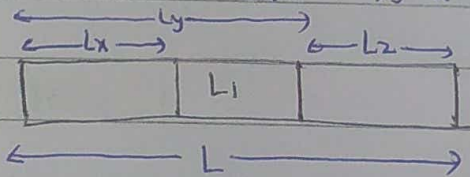


prefix(L1, L) :- app(L1, _, L).
// code

* Suffix: This function tells whether L1 is the suffix of L or not.

suffix(L1, L) :- app(_, L1, L). // code

* SUBLIST: Tells whether a list L1 is a sublist of list L. or not.



sublist(L1, L) :- app(_, L1, Ly), app(Ly, _, L).

* But this code for some cases, gives memory limit exceeded bcoz Ly is not ~~bounded~~ bounded. So if first cond^n is false, then that means it will search for all possible comb^n. (AND-OR Graph will go infinite)

sublist (L1, L) :- app(Ly, _, L), app(_, L1, Ly).

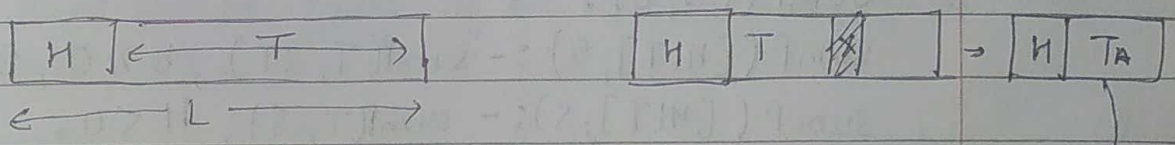
→ By doing thing we are bounding Ly as L is already a bounded list. So by app(Ly, _, L), it becomes bounded. So now in app(_, L1, Ly), Ly acts as a bounded list. So now there will be no memory limit exceeded.

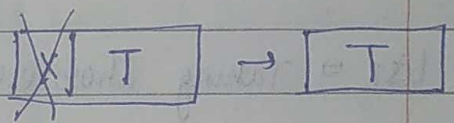
* Recursive code of member function:

member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

* adding is easy. For eg → 5|L means adding 5 to list.

* DELETE





① delete(X, [X|T], T).
② delete(X, [H|T], [H|TA]) :-
③ delete(X, T, TA).

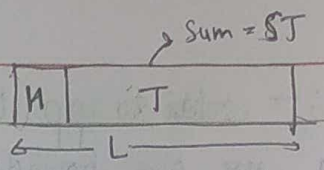
① Deleting X from a list with Head X & tail T will give me a list T.
② Deleting X from a list with Head H & tail T will give me a list with Head H & tail TA.
③ Deleting X from a list T will give me list → TA.

* If there are multiple occurence of a number, then ~~first one will get deleted~~. it will show all possible list if we are deleting that number.
* But it doesn't delete all occurences in one go.

eg → delete(2, [3, 2, 4, 2, 5, 2, 6, 2, 7], L).

L = [3, 4, 2, 5, 2, 6, 2, 7]
L = [3, 2, 4, 5, 2, 6, 2, 7]
L = [3, 2, 4, 2, 5, 6, 2, 7]
L = [3, 2, 4, 2, 5, 2, 6, 7]

* SUM OF ALL ELEMENTS.




```
sum([],0).   // Base Cond^n
sum([H|T],S):- sum(T,ST), S is ST+H.
```

→ "is" is a numerical equality in prolog.
→ "=" checks whether the objects are pointing to same reference or not.

→ Prolog can't do arbitrary addition. That means if sum is given then it will not give you the list whose sum is equal to given sum.

* SUM OF POSITIVE NUMBERS IN THE LIST

```
sumP([],0).
sumP([H|T],S):- sumP(T,ST), H>=0, S is ST+H.
sumP([H|T],S):- sumP(T,S), H<0.
```

* POSITIVE LIST ⇒ Taking those elements which are true.

```
posL([],[]).
posL([H|T],[H|T1]):- H>=0, posL(T,T1).
posL([H|T],L1):- H<0, posL(T,L1).
```

SETS IN PROLOG

* Set means if a element IS NOT present in the set, then we have to add that element to the set. Otherwise, do nothing.

* NOT

```
not(P) : P; fail.
otherwise, true.
```

CUT - OPERATOR (!)

* if Premise 1 then Consequent 1.
  if premise 2 then Consequent 2

} we use this previously

* if Premise 1 then consequent 1
  else if Premise 2 then consequent 2
  otherwise coresequent n.

⇒ The first case is an ~~and~~ AND - OR graph. in which ~~and~~ more than one premise can be true simultaneously.

⇒ The second case in an AND - OR graph. in which ~~only~~ if one premise is true then other will be false. If all the premise are false then otherwise will work.

* 2nd case is a heuristic that prolog could use. It is also a specification that ~~a pro~~ we can use. This thing we will do for the exclusivity of OR - NODES.

⇒ During executions time, this 2nd and-or is also ~~as~~ useful. ~~as~~ If one premise is true, then other premise will not be tried at all..

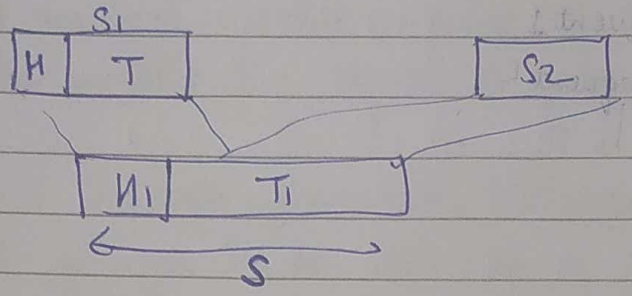
* adding to set. ⇒ $adds(X, S, A) = \begin{cases} S & \text{if } X \in S. \\ [X|S] & \text{if } X \notin S \text{ / otherwise.} \end{cases}$

```
addSet(X, S, S) :- member(X, S), !.
addSet(X, S, [X|S]).
```

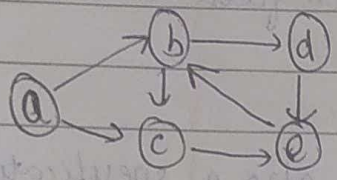
⇒ If member(X, S) fails then "!" will ensure that other fact will work.

* APPEND UNIQUE / UNION.

union([], S2, S2).
union([H|T], S2, S) :- member(H, S2), !, union(T, S2, S).
union([H,T], S2, [H,T1]) :- ~~member~~ union(T, S2, T1).



* Now coming back to our graph search. Now we will add visited to our graph.



