SAPIENZA Università di Roma – MSc. in Engineering in Computer Science
Artificial Intelligence & Machine Learning – July 15, 2024

Part 1 - Artificial Intelligence
(Time to complete the test: 2:30 hours)

A wolf, a sheep and a cabbage need to be transported from the left to the right side of a river. A boat can be used to cross the river, which can transport at most two objects at a time.

Assume this scenario is modelled as follows:

Constants:

- $boat$, denoting the boat.
- $left$ and $right$, denoting respectively the left and the right side of the river.
- $wolf$, $sheep$ and $cabbage$, denoting respectively the wolf, the sheep, and the cabbage.

Non-Fluents: CANNOT CHANGE

- $Opposite(x, y)$, denoting that side x is opposite to side y.

Fluents: CHANGE

- $At(x, y)$ denoting that object x is on side y.
- $On(x)$ denoting that object x is on the boat.

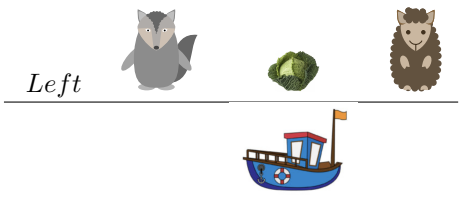
PREDICATES

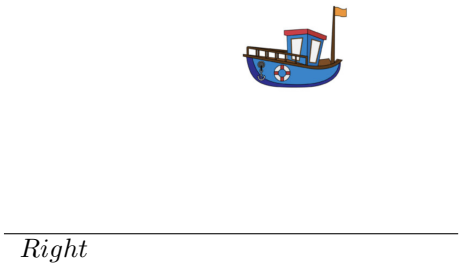
Actions:

- $load(x)$, which allows for loading object x on the boat.
  The action can be done only if:
  - x is not the boat;
  - x and the boat are on the same side;
  - the boat does not contain two objects already.

  The effect is that object x is on the boat and no longer on the riverside.
- $cross()$, which allows the boat to cross the river. The action can always be done. The effect is that the boat is on the opposite side with respect to that it was on when the action was executed;
- $unload()$, which allows all the objects on the boat to be unloaded. The action can be done only if there is something on the boat. The effect is that all the objects on the boat are no longer on it but on the side where the boat is.

Initial situation:
The only objects are the sheep, the cabbage, the wolf and the boat. All of them are on the left side. Nothing is on the boat. The left side is opposite to the right one and viceversa.





Exercise 1.

1. Formalize the above scenario as a Basic Action Theory.
2. Using regression, check whether the action sequence

$$\varrho_1 = load(wolf); load(sheep); cross(); unload(); cross()$$

   is executable in S_0.
3. Using regression, check whether the action sequence:

$$\varrho_2 = load(wolf); load(sheep); cross()$$

   leads to a situation where the sheep is on some side.

Exercise 2.

1. Considering goal $\gamma = \neg At(wolf, left) \wedge \neg At(sheep, left) \wedge \neg At(cabbage, left) \wedge \neg At(boat, left)$, formalize the above scenario as a PDDL domain file and a PDDL problem file;
2. Draw a fragment of the corresponding transition system such that:
   - the fragment includes at least 12 states;
   - for every state included in the fragment, all the executable actions (i.e., outgoing edges) are present, even if the successor state is not present in the fragment;
   - the fragment includes a plan for goal γ, together with all the states the plan reaches.
3. Solve planning for achieving γ using uninformed forward depth-first search, reporting the steps of the forward search computation and showing, in particular, the evolution of the open set (stack). Report the returned plan.

Exercise 3.

1. Consider the following FOL formulas:
   - $\phi_1 : \forall x \forall y \forall z.(Wolf(x) \wedge Sheep(y) \wedge At(x, z) \wedge At(y, z)) \supset Eats(x, y)$
   - $\phi_2 : \forall x \forall y \forall z.(At(x, z) \wedge Eats(x, y)) \supset \neg At(y, z)$
   - $\phi_3 : \forall x \forall y.(Wolf(x) \wedge At(x, y)) \supset \neg \exists z.Sheep(z) \wedge At(z, y) \wedge Eats(x, z)$

   Using the tableau method, check whether $\{\phi_1, \phi_2\} \models \phi_3$. If this is not the case, show a counter-model obtained from the tableau.

Exercise 2.

1. Considering goal $\gamma = \neg At(wolf, left) \wedge \neg At(sheep, left) \wedge \neg At(cabbage, left) \wedge \neg At(boat, left)$, formalize the above scenario as a PDDL domain file and a PDDL problem file;
2. Draw a fragment of the corresponding transition system such that:
   - the fragment includes at least 12 states;
   - for every state included in the fragment, all the executable actions (i.e., outgoing edges) are present, even if the successor state is not present in the fragment;
   - the fragment includes a plan for goal γ, together with all the states the plan reaches.
3. Solve planning for achieving γ using uninformed forward depth-first search, reporting the steps of the forward search computation and showing, in particular, the evolution of the open set (stack). Report the returned plan.

