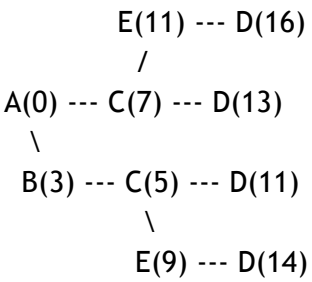
2014

1a.



1b.

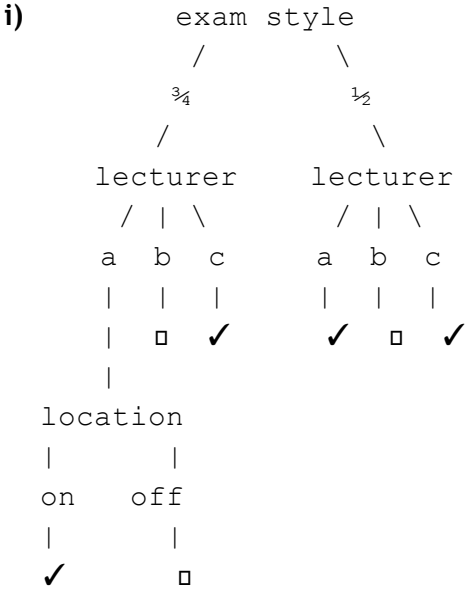
Heuristic h(n) = d_n/v is not good because if the road is perfectly straight (i.e. has length d) and the car goes at a greater speed i.e. v' > v, h*(n) = d_n/v' is smaller than h(n). Thus this heuristic is not *admissible* in all cases. (heuristic h(n) is admissible iff h(n) ≤ h*(n), where h(n) is the *true cost* to get to n)
A better heuristic would be h(n) = d_n/v_n, where d_n - the distance as the crow flies, v_n - the maximum speed allowed on the said road, or if there are multiple maximum speeds its weighted average (or just the greatest).

1c.

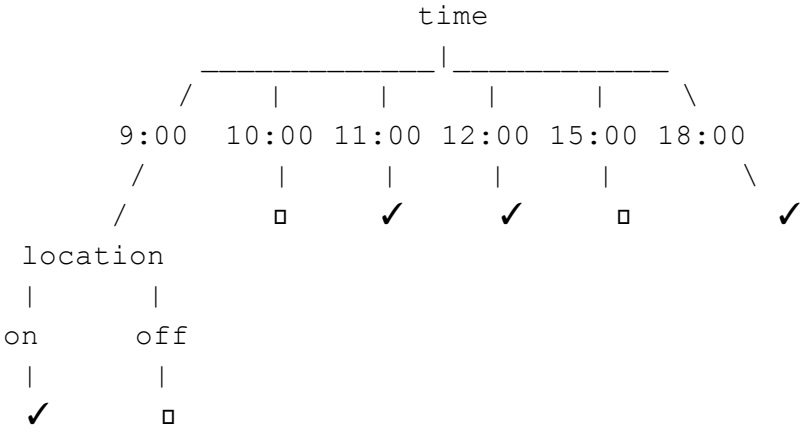
It is a *situation calculus* formula which describes an effect of the Move action.
At a first look this might appear as an effect axiom, yet there are no changes from the first state s to the next; this is a *frame axiom*, describing that a Move does not change the Colour of a block.

2a.

i)



ii) no, the tree in i) is not minimal, since there exists a smaller decision tree, of depth 2, obtained by choosing time and then location.



iii)

$\forall C.\text{choose\_course}(C) \leftarrow \text{exam\_style}(C,¾) \wedge \text{lecturer}(C, a) \wedge \text{location}(C, on)$
$\forall C.\text{choose\_course}(C) \leftarrow \text{exam\_style}(C,¾) \wedge \text{lecturer}(C, c)$
$\forall C.\text{choose\_course}(C) \leftarrow \text{exam\_style}(C,½) \wedge \text{lecturer}(C, a)$
$\forall C.\text{choose\_course}(C) \leftarrow \text{exam\_style}(C,½) \wedge \text{lecturer}(C, c)$