⇒ path(S, G, V) ⇒ Is There ~~~~ a path from S to G where V is the list of nodes that are already visited.

not(P) :- P, !, fail ; true.
path(S, G, V) :- edge(S, G).
path(S, G, V) :- edge(S, X), not(member(X, V)), path(X, G, [X|V]).
(X is not equal to G) X/=G
!visited(X).
[X added to visited]

→ X is not equal to G because we are getting answer for X=G from above fact. X/=G prunes a little bit.

⇒ In normal case, when the node is removed from the frontier, then we make it visited. But ~~in~~ here, we are ~~not~~ making it visited as soon as ~~not~~ it enters in the frontier.

* pathPrint(S, G, V, P) ⇒ Is there a path from S to G where V is the list of nodes that are already visited. If yes then store the path in P.

ⓐ → ⓑ → ⓒ → ⓓ → ⓔ

path (a, e, {a}, P)

x = b

edge (a, b) — path (b, e, {a, b}, P)

edge (b, c) — path (c, e, {a, b, c}, P)

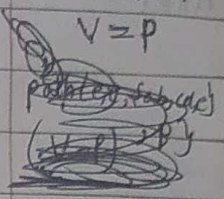
edge (c, d) — path (d, e, {a, b, c, d}, P)

edge (d, e)

P = V (At last step)

path (S, G, V, [G|V]) :- edge (S, G).

path (S, G, V, P) :- edge (S, X), X /= G, not (member (X, V)),
path (X, G, [X|V], P)



→ path (a, e, {a}, P)

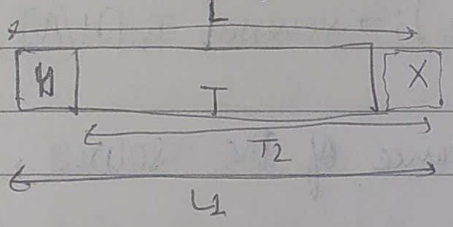
P = [e, c, b, a], [e, d, b, a], [e, c, a].

As we can notice, the answer is coming in reverse direction.

→ We can solve this problem by adding the elements to the end.

addAtEnd (X, [ ], [X]).

addAtEnd (X, [H|T], [H|T2]) :- addAtEnd (X, T, T2).



[H|T] = L

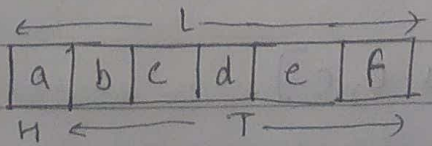
[H|T2] = L1

So the correct code is :-

path (S, G, V, P) :- edge (S, G), addAtEnd (G, V, P).
path (S, G, V, P) :- edge (S, X), X /= G,
not (member (X, V)),
addAtEnd (X, V, V2),
path (X, G, V2, P).

* REVERSE ⇒ It reverses the list L & store that reversed list into L1.

rev (L, L1)



① rev (T, T1) = b, e, d, c, b.

② addAtEnd = (H, T1, L1).

```
reverse([], []).
reverse([H|T], L1) :- reverse(T, T1), addAtEnd(H, T1, L1).
```

* Now we can use this reverse function in the path program.

Aliter ⇒

reverse([a,b,c,d], [], L2)

Head

a reverse([b,c,d], [a], L2)

b reverse([c,d], [b,a], L2)

c reverse([d], [c,b,a], L2)

d reverse([], [d,c,b,a], L2) L2 = [d,c,b,a]

```
reverse([], A, A).
reverse([H|T], A, L2) :- reverse(T, [H|A], L2).
```

* Now we want to return the distance of the source, to goal.

```
path(S, G, V, [G|V], 1) :- edge(S, G).
path(S, G, V, P, D) :- edge(S, X), X /= G,
                       not(member(X, V)),
                       path(X, G, [X|V], P, D1),
                       D is D+1.
```

⇒ path(a, e, [a], P2, 2), reverse(P2, P)

As we are giving D = 2, so P will be the path from a to e of distance 2.

P = [a, c, e].

* Iterative Deepening:

iterdeepening (S, G, L, P) :- path (S, G, [S], P2, L), reverse (P, P2).

L1 is L + 1, L1 < 25, iterdeepening (S, G, L1, P).

* Now we will make the graph with ~~state~~ node as state.

transition (up, cell (I, J), cell (I1, J)) :- I1 is I - 1, I1 >= 0.
(State)

transition (down, cell (I, J), cell (I1, J)) :- I1 is I + 1, I1 < 25.
transition (left, cell (I, J), cell (I, J1)) :- J1 is J - 1, J1 >= 0.
transition (right, cell (I, J), cell (I, J1)) :- J1 is J + 1, J1 < 25.

↳ Can be useful for grid problem.

⇒ transition (A, cell (3,3), X).
A = up, X = cell (2,3)
A = down, X = cell (4,3)
A = left, X = cell (3,2).
A = right, X = cell (3,4).

* Transition function here becomes the edge function in normal graph.

LECTURE - 10

CLASSIC PLANNING * So far we have studied,

• Search: Source ~~~~> Goal.
Problem here is you have to write the code from scratch for every search problem & for every search problem, the states & state space is different.

- LOGIC : Definition of functions where everything is defined as propositional logic true or false.

* Now we will combine the two things, That is search & logic. Also we will see what happens when my agent acts in a physical world & changes thing around.
* Also, my agent is now not asking questions. It needs a sequence of action or a plan which when executed sequentially step by set will lead all the way up to from the source to the goal.

* Now my states ~~are~~ & actions will be things which are true or false (following propositional logic). The state will be symbols which could be true/false.

* Classical planning is better than normal search because the heuristics are specific for a specific problem. But in classical planning, the heuristics will be generic as it will work only on propositional logic. In Normal search, we are adding a constraint that is usually bad. My variables could be anything (eg. 1, 2, 5, A, B, C). Now I am putting the constraints that it has to be a propositional logical variable (can only take True or false or unknown), This thing will help us in writing excellent heuristic functions.

* Logic, doesn't have the time factor attached to it. There is a generalization of logic where you could have the time factor attached to it. However the problem is by adding time constraint the no. of state got multiplied. So not all may be useful. Also, we can define some propositions such that we can interact b/w every states & every action possible. My logic has nothing which my classic search had which was, find out which actions are feasible & only apply those feasible actions to

a predecessor to generate the successor. Therefore they are representational problem.

substitution.

- Travel (PRG, NDLS, RK) ← Travel (from, to, Person)
  - ↑ Action.
  - ↑ Action Schema

- Action schema is a general template in which you could supply any from city ; any to city & any person & action will be executed.