1)

```
(DEFINE (DOMAIN RIVER_DOMAIN)  :ADL → ·TYPING :EQUALITY .DISJUCTIVE-PRECONDITIONS
                                      :QUANTIFIED-PRECONDITIONS .CONDITIONAL-EFFECTS
  (:REQUIREMENTS :ADL)
  (:TYPES SIDE ITEM)
                                  LIST                TYPE
  (:CONSTANTS LEFT RIGHT - SIDE   BOAT CABBAGE SHEEP WOLF - ITEM)
  (:PREDICATES
    (OPPOSITE ?x1 ?x2 - SIDE)
    (AT ?x - ITEM ?y - SIDE)
    (ON ?x - ITEM)
  )
  (:ACTION LOAD
    :PARAMETERS (?x - ITEM)
    :PRECONDITION (AND
                    (NOT (= ?x BOAT))
                    (EXISTS (?s - SIDE) (AND (AT ?x ?s)(AT BOAT ?s)))
                    (NOT (EXISTS (?x ?y - ITEM)(AND (NOT (= ?x ?y)) (ON ?x)(ON ?y))))
                  )
    :EFFECT (AND (NOT (EXISTS (?s - SIDE) (AT ?x ?s))) (ON ?x))
  ): END OF LOAD
  (:ACTION CROSS
    :PARAMETERS ()
    :PRECONDITION ()
    :EFFECT (FORALL (?x ?y - SIDE)
              (WHEN                                          WHEN
                (AND (AT BOAT ?x) (OPPOSITE ?x ?y))          CONDITION
                (AND (NOT (AT BOAT ?x))(AT BOAT ?y))         EFFECT
              )
            )
  ); END OF CROSS
  (:ACTION UNLOAD
    :PARAMETERS ()
    :PRECONDITION (EXISTS (?x - ITEM)(ON ?x))
    :EFFECT (FORALL (?x - ITEM ?s - SIDE)
              (WHEN
                (AND (AT BOAT ?s)(ON ?x))
                (AND (NOT (ON ?x))(AT ?x ?s))
              )
            )
  ); END OF UNLOAD
); END OF DEFINE DOMAIN
```

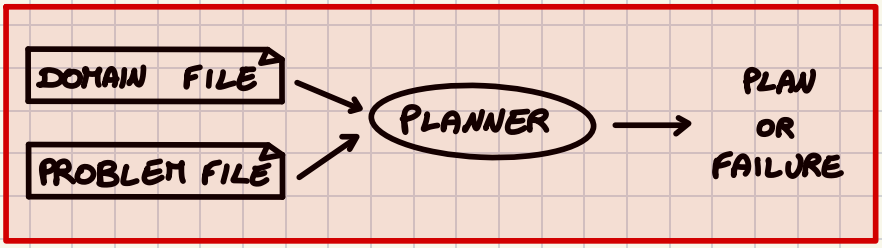
AFTER DEFINED THE DOMAIN FILE, WE DEFINE THE PROBLEM FILE.

γ: $\neg AT(WOLF, LEFT) \wedge \neg AT(SHEEP, LEFT) \wedge \neg AT(CABBAGE, LEFT) \wedge \neg AT(BOAT, LEFT)$