2b.

using set of clauses in 2a.iii) apply backward chaining to choose a course X

i) $\mathbf{S = \{ exam\_style(X, ½) \} \cup S_{iii}}$
$\mathbf{G = \{ chose\_course(X) \}}$
$\theta = \{ \}$

← choose_course(X)
| match choose_course(X) with choose_course(X) ← exam_style(X, ½), lecturer(X, a)
← exam_style(X, ½), lecturer(X, a)
| match exam_style(X, ½) with exam_style(X, ½)
← lecturer(X, a)
|
FAIL
| backtrack
← choose_course(X)
| match choose_course(X) with choose_course(X) ← exam_style(X, ½), lecturer(X, c)
← exam_style(X, ½), lecturer(X, c)
| match exam_style(X, ½) with exam_style(X, ½)
← lecturer(X, c)
|
FAIL
| nothing to backtrack to
FAIL

we don't know whether the course is going to be taken or not since we do not have enough information

ii) S = { exam_style(X, ¾), location(X, on), time(X, 9:00), lecturer(X, a) } ∪ S_{iii}
G = { choose_course(X) }
θ = { }

← choose_course(X)
| match choose_course(X) with choose_course(X) ← exam_style(X, ¾), lecturer(X, a), location(X, on)
← exam_style(X, ¾) ∧ lecturer(X, a) ∧ location(X, on)
| match exam_style(X, ¾) with exam_style(X, ¾)
← lecturer(X, a) ∧ location(X, on)
| match lecturer(X, a) with lecturer(X, a)
← location(X, on)
| match location(X, on) with location(X, on)
□ with θ = { }

the course is to be taken

2c.
i)

choose_course(X) ← lecturer(X, a), not abnormal_course(X)
abnormal_course(X) ← time(X, 9:00), not location(X, on)

ii) S = { location(X, on), time(X, 9:00), lecturer(X, a)} ∪ $S_{i)}$
G = { choose_course(X) }
θ = { }

← choose_course(X)
| match choose_course(X) with choose_course(X) ← lecturer(X, a), not abnormal_course(X)
← lecturer(X, a), not abnormal_course(X)
| match lecturer(X, a) with lecturer(X, a)
← not abnormal_course(X)
| all sub-computations must fail
| ← abnormal_course(X)
| | match abnormal_course(X) with abnormal_course(X) ← time(X, 9:00), not location(X, on)
| ← time(X, 9:00), not location(X, on)
| | match time(X, 9:00) with time(X, 9:00)
| ← not location(X, on)
| | all sub-computations must fail
| | ← location(X, on)
| | | match location(X, on) with location(X, on)
| | □ with θ = { }
| FAIL
| using failure and NAF
□ with θ = { }

the course is to be taken

2013

1a.

Depth-first search might have issues with this task because of the cycles in the graph. Suppose the function that chooses the next node to be expanded will prioritize D over E (e.g. because cost(D,B) > cost(D,E)), thus getting stuck in a loop as following A-B-C-D-B-C-D-B …

Iterative deepening will not have the same issue thanks to the depth limit. Take limit n=4, and suppose the last expanded node is D. The search will get stuck when performing the expansion of B (as opposed to depth-first which will loop), and will backtrack thus choosing E and finding the solution.

1b.