* But here total no. of actions possible are very high.
* So in classic search, actions are replaced by action schema.

↳ But out of all actions, some actions are infeasible.

* Travel (from, to, p). (ACTION SCHEMA)

Pre-Condition : At (p, from), The person p needs to be at from.

Domain Part { Place (from), From needs to be a valid place
Place (to), To needs to be a valid place
Person (p) p needs to be a person.

if applicable

⇒ If Precondition doesn't hold good Then action is not feasible.
⇒ If Precondition is valid Then we called action Applicable.

Effect : At (p, to) ∧ ¬ At (p, from).
(p will be no longer in from).

* States are sets of logical variables.
eg⇒ At (RK, PRG), ¬At (RK, DEL), Teaching (AI, RK), Speaking (RK).

* In logical programming, the no. of true variables are far less than no. of false var.

eg⇒ ~~#S~~

ASSUMPTION : 1) Closed World.

2) We restrict ourself to ground, Qualifier ($\exists, \forall$) free variables

eg → $\forall s_1, s_2$ At(p, d) → ¬At(p, s_2)

(first order logics have complications)

* As no. of true statements are significantly less than no. of false statements, So in my state I only write the variables which are true.

* In prolog we only define those facts which are positive. We doesn't consider the negatives.

At(RK, PRG) —Travel(from, to, p)— subs(from/Prg, to/Del, p/RK) → At(RK, DEL) —Removing False statements→ At(RK, DEL) ¬At(RK, PRG)

(All preconditions satisfied)
(Action is applicable)

* State Vector ⇒ It can array / variable size array / linked list of all the propositions which are known to be true. (With time it can change)

→ Anything which is true at the precondition should be there in the state vector.

* In the closed world assumption, we differentiate effects into +ve effects & -ve effects.

* If something is true, then it is added to the state vector. If something is false Then it is deleted from the state vector.

Travel(from, to, p)

PRE : At(p, from) ∧ Place(from) ∧ place(to) ∧ person(p)

Effects { ADD List : At(p, to).
Delete List : ¬At(p, from)

$S \xrightarrow{a} S'$

→ if a is applicable on S, $S' = (S - \text{DeleteList}(a)) \cup \text{AddList}(a)$.

3) All the assumption related to database semantics holds.
(Every object has unique name).

* In our precondition, we ~~should~~ can add from ≠ to.

AIR CARGO PROBLEM

Init (At (C1, SFO) ∧ At (C2, JFK) ∧ At (P1, SFO) ∧ At (P2, JFK)
∧ Cargo (C1) ∧ Cargo (C2) ∧ Plane (P1) ∧ Plane (P2) ∧ Airport (JFK) ∧ Airport (SFO))
Goal (At (C1, JFK) ∧ At (C2, SFO)).
Action (Load (c, p, a),
PRECOND : At (c, a) ∧ At (p, a) ∧ Cargo (c) ∧ Plane (p) ∧ Airport (a)
EFFECT : ¬At (c, a) ∧ In (c, p)) //(Cargo is inside the plane & cargo is at no man's land)
Action (UnLoad (c, p, a),
PRECOND : In (c, p) ∧ At (p, a) ∧ Cargo (c) ∧ Plane (p) ∧ Airport (a).
EFFECT : At (c, a) ∧ ¬In (c, p)). // (Cargo is outside the plane & cargo is at airport)
Action (Fly (p, from, to),
PRECOND : At (p, from) ∧ Plane (p) ∧ Airport (from) ∧ Airport (to)
EFFECT : ¬At (p, from) ∧ At (p, to)).

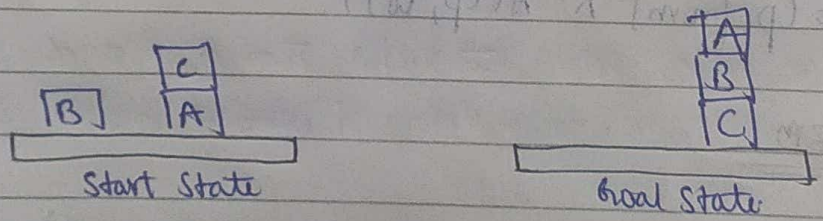
FLAT TIRE PROBLEM

Init (Tire (Flat) ∧ Tire (Spare) ∧ At (Flat, Axle) ∧ At (Spare, Trunk)).
Goal (At (Spare, Axle)).
Action (Remove (obj, loc),
PRECOND : At (obj, loc)
EFFECT : ¬At (obj, loc) ∧ At (obj, Ground))

Action (PutOn (t, Axle),
PRECOND: Tire(t) ∧ At(t, Ground) ∧ ¬At(flat, axle) ∧ ¬At(Spare, Axle)
EFFECT: ¬At(t, Ground) ∧ At(t, Axle)).
Action (Leave Overnight, // This is stealing.
PRECOND:
EFFECT: ¬At(Spare, Ground) ∧ ¬At(Spare, Axle) ∧ ¬At(Spare, Trunk)
∧ ¬At(flat, Ground) ∧ ¬At(Flat, Axle) ∧ ¬At(flat, Trunk)).

BLOCK WORLD PROBLEM Assumption: Table has Infinite Capacity.

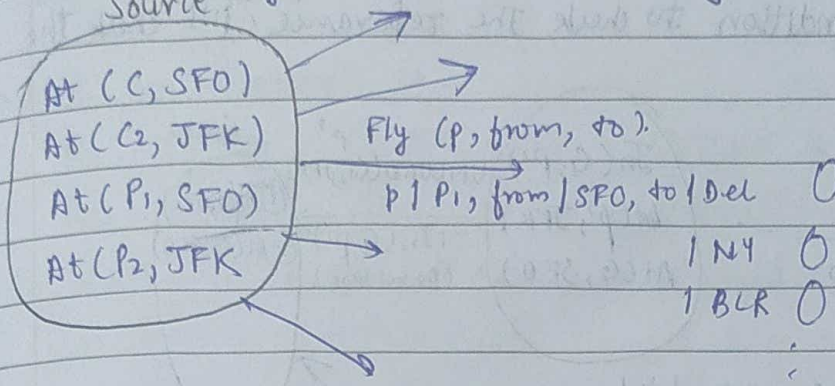
Init (On(A, table) ∧ On(B, Table) ∧ On(C, A) ∧ Block(A) ∧ Block(B)
∧ Block(C) ∧ Clear(B) ∧ Clear(C) ∧ Clear(Table)). // Clear Means you can put something on top.
Goal (On(A, B) ∧ On(B, C)).
Action (Move(b, x, y),
PRECOND: On(b, x) ∧ clear(b) ∧ clear(y) ∧ Block(b) ∧ Block(y) ∧
(b ≠ x) ∧ (b ≠ y) ∧ (x ≠ y).
EFFECT: On(b, y) ∧ Clear(x) ∧ ¬On(b, x) ∧ ¬Clear(y)).
Action (MoveToTable(b, x),
PRECOND: On(b, x) ∧ Clear(b) ∧ Block(b) ∧ Block(x),
EFFECT: On(b, Table) ∧ Clear(x) ∧ ¬On(b, x)).