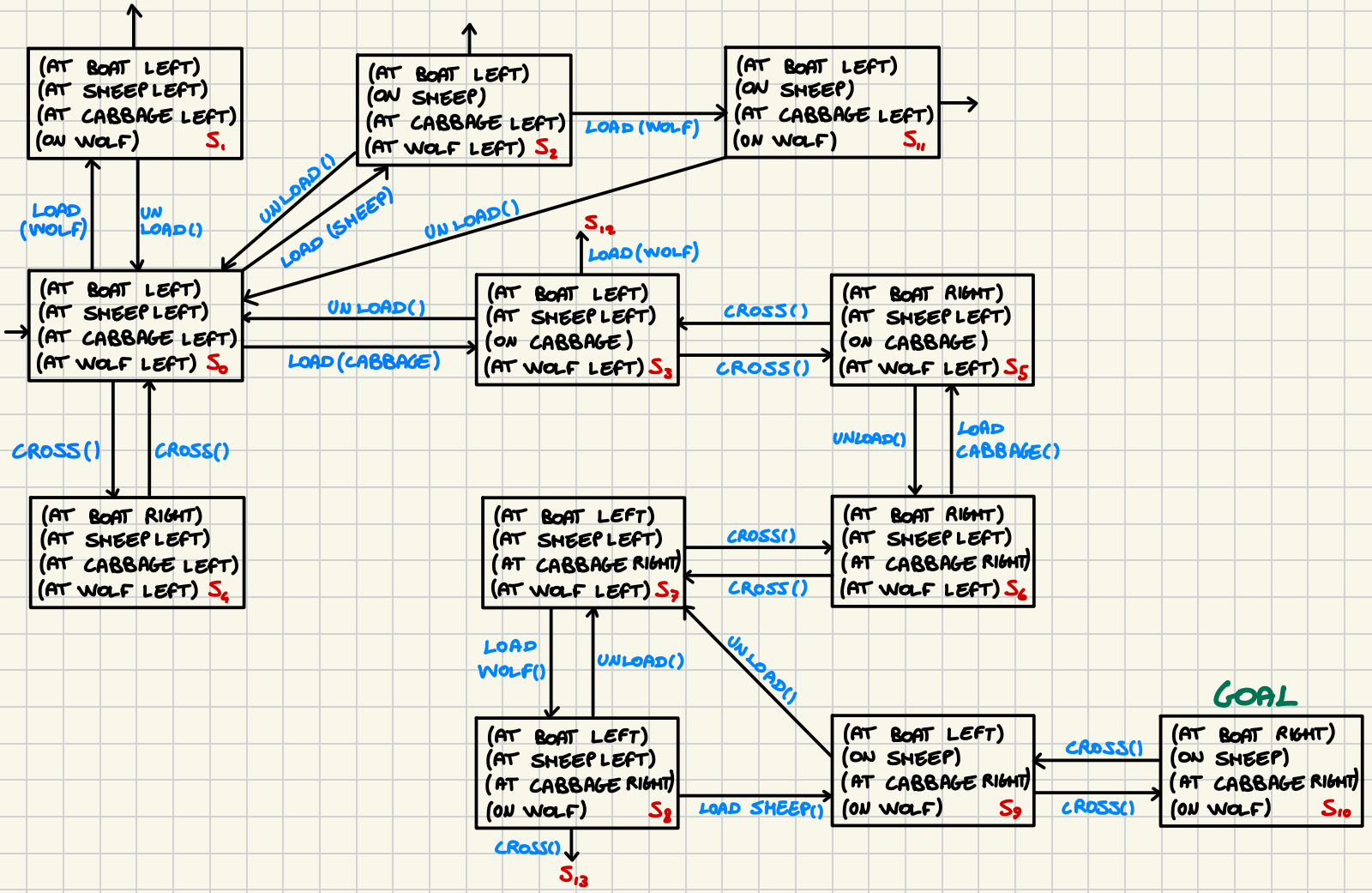
```
(DEFINE (PROBLEM RIVER_PROBLEM) (:DOMAIN RIVER_DOMAIN)
  (:OBJECTS )  WE HAVE NO ADDITIONAL COSTANTS
  (:INIT (AT BOAT LEFT)(AT WOLF LEFT)(AT SHEEP LEFT)(AT CABBAGE LEFT)
         (OPPOSITE RIGHT LEFT)(OPPOSITE LEFT RIGHT))
  (:GOAL (AND
           (NOT (AT BOAT LEFT))
           (NOT (AT WOLF LEFT))
           (NOT (AT SHEEP LEFT))
           (NOT (AT CABBAGE LEFT))
         )
  )
); END OF DEFINE PROBLEM
```

OR →

```
(:GOAL
  (FORALL (?x - ITEM) (NOT (AT ?x LEFT)))
)
```



2) * OPPOSITE NOT REPORTED BECAUSE CANNOT CHANGE (OPPOSITE RIGHT LEFT) (OPPOSITE LEFT RIGHT)



3) $\gamma = \neg AT(WOLF, LEFT) \wedge \neg AT(SHEEP, LEFT) \wedge \neg AT(CABBAGE, LEFT) \wedge \neg AT(BOAT, LEFT)$

DFS Algorithm (variant with marking check before pushing):

```
DFS(domain d, state init, formula goal){
  // d:  input domain;
  // init:  initial state;
  // goal:  goal formula;
  Stack t = [(init, empty)]; // Open set (stack)
  Set m = {init}; // Marked states
  while (!t.empty()) {
    (state,plan) = t.pop(); // plan is action sequence followed to reach s
    if (s ⊨ goal) return plan; // if s satifies goal state, return plan
    forall (actions a executable in s)
      s'=d.delta(s,a) // s' is successor of s under a (d.delta is transition function of d)
      if (s' ∉ m) {// if s' not marked
        m.add(s'); // mark s'
        t.push((s',plan·a)); // add s' with plan extended by a, to open set t}
  }
  return noplan; // no plan found
}
```

MI BASO SULLA MIA SOLUZIONE 2)

OLTRE DFS, BISOGNA VEDERE BFS E A*

0. $\mathcal{T} = [(S_0, EMPTY)]$
   $m = \{S_0\}$

1. (STATE, PLAN) = $\mathcal{T}$.POP () $(S_0, EMPTY)$
   m ADD (S_1)
   $\mathcal{T}$. PUSH $((S_1, LOAD(WOLF)))$
   m ADD (S_2)
   $\mathcal{T}$. PUSH $((S_2, LOAD(SHEEP)))$
   m ADD (S_4)
   $\mathcal{T}$. PUSH $((S_4, CROSS()))$
   m ADD (S_3)
   $\mathcal{T}$. PUSH $((S_3, LOAD(CABBAGE)))$

CONTENT OF $\mathcal{T}$ AND m AT END OF ITERATION:

$\mathcal{T} = [(S_3, LOAD(CABBAGE)),$
$(S_4, CROSS()),$
$(S_2, LOAD(SHEEP)),$
$(S_1, LOAD(WOLF))]$

$m = \{S_0, S_1, S_2, S_3, S_4\}$

2. (STATE, PLAN) = $\mathcal{T}$.POP() (S_3, LOAD(CABBAGE))
m.ADD(S_{12})
$\mathcal{T}$.PUSH((S_{12}, LOAD(CABBAGE) LOAD(WOLF)))
m.ADD(S_5)
$\mathcal{T}$.PUSH((S_5, LOAD(CABBAGE) CROSS()))