For this planning problem to be suitable for SAT planning, the formulae given need to be translated in CNF (Conjunctive Normal Form). Assuming a maximum of 2 steps are needed for a plan:

Initial Situation: Holds-GotRing-0 ∧ ¬ Holds-Safe-0 ∧ ¬ Holds-AboveChasm-0
Goal state: Holds-Safe-2
Preconditions: (Holds-AboveChasm-0 ∨ Holds-GotRing-0 ∨ Happens-Drop-0) ∧
(Holds-AboveChasm-1 ∨ Holds-GotRing-2 ∨ Happens-Drop-1)
Effects: (Happens-GoChasm-0 ∨ Holds-AboveChasm-1) ∧
(Happens-GoChasm-1 ∨ Holds-AboveChasm-2) ∧
(Happens-Drop-0 ∨ Holds-Safe-1) ∧ (Happens-Drop-1 ∨ Holds-Safe-2)
Frame axioms: (Holds-GotRing-0 ∨ Holds-GotRing-1) ∧
(Holds-GotRing-1 ∨ Holds-GotRing-2) ∧
(Happens-GoChasm-1 ∨ Holds-AboveChasm-2) ∧
(Holds-AboveChasm-0 ∨ Holds-AboveChasm-1) ∧
(Holds-AboveChasm-1 ∨ Holds-AboveChasm-2) ∧
(Holds-Safe-0 ∨ Holds-Safe-1) ∧ (Holds-Safe-1 ∨ Holds-Safe-2)

1c.

in Situation Calculus, given Initial State S_0 and its describing formulae Δ, final state σ, goal G, and set of effect and frame axioms Σ^+ as following

Σ^+ = { (Holds(Safe, Result(Drop, s)) ← Holds(AboveChasm, s) ∧ Holds(GotRing, s))
∧ (Holds(AboveChasm, Result(GoChasm, s))) }
Δ = Holds(GotRing, s_0) ∧ ¬ Holds(Safe, s_0) ∧ ¬ Holds(AboveChasm, s_0)
G = Holds(Safe, σ)

a *plan* is a sequence of actions a_1 to a_m such that
$\Sigma^+ \wedge \Delta \models$ Holds(Safe, σ)
where σ = Result(a_m,Result(a_m−1,…Result(a_1,S_0)…))

In our case, the plan requires a two actions, which are a_1 = GoChasm, a_2 = Drop

2a.

```
domain(Produces, Country)
range(Produces, StapleFood)
domain(CanBeFoundIn, Food)
range(CanBeFoundIn, Country)
domain(Size, Country)
range(Size,Number)
subPropertyOf(Produces, CanBeFoundIn) // redundant?
subClassOf(FastFood, Food)
subClassOf(StapleFood, Food)
type(potatoes, StapleFood)
type(uk, Country)
produces(uk, potatoes)
```

2b.

i) domain(Produces, Country)
range(Produces, StapleFood)
domain(IsProducedBy, StapleFood)
range(IsProducedBy, Country)

ii) sameClassAs(Population, Size)

iii) subClassOf(SelfSufficientCountry, allValuesFrom(Produces, StapleFood))

2c. First split the givens S' in a set of facts E and a set of rules (definite clauses) Pr.

E = { stapleFood(potatoes), country(uk), produces(uk, potatoes) }

Pr = { food(X)←stapleFood(X), food(X)←fastFood(X), canBeFoundIn(X,Y)←produces(Y,X),
country(X)←produces(X,Y), stapleFood(Y)←produces(X,Y),
country(Y)←canBeFoundIn(X,Y), food(X)←canBeFoundIn(X,Y),
country(X)←size(X,Y), number(Y)←size(X,Y)
}

Apply the rules in Pr to the facts in E to obtain E^+.

E' = { food(potatoes)←stapleFood(potatoes), canBeFoundIn(potatoes,uk)←produces(uk,potatoes),
country(uk)←produces(uk,potatoes), stapleFood(potatoes)←produces(uk,potatoes),
country(uk)←canBeFoundIn(potatoes,uk), food(potatoes)←canBeFoundIn(potatoes,uk)
}

with mgu θ = {potatoes/X, UK/Y}

Add the results to the original facts, E = E∪E'. No new facts can be generated since using the formulae which have not been used require a ground X s.t number(X) and also the fact fastFood(potatoes).