* By using redundant variables, the first Order logic can be converted into second order logic. Clear() is redundant in this example.

∄4 ¢ On(B, 4) means Clear(B).

* Now lets try to solve Air Cargo Problem.



~~Complexity~~ Generating children of source is very expensive, no matter what algo you are using (BFS, DFS, Dijkstra, UCS).

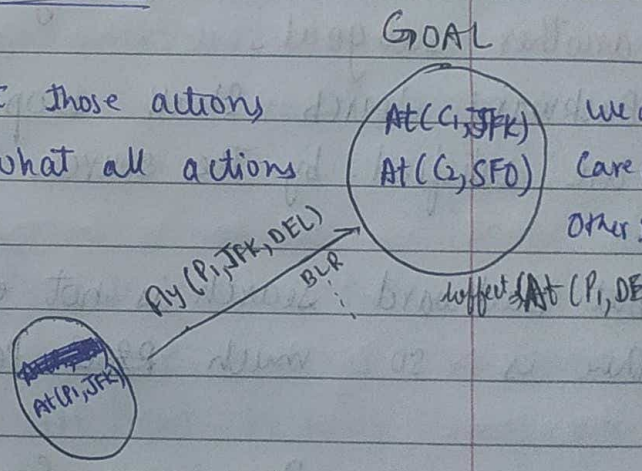
computationally too Expensive.

S ← extract from Queue
for all applicable action a considering all action schema & substitution
S' ← generate children.

⇒ So forward search atleast as of now is not making any sense.

⇒ So we will try to do BACKWARD SEARCH.

⇒ In forward search, we were looking at those actions which were applicable. Now we will see what all actions can be applied.

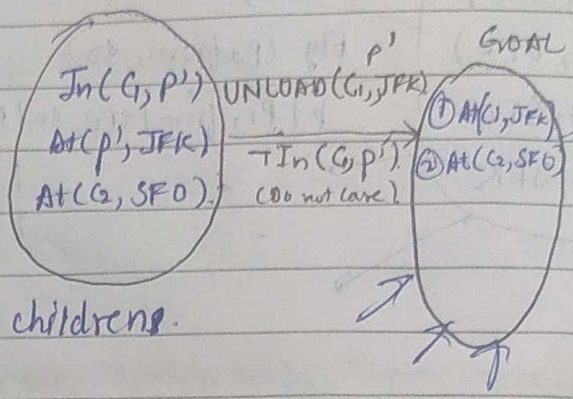


⇒ In this case the branching factor is still infinite. So Backward search didn't solve the problem.

⇒ So we will apply one constraint ⇒ Only select actions which are relevant (they obtain at least 1 sub goal).

Subgoals ⇒ clauses of your goal.

* Backwards searches into my priority queue & I will apply all the actions instead of checking the precondition to check the relevance, I'll check the goal.



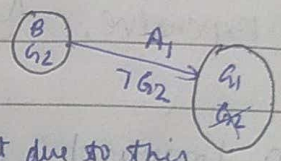
* P' is unsubstituted value of P.
If in further search we get particular value, then we can fill it up. Otherwise I'll get too many childrens.

* As goal don't care about ¬In(G1,P'), thats why we aren't adding it up.
* I didn't care about other subgoals being produced which are do not care.

* Consider: Goal : $G_1 \wedge G_2$

Actions:

A_1, Pre: B, Effects : $G_1 \wedge \neg G_2$



(But due to this G_2 gets deleted.)

* So In Backward search, only select those actions which do not negate another subgoal.
* Backward search shall stop if (By some substitution) all the variables are satisfied by the source.
* But Backward search is not efficient, due to unsubstituted variables, there is so much Permutation & combinations possible.

HEURISTICS IN BACKWARD SEARCH

Let h(S) ⇒ This function tries to find the distance from S to Goal.

RELAX CONSTRAINTS

Consider an Example ⇒

A_1: Pre : $X \wedge Y \wedge \neg Z$, EFFECTS : $A \wedge P \wedge Q$

A_2: Pre: $\neg X \wedge Z \wedge P$, EFFECTS: $B \wedge R \wedge S$

A_3: Pre: $\neg X \wedge Y \wedge Z$, EFFECTS: $C \wedge T$

Source : $F \wedge G \wedge X$

Goal : $A \wedge B \wedge C$

conditions in Relax constraints are :-

(1) No precondition.

Heuristic : # subgoals that are not satisfied.

(2) Sub Goal Independence ⇒ There is no ~~subgoal~~ actions that attains two subgoals Directly.

(3) No Interaction of Negative Goals.

eg⇒ any effects ~~does not have the form~~ will not be in the form : $A \wedge P \wedge Q \wedge \neg C$

⇒ Now, let us consider that we are not taking Sub goal Independence to be in assumption.

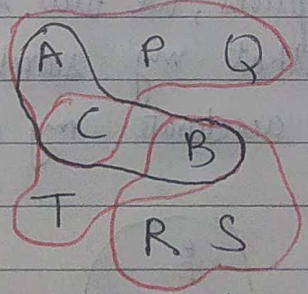
A_1: PreCon: , Effects: $A \wedge P \wedge Q \wedge C$

A_2: Pre: , Effects: $B \wedge R \wedge S$.

A_3: Pre: , Effects: $C \wedge T \wedge B$.

GOAL: $A \wedge B \wedge C$ ($h(s) \Rightarrow 2$).

SET COVER PROBLEM



By Set cover Problem, we can get $h(s) = 2$. But Set Cover Problem in NP Hard. By using Greedy we can solve this in polynomial time but there is an assumption that Load, Unload, Fly have the same cost. Also This compromises optimality for getting good heuristics.

⇒ This is called Oldest Version/Version 1 of PDDL (Planning Domain Definition Language).

* Set cover Problem Returns very small value of heuristics. Due to this it saves very less time.

* Another relaxation that we can do is to remove the -ve effects. If we remove the -ve effects then problem is solvable by a class of local algorithm in which from source we selects random actions to reach the goal. Every action can lead us to the goal.

* Another heuristic is state abstraction. For eg - P' which was unsubstituted. P is not in the goal.