CONTENT OF $\mathcal{T}$ AND m AT END OF ITERATION:

$\mathcal{T}$ = [(S_5, LOAD(CABBAGE) CROSS()),
(S_{12}, LOAD(CABBAGE) LOAD(WOLF)),
(S_4, CROSS()),
(S_2, LOAD(SHEEP)),
(S_1, LOAD(WOLF))]

m = {$S_0, S_1, S_2, S_3, S_4, S_5, S_{12}$}

3. (STATE, PLAN) = $\mathcal{T}$.POP() (S_5, LOAD(CABBAGE) CROSS())
m.ADD(S_6)
$\mathcal{T}$.PUSH((S_6, LOAD(CABBAGE) CROSS() UNLOAD()))

CONTENT OF $\mathcal{T}$ AND m AT END OF ITERATION:

$\mathcal{T}$ = [(S_6, LOAD(CABBAGE) CROSS() UNLOAD()),
(S_{12}, LOAD(CABBAGE) LOAD(WOLF)),
(S_4, CROSS()),
(S_2, LOAD(SHEEP)),
(S_1, LOAD(WOLF))]

m = {$S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_{12}$}

4 (STATE, PLAN) = $\mathcal{T}$.POP() (S_6, LOAD(CABBAGE) CROSS() UNLOAD())
m.ADD(S_7)
$\mathcal{T}$.PUSH((S_7, LOAD(CABBAGE) CROSS() UNLOAD() CROSS()))

$\mathcal{T}$ = [(S_7, LOAD(CABBAGE) CROSS() UNLOAD() CROSS()),
(S_{12}, LOAD(CABBAGE) LOAD(WOLF)),
(S_4, CROSS()),
(S_2, LOAD(SHEEP)),
(S_1, LOAD(WOLF))]

m = {$S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_{12}$}

5. (STATE, PLAN) = $\mathcal{T}$.POP() (S_7, LOAD(CABBAGE) CROSS() UNLOAD() CROSS())
m.ADD(S_8)
$\mathcal{T}$.PUSH((S_8, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF)))

$\mathcal{T}$ = [(S_8, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF)),
(S_{12}, LOAD(CABBAGE) LOAD(WOLF)),
(S_4, CROSS()),
(S_2, LOAD(SHEEP)),
(S_1, LOAD(WOLF))]

m = {$S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_{12}$}

6. (STATE, PLAN) = $\mathcal{T}$.POP() (S_8, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF))

m.ADD(S_{13})
$\mathcal{T}$.PUSH((S_{13}, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF) CROSS()))
m.ADD(S_9)
$\mathcal{T}$.PUSH((S_9, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF) LOAD(SHEEP)))

$\mathcal{T}$ = [(S_9, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF) LOAD(SHEEP)),
(S_{13}, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF) CROSS()),
(S_{12}, LOAD(CABBAGE) LOAD(WOLF)),
(S_4, CROSS()),
(S_2, LOAD(SHEEP)),
(S_1, LOAD(WOLF))]

m = {$S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{12}, S_{13}$}

7. (STATE, PLAN) = $\mathcal{T}$.POP() (S_9, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF) LOAD(SHEEP))

m.ADD(S_{10})
$\mathcal{T}$.PUSH((S_{10}, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF) LOAD(SHEEP) CROSS()))

$\mathcal{T}$ = [(S_{10}, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF) LOAD(SHEEP) CROSS()),
(S_{13}, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF) CROSS()),
(S_{12}, LOAD(CABBAGE) LOAD(WOLF)),
(S_4, CROSS()),
(S_2, LOAD(SHEEP)),
(S_1, LOAD(WOLF))]

m = {$S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}, S_{12}, S_{13}$}

8. (STATE, PLAN) = $\mathcal{T}$.POP()

(S_{10}, LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF) LOAD(SHEEP) CROSS())

RETURNED PLAN: LOAD(CABBAGE) CROSS() UNLOAD() CROSS() LOAD(WOLF) LOAD(SHEEP) CROSS()

Solution

Precondition Axioms:

- $Poss(load(x), s) \equiv x \neq boat \wedge (\exists y.At(boat, y, s) \wedge At(x, y, s)) \wedge \neg(\exists w \exists z.w \neq z \wedge On(w, s) \wedge On(z, s))$
- $Poss(cross(), s) \equiv true$
- $Poss(unload(), s) \equiv \exists x.On(x, s)$

Successor State Axioms:

1. Start with *Effect Axioms*:
   - $a = load(x) \supset (On(x, do(a, s)) \wedge (At(x, y, s) \supset \neg At(x, y, do(a, s))))$
   - $a = cross() \supset (At(boat, x, s) \supset (\neg At(boat, x, do(a, s)) \wedge (Opposite(x, y) \supset At(boat, y, do(a, s)))))$
   - $a = unload() \supset (\neg On(x, do(a, s)) \wedge ((On(x, s) \wedge At(boat, z, s)) \supset At(x, z, do(a, s))))$
2. Normalize Effect Axioms:
   - $a = load(x) \supset On(x, do(a, s))$
   - $a = load(x) \wedge At(x, y, s) \supset \neg At(x, y, do(a, s))$
   - $a = cross() \wedge x = boat \wedge At(x, y, s) \supset \neg At(x, y, do(a, s))$
   - $a = cross() \wedge x = boat \wedge (\exists z.At(x, z, s) \wedge Opposite(z, y)) \supset At(x, y, do(a, s))$
   - $a = unload() \supset \neg On(x, do(a, s))$
   - $a = unload() \wedge On(x, s) \wedge At(boat, y, s) \supset At(x, y, do(a, s))$
3. Apply *Explanation Closure* to obtain Successor-State Axioms:
   - $At(x, y, do(a, s)) \equiv (a = cross() \wedge x = boat \wedge (\exists z.At(x, z, s) \wedge Opposite(z, y))) \vee$
     $(a = unload() \wedge On(x, s) \wedge At(boat, y, s)) \vee$
     $(At(x, y, s) \wedge \neg((a = load(x) \wedge At(x, y, s)) \vee (a = cross() \wedge x = boat \wedge At(x, y, s))))$
     which simplifies as follows:
     $At(x, y, do(a, s)) \equiv (a = cross() \wedge x = boat \wedge (\exists z.At(x, z, s) \wedge Opposite(z, y))) \vee$
     $(a = unload() \wedge On(x, s) \wedge At(boat, y, s)) \vee$
     $(At(x, y, s) \wedge \neg(a = load(x) \vee (a = cross() \wedge x = boat)))$
   - $On(x, do(a, s)) \equiv a = load(x) \vee (On(x, s) \wedge \neg a = unload()).$

Initial Situation:

$\mathcal{D}_{S_0}$ is the set including the following formulas:

- $Opposite(x, y) \equiv (x = right \wedge y = left) \vee (y = right \wedge x = left)$
- $At(x, y, S_0) \equiv y = left \wedge (x = wolf \vee x = sheep \vee x = cabbage \vee x = boat)$
- $\forall x.\neg On(x, S_0)$

Regression:

For $\varrho_1 = load(wolf); load(sheep); cross(); unload(); cross()$, let:
$S_1 = do(load(wolf), S_0)$
$S_2 = do(load(sheep), S_1)$
$S_3 = do(cross(), S_2)$
$S_4 = do(unload(), S_3)$
$S_5 = do(cross(), S_4)$

To check whether ϱ_1 is executable in S_0, we need to check whether:
$load(wolf)$ is executable in S_0
$load(sheep)$ is executable in S_1
$cross()$ is executable in S_2
$unload()$ is executable in S_3
$cross()$ is executable in S_4

These are equivalent to checking whether:

- $\mathcal{D}_0 \cup \mathcal{D}_{una} \models \mathcal{R}[Poss(load(wolf), S_0)]$;
- $\mathcal{D}_0 \cup \mathcal{D}_{una} \models \mathcal{R}[Poss(load(sheep), S_1)]$;
- $\mathcal{D}_0 \cup \mathcal{D}_{una} \models \mathcal{R}[Poss(cross(), S_2)]$.
- $\mathcal{D}_0 \cup \mathcal{D}_{una} \models \mathcal{R}[Poss(unload(), S_3)]$.
- $\mathcal{D}_0 \cup \mathcal{D}_{una} \models \mathcal{R}[Poss(cross(), S_4)]$.

Let's regress each of the formulas above.

- $\mathcal{R}[Poss(load(wolf), S_0)] =$
  $\mathcal{R}[wolf \neq boat \wedge (\exists y.At(boat, y, S_0) \wedge At(wolf, y, S_0)) \wedge \neg(\exists w \exists z.w \neq z \wedge On(w, S_0) \wedge On(z, S_0))] =$
  $wolf \neq boat \wedge (\exists y.At(boat, y, S_0) \wedge At(wolf, y, S_0)) \wedge \neg(\exists w \exists z.w \neq z \wedge On(w, S_0) \wedge On(z, S_0))]$,
  which is true for $y = left$, thus $load(wolf)$ is executable in S_0.
- $\mathcal{R}[Poss(load(sheep), S_1)] =$
  $\mathcal{R}[sheep \neq boat \wedge (\exists y.At(boat, y, S_1) \wedge At(sheep, y, S_1)) \wedge \neg(\exists w \exists z.w \neq z \wedge On(w, S_1) \wedge On(z, S_1))] =$
  $(\exists y.\mathcal{R}[At(boat, y, S_1)] \wedge \mathcal{R}[At(sheep, y, S_1)]) \wedge \neg(\exists w \exists z.w \neq z \wedge \mathcal{R}[On(w, S_1)] \wedge \mathcal{R}[On(z, S_1)])$