2d.

Logic programming view:

S = { stapleFood(potatoes), country(uk), produces(uk, potatoes),
food(X)←stapleFood(X), food(X)←fastFood(X), canBeFoundIn(X,Y)←produces(Y,X),
country(X)←produces(X,Y), stapleFood(Y)←produces(X,Y),
country(Y)←canBeFoundIn(X,Y), food(X)←canBeFoundIn(X,Y),
country(X)←size(X,Y),
number(Y)←size(X,Y) }

G = { food(potatoes) }

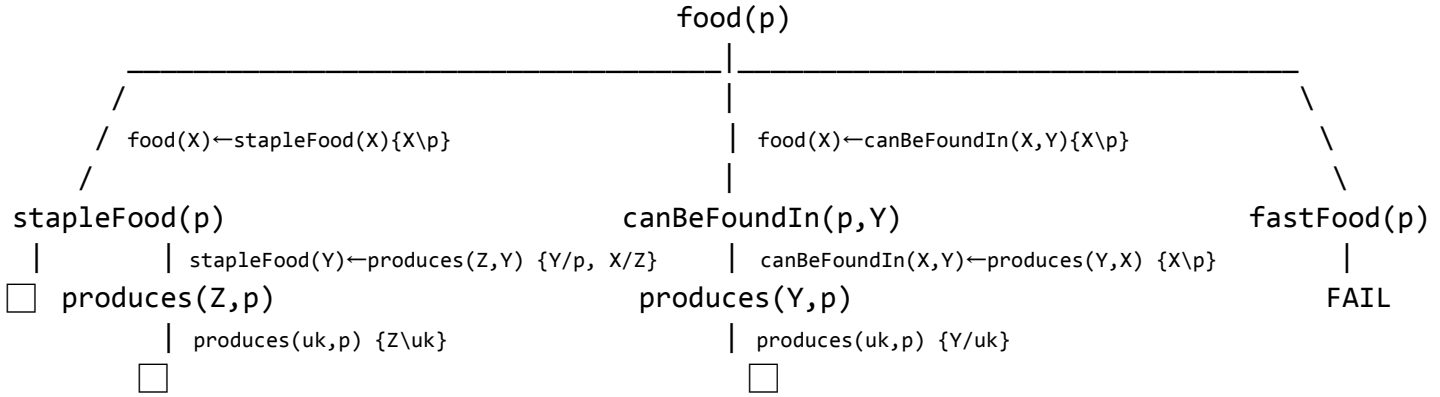
θ_0 = { }

← food(potatoes)
θ_1 | match food(potatoes) with food(X) ← stapleFood(X), with θ_1 = {X\potatoes}
← stapleFood(potatoes)
θ_2 | match stapleFood(potatoes) with stapleFood(Y)←produces(Z, Y) with θ_2 = {Y/potatoes, X/Z}
← produces(Z, potatoes)
θ_3 | match produces(Z, potatoes) with produces(uk, potatoes) with θ_3 = {Z/uk}
□ with θ = {X/potatoes, Y/potatoes, X/Z, Z/uk}

← food(potatoes)
θ_1 | match food(potatoes) with food(X) ← stapleFood(X), with food(potatoes)θ_1 = food(X)θ_1
← stapleFood(X)
θ_1 | match stapleFood(X) with stapleFood(potatoes) with stapleFood(X)θ_1 = stapleFood(potatoes)θ_1
□ with θ = θ_1 = {X/potatoes}

← food(potatoes)
θ_1 | match food(potatoes) with food(X) ← canBeFoundIn(X,Y), with θ_1 = {X\potatoes}
← canBeFoundIn(potatoes, Y)
θ_1 | match canBeFoundIn(potatoes, Y) with canBeFoundIn(X, Y)←produces(Y, X) with θ_1
← produces(Y, potatoes)
θ_2 | match produces(Y, potatoes) with produces(uk, potatoes) with θ_2 = {Y/uk}
□ with θ = {X/potatoes, Y/uk}

Resolution view



2e.
The advantage of FC is that you find all solutions in one go, whereas in BC you have to backtrack for all solutions (and, moreover, can produce extra computations that end up with failure). On the other hand, the disadvantage of FC appears when you have a specific goal in mind, because FC might produces lots of superfluous information while BC will return something specific.