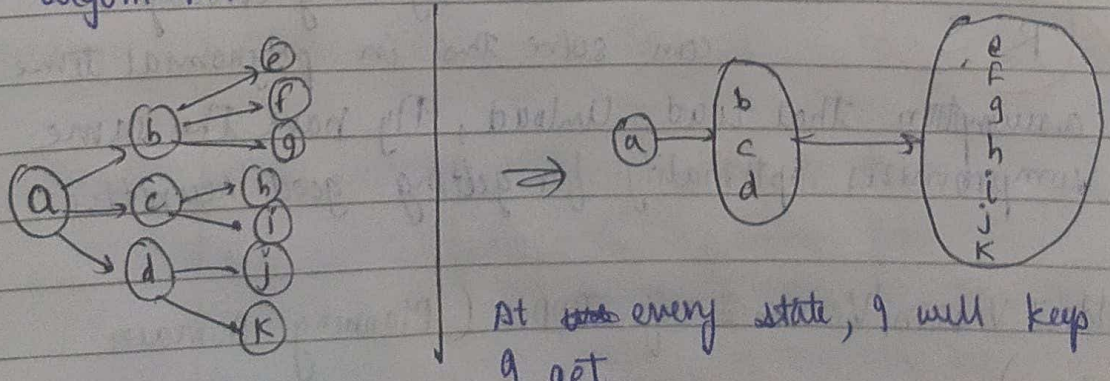
* Another heuristic is going with Subgoal Independence. Attaining a subgoal is a very small problem.

Small Problem (G1) ∧ G2 ∧ . . . ∧ Gn = Goal (Large Problem)

c1 + c2 + . . . cn . (Total Cost)

LECTURE - 11

* But after applying heuristic in Backward search, the complexity is still exponential.
* The heuristics estimates by relaxation are way too simpler than the real world costs are. This increases the heuristic error & the search will go extremely slow.
* So what we will do is by some assumptions, we will try to convert the tree (search) into a linked list so that my search goes linearly with time. By this I will be able to accelerate my actual search algorithm.



At every state, I will keep clubbing in whatever I get.

PLANNING GRAPH

HAVE YOUR CAKE & EAT IT TOO

INIT : HAVE (CAKE), ¬ EATEN (CAKE).

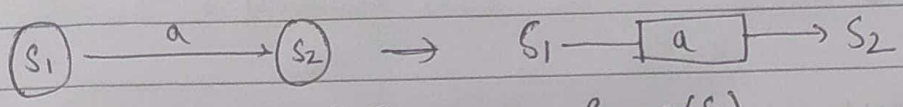
GOAL: HAVE (CAKE) ∧ EATEN (CAKE).

ACTIONS :

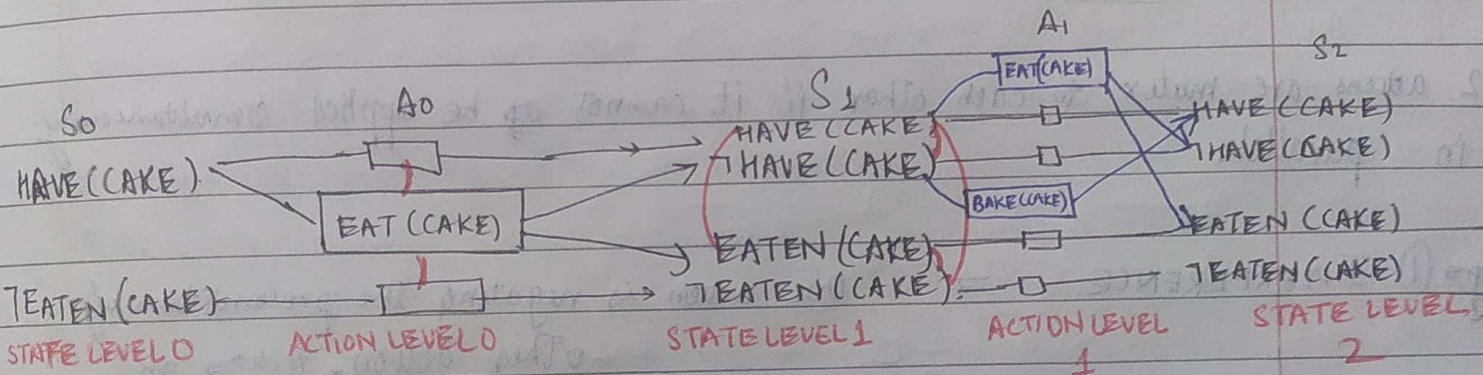
① EAT (CAKE), PRE : Have (Cake), EFFECT : ¬HAVE (CAKE) ∧ EATEN (CAKE)

② BAKE (CAKE), PRE : ¬HAVE (CAKE), EFFECT : HAVE (CAKE).

Note: Change of Convention



both means $S_2 = a(S_1)$.



* PERSISTENE ACTIONS (No operation).

PRE - COND = EFFECT = NAME [ ]

eg→ HAVE (CAKE), PRE: HAVE (CAKE), EFFECT: HAVE (CAKE).

⇒ Having Have (Cake), ¬ Have (Cake) in same state is a part of our assumption.

⇒ So we can say that if I have got N ground propositional variables, each one of them can take either true or false, I have got a maximum of 2*n propositional variables, n entries as TRUE & n entries as FALSE.

So my any state level will have linear no. of propositional variables.

* The assumption that Have & ¬Have in the same state is done by introducing mutex.

MUTEX RELATIONSHIP ⇒ Mutex relationship between two propositional literals means that both cannot occur simultaneously at the same time.

Represented by Have(Cake)) ¬Have(Cake)

* Now we will add new edges in our planning graph, which will denote two things that cannot occur simultaneously. It could be between 2 actions & 2 states.

MUTEX B/W ACTIONS

2 actions are mutex to each other if it cannot be applied simultaneously in parallel.

① INTERFERENCE ⇒ One action is negating the precondition of other action. Actions are interfering with each other.

In Front System → [Watching AI] A2

A1 [Washroom Go] → ¬ In Front System

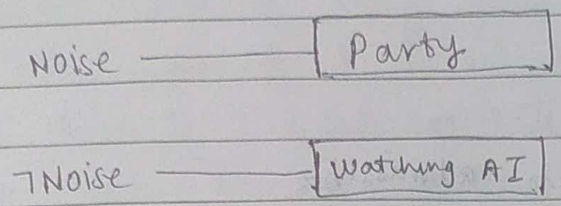
Effects of A1 negates the pre-condition of A2, Then A1 & A2 are mutex to each other.