  We have:
  $\mathcal{R}[At(boat, y, S_1)] =$
  $\mathcal{R}[At(boat, y, do(load(wolf), S_0))] =$
  $\mathcal{R}[(load(wolf) = cross() \wedge boat = boat \wedge (\exists z.At(boat, z, S_0) \wedge Opposite(z, y))) \vee$
  $(load(wolf) = unload() \wedge On(boat, S_0) \wedge At(boat, y, S_0)) \vee$
  $(At(boat, y, S_0) \wedge \neg(load(wolf) = load(boat) \vee (load(wolf) = cross() \wedge boat = boat)))] =$
  $(load(wolf) = cross() \wedge boat = boat \wedge (\exists z.\mathcal{R}[At(boat, z, S_0)] \wedge \mathcal{R}[Opposite(z, y)])) \vee$
  $(load(wolf) = unload() \wedge \mathcal{R}[On(boat, S_0)] \wedge \mathcal{R}[At(boat, y, S_0)]) \vee$
  $(\mathcal{R}[At(boat, y, S_0)] \wedge \neg(load(wolf) = load(boat) \vee (load(wolf) = cross() \wedge boat = boat))) =$
  $(false \wedge true \wedge (\exists z.\mathcal{R}[At(boat, z, S_0)] \wedge \mathcal{R}[Opposite(z, y)])) \vee$
  $(false \wedge \mathcal{R}[On(boat, S_0)] \wedge \mathcal{R}[At(boat, y, S_0)]) \vee$
  $(At(boat, y, S_0) \wedge \neg(false \vee (false \wedge true))) =$
  $(false) \vee (false) \vee At(boat, y, S_0) \wedge true = At(boat, y, S_0)$

  Almost analogously, we obtain: $\mathcal{R}[At(sheep, y, S_1)] = At(sheep, y, S_0)$

  Moreover, we have:
  $\mathcal{R}[On(w, S_1)] =$
  $\mathcal{R}[On(w, do(load(wolf), S_0))] =$
  $\mathcal{R}[load(wolf) = load(w) \vee (On(w, S_0) \wedge \neg load(wolf) = unload())] =$
  $load(wolf) = load(w) \vee (On(w, S_0) \wedge \neg load(wolf) = unload()) =$
  $load(wolf) = load(w) \vee On(w, S_0) =$
  $w = wolf \vee On(w, S_0) =$
  $w = wolf$, since $On(w, S_0)$ is false for any w, as the boat contains no object in S_0.

  Analogously: $\mathcal{R}[On(z, S_1)] = z = wolf$

  Summing up:
  $\mathcal{R}[Poss(load(sheep), S_1)] =$
  $(\exists y.\mathcal{R}[At(boat, y, S_1)] \wedge \mathcal{R}[At(sheep, y, S_1)]) \wedge \neg(\exists w \exists z.w \neq z \wedge \mathcal{R}[On(w, S_1)] \wedge \mathcal{R}[On(z, S_1)]) =$
  $(\exists y.At(boat, y, S_0) \wedge At(sheep, y, S_0)) \wedge \neg(\exists w \exists z.w \neq z \wedge w = wolf \wedge z = wolf) =$
  $(\exists y.At(boat, y, S_0) \wedge At(sheep, y, S_0)) \wedge \neg false =$
  $(\exists y.At(boat, y, S_0) \wedge At(sheep, y, S_0))$,
  which is true for $y = left$, thus $load(sheep)$ is executable in S_1
- $\mathcal{R}[Poss(cross(), S_2)] = true$, thus $cross()$ is executable in S_2
- $\mathcal{R}[Poss(unload(), S_3)] =$
  $\mathcal{R}[\exists x.On(x, S_3)] =$
  $\mathcal{R}[\exists x.On(x, do(cross(), S_2))] =$
  $\exists x.\mathcal{R}[cross() = load(x) \vee (On(x, S_2) \wedge \neg cross() = unload())] =$
  $\exists x.false \vee (\mathcal{R}[On(x, S_2)] \wedge \neg false) =$
  $\exists x.\mathcal{R}[On(x, S_2)] =$
  $\exists x.\mathcal{R}[On(x, do(load(sheep), S_1))] =$
  $\exists x.\mathcal{R}[load(sheep) = load(x) \vee (On(x, S_1) \wedge \neg load(sheep) = unload())] =$
  $\exists x.load(sheep) = load(x) \vee \mathcal{R}[On(x, S_1)]$,
  which is true for $x = sheep$, thus $unload()$ is executable in S_3.

- $\mathcal{R}[Poss(cross(), S_4)] = true$, thus $cross()$ is executable in S_4.

We thus conclude that ϱ_1 is executable in S_0.

For $\varrho_2 = load(wolf); load(sheep); cross()$, let S_1, S_2, and S_3 be as above.