2012

1a.

Suppose DF visits the nodes in alphabetical order, thus having:

H→D→B→J→F→A→L→M→N→K→E→C→G→P→Q

1b.

A search will consider all nodes that have not been expanded yet and expand the one with lowest h*

f(n) = g(n)+h(n)

expand H{f(H)=40, g(H)=0}

→ add to queue D{f(D)=40, g(D)=10}, K{f(K)=42, g(K)=20}

→ expand D since it has smallest f (queue: ~~D(40)~~, K(42))

→ add to queue F{f(F)=77, g(F)=22}, B{f(B)=60, g(B)=30}, L{f(L)=41, g(L)=40}

→ expand L since it has smallest f (queue: F(77), B(60), ~~L(41)~~, K(42))

→ add to queue M{f(M)=65, g(M)=60}, N{f(N)=55, g(N)=55}

→ expand K since it has smallest f (queue: M(65), N(55), F(77), B(60), ~~K(42)~~)

→ add to queue P{f(P)=68, g(P)=38}, E{f(E)=44, g(E)=30}

→ expand E since it has smallest f (queue: P(68), ~~E(44)~~, M(65), N(55), F(77), B(60))

→ add to queue G{f(G)=56, g(G)=50}, C{f(C)=45, g(C)45}

→ expand C since it has smallest f (queue: G(56), ~~C(45)~~, P(68), M(65), N(55), F(77), B(60))

C is a goal node, yet still need to verify for others for a better solution

→ expand N since it has smallest f (queue: G(56), P(68), M(65), ~~N(55)~~, F(77), B(60))

N is a goal node, worse solution than C yet still need to verify the others for a better solution

→ expand G - stuck (queue: ~~G(56)~~, P(68), M(65), F(77), B(60))

→ expand B - worse solution than C, not continuing this path - stuck (queue: P(68), M(65), F(77))

→ expand M - stuck (queue: P(68), F(77))

→ expand P - worse solution than C, not continuing this path - stuck (queue: F(77))

→ expand M - stuck (queue:)

Thus, the solution is N with f(N)=45

1c.

One object can't have two colours

¬(Holds(Colour(x,c)) ∧ Holds(Colour(x,d)) ∧ c≠d)

Effect axioms

Holds(Colour(x,c), Result(Paint(x,c),s)) ← Holds(Dry(x),s)

¬Holds(Colour(x,d), Result(Paint(x,c),s)) ← Holds(Dry(x),s), Holds(Colour(x,d),s), c≠d

¬Holds(Dry(x), Result(Paint(x,c),s)) ← Hold(Dry(x),s)

Frame axioms

Holds(Colour(x,c), Result(Paint(y,d),s)) ← Holds(Dry(y),s), Holds(Colour(x, c), s)

Holds(Dry(x), Result(Paint(y,d),s)) ← Holds(Dry(y),s), Holds(Dry(x),s)

¬Holds(Dry(x), Result(Paint(y,d),s)) ← Holds(Dry(y),s), ¬Holds(Dry(x),s)

2a.
RDF is a descriptive language which allows triples of the form <object-attribute-value>. It is a data model for objects and relations between them. It does not show any meta-information on types or hierarchies, and does not define semantics of any particular domain. It can be represented using XML.
RDF-Schema expands RDF with semantics for general hierarchies. It allows class and property hierarchies (subClassOf, subPropertyOf), relationships between instances and classes (type), restrictions on domains(domain) and values(range) of properties. Does not allow disjointness or boolean combinators(like union), characteristics of properties(like transitivity) or cardinality.
OWL expands RDF schema with the mentioned missing properties. Some of its constructs are sameClassAs, samePropertyAs, allValuesFrom (anonymous classes defined by their properties). It has 3 sub-languages.