② INCONSISTENT EFFECT ⇒ When effect of one action is exactly opposite to other action.

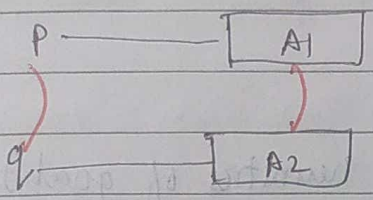
[SLEEP] → ¬ WORKING BRAIN

[Watching AI] → WORKING BRAIN

③ COMPETING NEEDS. ⇒ When two actions have opposite preconditions.



④ INCONSISTENT SUPPORT ⇒ When two actions have mutex preconditions, then they are mutex.



MUTEX B/W ~~STATES~~ LITERALS

Two Propositional Literals have mutex relationship to each other if all the ways of producing them simultaneously are mutex to each other.

or

p & q are ~~literals~~ mutex if all comb^n of Actions that can simultaneously produce them are mutex to each other.

eg⇒ $p \rightarrow A_1, p, A_2$
$q \rightarrow A_3, A_4, q$

If $A_1 \frown A_3$, $A_1 \frown A_4$, $A_1 \frown q$, $p \frown A_3$, $p \frown A_4$, $p \frown q$, $A_2 \frown A_3$, $A_2 \frown A_4$, $A_2 \frown q$ then ⇒ p & q are mutex.

* My Planning Graph levels off. That means we have generated all the literals that could be generated. which means the actions which were applicable at time a_{n-1} will be applicable at time a_n. All the literals, mutexes, actions will remain the same. Once this happen, we stop.

* Before levelling off, state mutexes decreases & # actions increases.

HEURISTICS FOR A PLANNING GRAPH

1) MAX - LEVEL HEURISTIC' → This is admissible

GOAL: $a \wedge \neg b \wedge c \wedge \neg d \wedge \neg e$.

LEVEL: MAX (3, 2, 2, 0, 1) = 3

2) SUM - LEVEL HEURISTIC

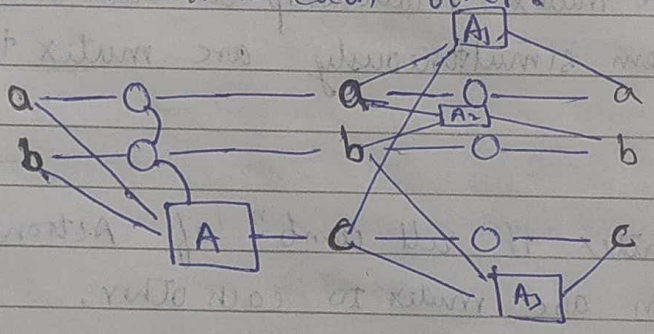
Assume sub-goal Independence.

Considering Previous Example

$3 + 2 + 2 + 0 + 1 = 8$. (Sum level heuristics of goal).

3) SET - LEVEL HEURISTIC.

At what level do 9 first get all subgoals such that no two sub goals are mutex to each other.



⇒ But They are not good heuristics. Sum Level Heuristics is not even admissible & also there are no negative interactions modeled, mutex is extremely Rudimentary. Even though its linear it not that good Heuristic.

⇒ Now 9 will expand my aim further. 9 will try to extract a solution from this planning graph. Its somewhere hidden.

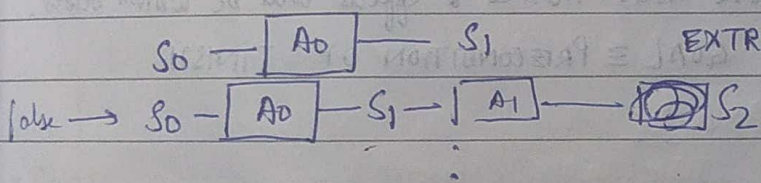
* So now, my problem is to extract a solution. I need to select one action out of every two actions that have a mutex to each other. And if I select an action, the precondition of it has to be true. If I select an action, then its effects also needs to be true. So this sounds like a constrained satisfaction problem.

* Try to Extract a solution is a CSP where mutexes, precondition being applicable & effects are constraints.

* By CSP, it is possible that we got solution much deeper. So it sounds like iterative deppening. (In normal case, it got levelled off).

* In our planning graph, everything needs to be coloured as black & white. Black for mutex showing & white for which actions to choose

GRAPH PLAN ALGORITHM.



* By set level heuristics, we know that there is no solution upto some level. So I could first expand the graph till that level, Then I can do above steps.

* Each extract solution is a CSP.

* If set level heuristic is ∞, means till levelling off, atleast one pair of subgoals are mutex. → No Soln.

* CSP also gives us the least no. of subgoals that is unreachable if there is no solution.

NO GOODS

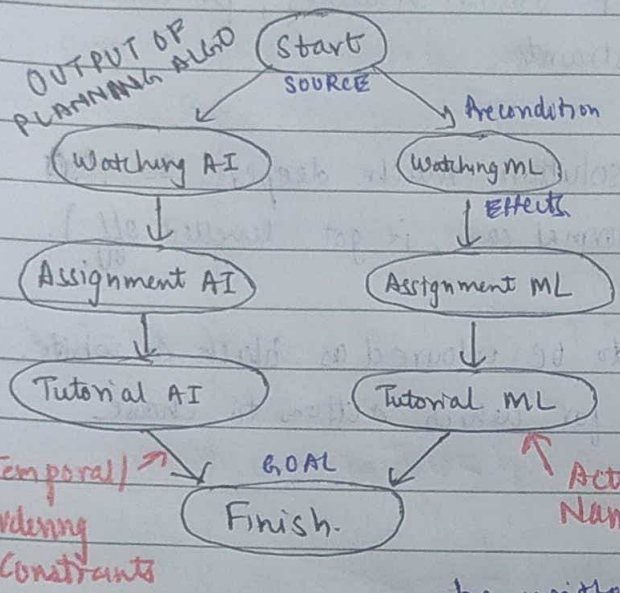
* Time complexity of Planning Graph.

$$CSP + (n+a)^2 \times l_{max} \Rightarrow \text{Polynomial.}$$

a → list of actions.
n → no. of literals.

If two relations have a mutex to each other or not

max level that we can go.



PARTIAL ORDER PLANNING

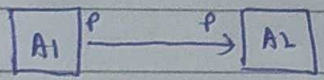
Linearizing this graph is topological sort:

S → AIV → MLV → ASS AI → TUT AI → ASS ML → TUT ML → Finish.

* If you need to write down the classic states as in the propositional literals, then the preconditions would be written above Action names & effects will be written below.