To check whether ϱ_2 results in a situation where the sheep is on some side, we need to check whether:

$$\mathcal{D}_0 \cup \mathcal{D}_{una} \models \mathcal{R}[\exists y.At(sheep, y, S_3)]$$

Let's regress the formula:
$\mathcal{R}[\exists y.At(sheep, y, S_3)] =$
$\exists y.\mathcal{R}[At(sheep, y, do(cross(), S_2))] =$
$\exists y.\mathcal{R}[(cross() = cross() \wedge sheep = boat \wedge (\exists z.At(sheep, z, S_2) \wedge Opposite(z, y))) \vee$
$(cross() = unload() \wedge On(sheep, S_2) \wedge At(boat, y, S_2)) \vee$
$(At(sheep, y, S_2) \wedge \neg(cross() = load(sheep) \vee (cross() = cross() \wedge sheep = boat)))] =$
$\exists y.(true \wedge false \wedge (\exists z.\mathcal{R}[At(sheep, z, S_2) \wedge Opposite(z, y)])) \vee$
$(false \wedge \mathcal{R}[On(sheep, S_2)] \wedge \mathcal{R}[At(boat, y, S_2)]) \vee$
$(\mathcal{R}[At(sheep, y, S_2)] \wedge \neg(false \vee (true \wedge false))) =$
$\exists y.\mathcal{R}[At(sheep, y, S_2)] =$
$\exists y.\mathcal{R}[At(sheep, y, do(load(sheep), S_1))] =$
$\exists y.\mathcal{R}[(load(sheep) = cross() \wedge sheep = boat \wedge (\exists z.At(sheep, z, S_1) \wedge Opposite(z, y))) \vee$
$(load(sheep) = unload() \wedge On(sheep, S_1) \wedge At(boat, y, S_1)) \vee$
$(At(sheep, y, S_1) \wedge \neg(load(sheep) = load(sheep) \vee (load(sheep) = cross() \wedge sheep = boat)))] =$
$\exists y.(false \wedge false \wedge (\exists z.\mathcal{R}[At(sheep, z, S_1) \wedge Opposite(z, y)])) \vee$
$(false \wedge \mathcal{R}[On(sheep, S_1)] \wedge \mathcal{R}[At(boat, y, S_1)]) \vee$
$(\mathcal{R}[At(sheep, y, S_1)] \wedge \neg(true \vee (false \wedge false))) =$
$\exists y.(\mathcal{R}[At(sheep, y, S_1)] \wedge false) = false$
Thus, ϱ_2 does not result in a situation where the sheep is on some side (it is indeed on the boat).

PDDL:
Domain file:

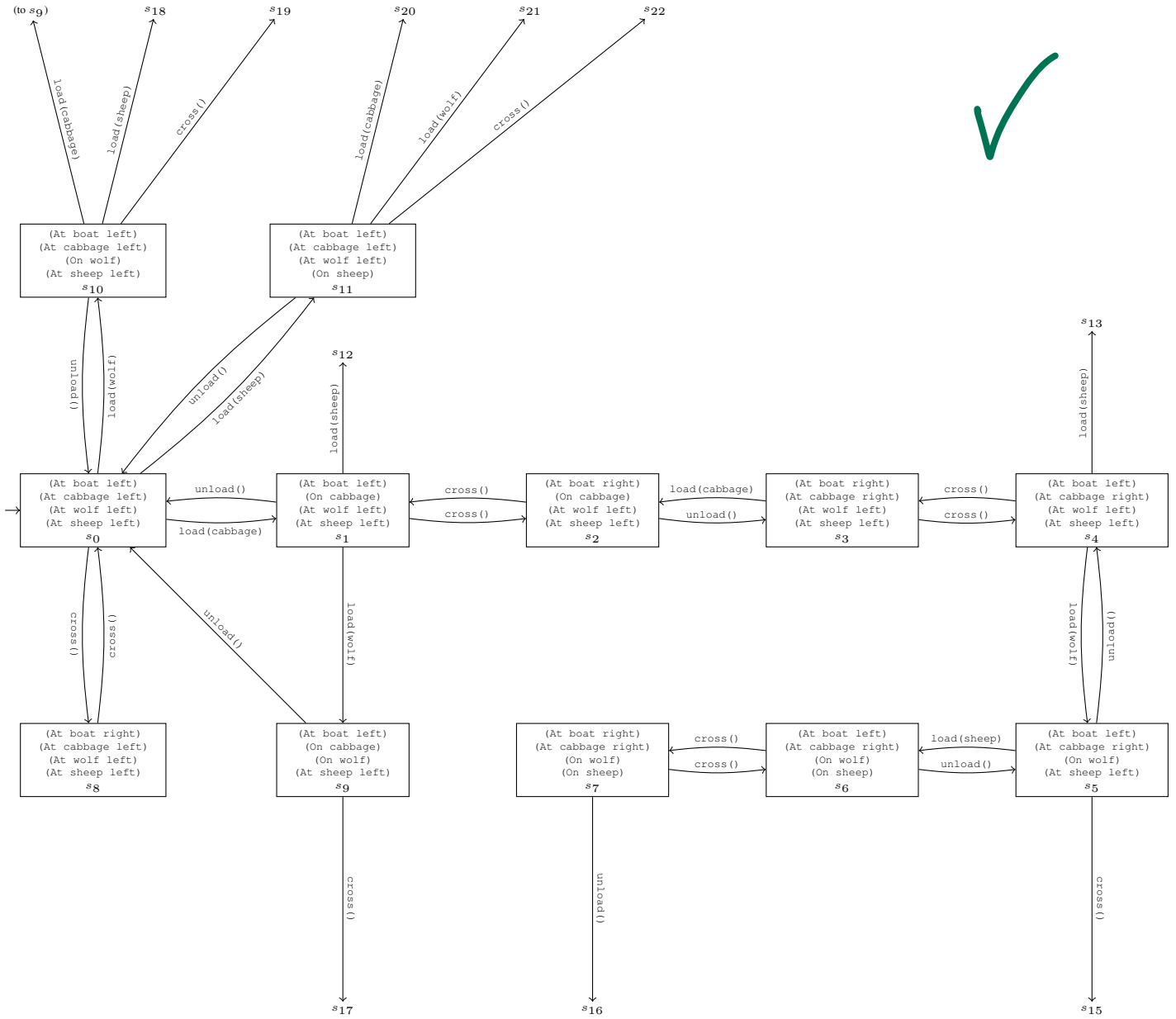
```
(define (domain river_domain)

  (:requirements :adl)
  (:types side item)
  (:constants left right - side boat cabbage wolf sheep - item)
  (:predicates
    (opposite ?x1 ?x2 - side)
    (At ?x - item ?y - side)
    (On ?x - item)
  )

  (:action load
    :parameters (?x - item)
    :precondition (and
                    (not (= ?x boat))
                    (exists (?s-side) (and (At ?x ?s) (At boat ?s)))
                    (not (exists (?y ?z - item) (and (not (= ?y ?z)) (On ?x) (On ?y))))
                  )
    :effect (and (not (exists (?s -side) (At ?x ?s))) (On ?x))
  ); end of load

  (:action cross
    :parameters ()
    :precondition ()
    :effect (forall (?x ?y - side)
                (when
                     (and (At boat ?x) (Opposite ?x ?y))
                     (and (not (At boat ?x))(At boat ?y))
                )
            )
  ); end of cross
```

```
  (:action unload
    :parameters ()
    :precondition (exists (?x-item) (On ?x))
    :effect (forall (?x-item ?s-side)
                (when
                     (and (At boat ?s) (On ?x))
                     (and (not (On ?x))(At ?x ?s))
                )
            )
  ); end of unload
); end of define domain
```