2b.

```
movie(X) ← isDirectedBy(X, Y)
director(Y) ← isDirectedBy(X,Y)
movie(x) ← involves(X, Y)
movieCrew(Y) ← involves(X, Y)
movieCrew(X) ← named(X, Y)
string(Y) ← named(X, Y)
```

```
movieCrew(X) ← director(X)
movieCrew(X) ← producer(X)
involves(X, Y) ← isDirectedBy(X, Y)

director(Spielberg)
movie(War Horse)
isDirectedBy(War Horse, Spielberg)
```

2c.
i) isDirectedBy(Y, X) ← directs(X, Y)
ii) important(X) ← director(X), important(X) ← producer(X)
iii) string(X) ← literal(X), literal(X) ← string(X)

2d.
recommended(X) ← movie(X), directedBy(X, Spielberg), not fromNovel(X)
recommended(X) ← movie(X), justReleased(X)
fromNovel(War Horse), justReleased(War Horse)

2e.
← recommended(WH)
θ_1 | match recommended(WH) with recommended(X)←movie(X),directedBy(X,Sp),not fromNovel(X), with θ_1={WH/X}
← movie(WH), directedBy(WH, Sp), not fromNovel(WH)
| match movie(WH)
← directedBy(WH, Sp), not fromNovel(WH)
| match directedBy(WH, Sp)
← not fromNovel(WH)
| all sub-computations must fail
| ← fromNovel(WH)
| | match fromNovel(WH)
| □

FAIL

First computation failed, so attempt the second rule

← recommended(WH)

θ_1 | match recommended(WH) with recommended(X)←movie(X), justReleased(X), with θ_1={WH/X}

← movie(WH), justReleased(WH)

| match movie(WH)

← justReleased(WH)

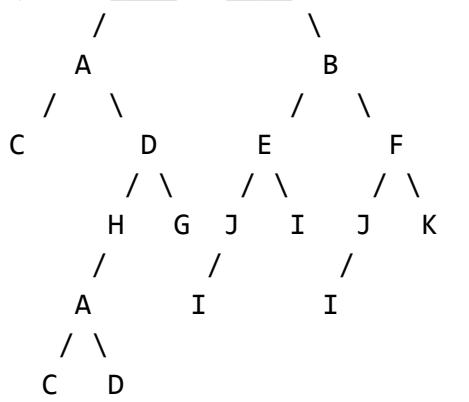
| match justReleased(WH)

□ with θ = {X/WH}

2011

1a.

i)



ii)

DF search in general might not always reach a solution because it might get stuck in the graph's cycles. For example in our situation, assume DF chooses the next node to expand in such a way to give the following path S→A→D→H→A. We can notice a cycle in the graph. If DF does not remember the nodes already visited it will continue expanding D and get stuck in a loop, thus never arriving to I.

iii)

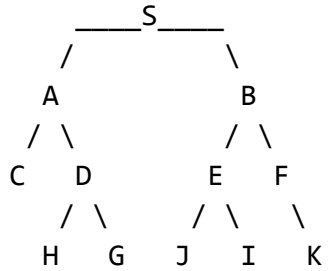
From the search tree in **i)** it is noticeable that goal node I is reachable at depth 3, thus the smallest depth limit that guarantees finding the route is 3.

iv)

LDF is not complete because even though it cannot follow infinitely long paths or get stuck in cycles, it can neither find goals which lie beyond the search depth.

v)

The DF algorithm can be modified in such a way that it never expands a node twice. In this way the search tree at i) gets modified because A will not be expanded furthermore and J and I will be visited only once. The final search tree has only 4 levels (depth = 3).