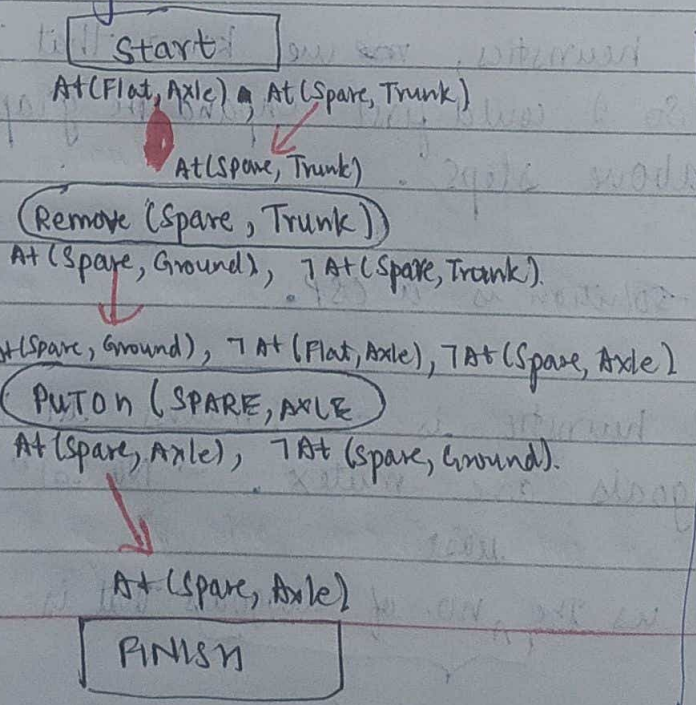
* SOURCE ≡ EFFECT OF START ; GOAL ≡ PRECONDITION OF FINISH.

* Consider Flat-Tire Problem Again.



A1 Achieves p for A2

$A_1 \xrightarrow{p} A_2$



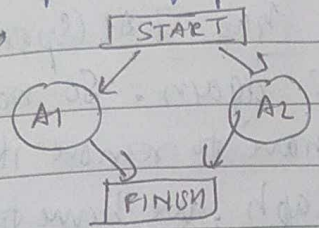
We say that at goal, there are conditions which are unmet & 9 will satisfy this unmet conditions. This is more generally called as a ~~flaw~~ flaw.

* We did ~~backward~~ forward search here.

* The graph generated by forward search is one of the states. There are some conditions which remain unmet. But we have drawn only one part of order planning graph. (¬At(Flat, axle) can be satisfied by applying Leave overnight).

* This partial graph ultimately has to be linearized.

eg →

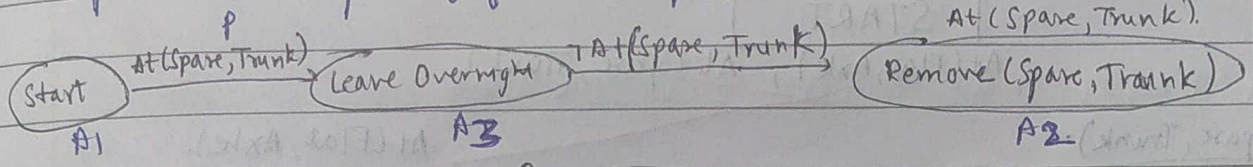


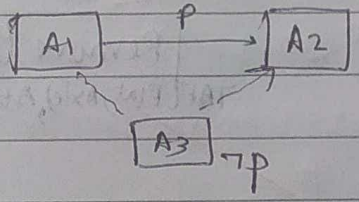
After Linearizing, we get

1) Start → A1 → A2 → Finish

2) Start → A2 → A1 → Finish.

Consider previous example of flat tire problem.





Here A_3 undos the p condition p for A_2. (As effect of A_3 = ¬p & precondition of A_2 = p)

$A_1 \xrightarrow{p} A_2$, A_3 has effect ¬p ⇒ A_3 is a Threat to $A_1 \xrightarrow{p} A_2$

* That means linearization. $A_1 \rightarrow A_3 \rightarrow A_2$ is not possible.

PROMOTION

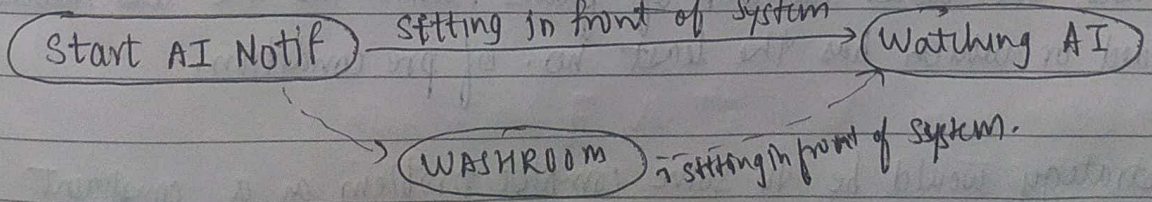
Just We have to ensure that A_3 should not be in the middle.

DEMOTION

$A_1 \xrightarrow{p} A_2 \prec A_3$

$A_3 \prec A_1 \xrightarrow{p} A_2$.

* Another Example of Threat can be



Promotion ⇒ Start AI Notif → Watching AI → Washrooms

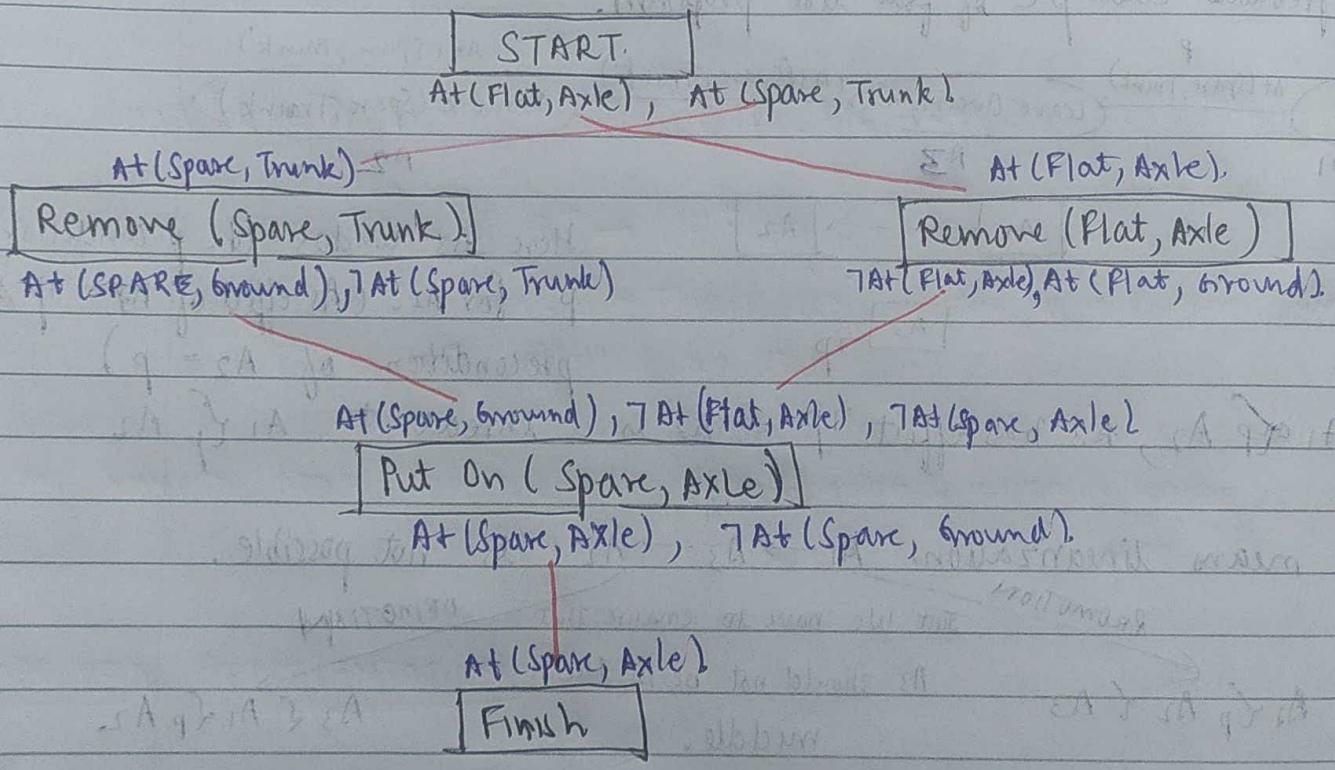
Demotion → Washroom → Start AI Notif → Watching AI.

* But, nothing can be before start, so in demotion, if something is before start, then it is not acceptable. Same point is applicable for promotion. when there is finish.

START $\prec$ Ai $\prec$ FINISH

* So in our previous flat tyre problem. we can do promotion. But that will arise another problem. Leave overnight (As) has ¬At (Spare, Ground) as its effect. so that will be a threat again. So now nothing is possible for Leave Overnight, so we have to remove it.

* Now To satisfy our partial order planning graph, we have to use another action, which is Remove (flat, Axle)



* Heuristic for This can be → you have to ~~minimize flaws~~ choose those actions, which increases the least number of flaws. which has the least number of new flaws added or which has the least no. of pre condition.

* Now our strategy would be to solve complete problem as a constraint satisfaction problem (CSP).

* But CSP has nothing called as a variable, which is temporal.

* Let us take AIR-CARGO problem as an example. For converting a planning problem into CSP, I suffix time on all the states & all the actions. And these will become my variables.

Variables: All time-suffixed literals & actions.

eg ⇒ $At^0(C_1, SFO) = T$, $At^1(C_1, S, F, 0) = F$, $At^2(C_1, SFO) = F$

$Load^1(C_1, P_1, SFO) = T$, $Load^2(C_1, P_1, SFO) = F$.

⇒ So we have to list down all the literals possible & all the actions possible. eg ⇒ for load (C, P, a), no of actions ⇒ (#cargos).(#plane).(#airports).

or I will proportionalize all variables, all actions so that I get ground literals, ground variables & ground actions. After this I suffix time over it. Also we have values associated to these variables, literals & actions.

⇒ A^t ⇒ Precondition A^t (Action applied at time t ⇒ Precond" should be true at t)

A^t ⇒ Effects A^{t+1} (Action applied at time t ⇒ Effect needs to be true at t+1)

⇒ Source variable is always true. This is a fact.

$Source^0 = True$.

Also $Goal^{t_{finish}} = True$ (t_{finish} → time of completion).

$At^0(C_1, SFO)$
$At^0(C_2, JFK)$ — $Load^1(C_1, P_1, SFO)$ →
$At^0(P_1, SFO)$
$At^0(P_2, JFK)$

~~$\neg At^1(C_1, SFO)$~~
$In^1(C_1, P_1)$
$At^1(C_2, JFK)$
$At^1(P_1, SFO)$
$At^1(P_2, JFK)$

* Now there is a constraint that $At^0(C_2, JFK)$ is true, that means $At^1(C_2, JFK)$ is also true. (As we didn't apply any action on C_2).

* But in ρ this 1, if I change the value of $At^1(C_2, JFK)$ from true to false, it will not create any problem as precondition is true & effect is true for action applied. That means we have to apply constraint that will stop $At^1(C_2, JFK)$ value to be false.

* So this adding constraints is called a FRAME Problem.

FRAME PROBLEM ⇒ I need to add a constraint that says if P^t & no action affected p^t, then $p^{t+1} = p^t$. or in other words we can say that if no action applied on a literal ⇒ do nothing.

eg → $Load^1(C_1, P_1, SFO) \Rightarrow (At^t(C_2, JFK) \Leftrightarrow At^{t+1}(C_2, JFK))$

In general we can write, $A^t \Rightarrow (F^t \Leftrightarrow F^{t+1})$

where F ⇒ Fluent / temporal variable that changes with time
F is not affected by A

* we have to apply this constraint for each action & for each frame, i.e. $|A| \times |F|$ time, which is very large.

IMPROVED REPRESENTATION :

→ If I can guess the value of fluent at time t+1, given its value at time t, so we can get rid of that no. of actions in the multiplication. i.e. from $|A| \times |F|$, it will become $|F|$.

$$F^{t+1} \Leftrightarrow \underbrace{\text{Action with Effect } F^t}_{(1)} \vee \underbrace{\left(F^t \wedge \neg \text{Action with Effect } \neg F^t\right)}_{(2)}$$

→ F will be true at time $t+1$ iff

① I apply action with F as effect as time t

② F was true at time t as long as I do not apply an action with $\neg F$ as effect.

eg ⇒ $At^{t+1}(C_1, JFK) \Leftrightarrow (Unload^{t}(C_1, P_1, JFK) \vee Unload(C_1, P_2, JFK)) \vee (At^{t}(C_1, JFK) \wedge (\neg Load(C_1, P_1, JFK) \wedge \neg Load(C_1, P_2, JFK)))$

1

2

★ The Actions are our true variables. The literals & fluent can be derived from the actions applied. So CSP solver needs to find out whether the action will be true or false.