Another variant would be to use iterative deepening.

1b.

i) Smallest cost is when choosing path S→B→F→J→I and cost is g(I)=3+√2

ii) BF search is not optimal in this case because the shortest path to I(disregarding the costs) guaranteed to be found by BF search is S→B→F→J→I. This path is not optimal in our case because of the "duplicating node" E.

iii) An admissible heuristic needs to be smaller than or equal to the real distance to the node. Therefore a possible choice would be assigning 1 to each arrow, such that the estimated cost is the shortest path in the graph to the goal node. For the nodes which are not connected to the goal node a default value of say, 5 (the longest chain from start node S to goal node I incremented by 1) could be chosen.

$$h(n) = \begin{cases} length(shortest\_path(n, I)), \text{ if } n \text{ connected to } I \\ 5, \text{ if } n \text{ not connected to } I \end{cases}$$

iv) f(E) = g(E) + h(E) = ((1+1)*2) + 1 = 5
f(F) = g(F) + h(F) = (1+1) + (1+1) = 4
The first node expanded by A* will be F since it has a lower estimated cost.

v) Let C be the cost of the *shortest path* to goal node G.
Suppose A* selects a sub-optimal goal node G' s.t $g(G') > C$
Suppose there exists an unexpanded node n such that n is on the lowest cost path to G.
h is admissible, thus $f(n) = g(n) + h(n) \leq C$ (by definition of admissibility).
but the path to G' is not the shortest, i.e $f(G') > C$ since $g(G') > C$
thus $f(n) \leq C < f(G')$ therefore A* must select n before reaching the sub-optimal goal G', thus guaranteeing the fact it will always reach the optimal goal node.

For the previous proof, suppose $h(n)$ is not admissible.
Therefore $f(n) = g(n) + h(n) > C$.
This brings us to $f(n) > C$ and $f(G') > C$ from which we can draw no conclusion on nodes' priorities.

2a.

The axioms of Event Calculus

HoldsAt(f,t) ← Initially(f) ∧ ¬ Clipped(0,f,t)
HoldsAt(f,t2) ← ∃ e,t1 [Happens(e,t1) ∧ Initiates(e,f,t1) ∧ t1 < t2 ∧ ¬ Clipped(t1,f,t2)]
Clipped(t1,f,t3) ↔ ∃ e,t2 [Happens(e,t2) ∧ t1 < t2 < t3 ∧ Terminates(e,f,t2)]

2b.

fluents: Empty, Clean, Watered
actions: Fill, Water, Clean_up

initial state: Initially(Full) ∧ Initially(Dirty)

goal state: HoldsAt(Clean, t) ∧ HoldsAt(Watered, t)

axioms:
Initiates(Water, Watered, t) ← HoldsAt(Full, t)
Initiates(Water, Empty, t) ← ¬HoldsAt(Empty, t)
Terminates(Fill, Empty, t) ← HoldsAt(Empty, t)
Initiates(Clean_up, Clean, t) ← ¬HoldsAt(Clean, t)
Terminates(Fill, Clean, t) ← HoldsAt(Empty, t)

2c.
In abductive logic programs, given
• T (theory presentation) - initial state and axioms
T = { Full ∧ Dirty ∧ (Watered ← Water ∧ Full) ∧
(Empty ← Water ∧ ¬Empty) ∧ (¬Empty ← Fill ∧ Empty) ∧
(Clean ← Clean_up ∧ ¬Clean) ∧ (¬Clean ← Fill ∧ Empty) }
• H (candidate hypotheses) - possible actions a robot can make at time step t
H = { Empty ∧ Clean_up ∧ Water }
• O (observation) - the goal state
O = { Watered ∧ Clean }
Find E (explanation) such that
1) $T \cup E \vdash O$
2) $T \cup E$ is consistent (equivalently $T \cup E \nvdash$ false)
3) $E \subseteq H$

2d. to be continued