Problem file:

```
(define (problem river_problem) (:domain river_domain)
  (:objects )
  (:init (At boat left) (At cabbage left) (At wolf left) (At sheep left))
  (:goal (and
              (not (At boat left))
              (not (At cabbage left))
              (not (At wolf left))
              (not (At sheep left))
         )
  )
) ; end of define problem
```

Transition System:



DFS Algorithm (variant with marking check before pushing):

```
DFS(domain d, state init, formula goal){
  // d:  input domain;
  // init:  initial state;
  // goal:  goal formula;
  Stack t = [(init, empty)]; // Open set (stack)
  Set m = {init}; // Marked states
  while (!t.empty()) {
    (state,plan) = t.pop(); // plan is action sequence followed to reach s
    if (s ⊨ goal) return plan; // if s satifies goal state, return plan
    forall (actions a executable in s)
      s'=d.delta(s,a) // s' is successor of s under a (d.delta is transition function of d)
      if (s' ∉ m) {// if s' not marked
        m.add(s'); // mark s'
        t.push((s',plan·a)); // add s' with plan extended by a, to open set t}
  }
  return noplan; // no plan found
}
```

Current node, open set and marked states at end of every iteration of call
`DFS(shapes_domain,`s_0`, (and (not (agentAt A)) (not (agentAt B)) (not (agentAt C))):`

0. `t = [(`s_0`, empty)]`
   `m =` $\{s_0\}$
1. `(state,plan)=t.pop() // (`s_0`, empty)`
   `m.add(`s_8`)`
   `t.push((`s_8`, cross()))`
   `m.add(`s_{10}`)`
   `t.push((`s_{10}`, load(wolf)))`
   `m.add(`s_{11}`)`
   `t.push((`s_{11}`, load(sheep)))`
   `m.add(`s_1`)`
   `t.push((`s_1`, load(cabbage)))`

   Content of `t` and `m` at end of iteration:
   `t=[(`s_1`, load(cabbage))]`
   `[(`s_{11}`, load(sheep))]`
   `[(`s_{10}`, load(wolf))]`
   `[(`s_8`, cross())]`
   `m =` $\{s_0, s_1, s_8, s_{10}, s_{11}\}$
2. `(state,plan)=t.pop() // (`s_1`, load(cabbage))`
   `m.add(`s_{12}`)`
   `t.push((`s_{12}`, load(cabbage) load(sheep)))`
   `m.add(`s_9`)`
   `t.push((`s_9`, load(cabbage) load(wolf)))`
   `m.add(`s_2`)`
   `t.push((`s_2`, load(cabbage) cross()))`

   Content of `t` and `m` at end of iteration:
   `t=[(`s_2`, load(cabbage) cross())]`
   `[(`s_9`, load(cabbage) load(wolf))]`
   `[(`s_{12}`, load(cabbage) load(sheep))]`
   `[(`s_{11}`, load(sheep))]`
   `[(`s_{10}`, load(wolf))]`
   `[(`s_8`, cross())]`
   `m =` $\{s_0, s_1, s_2, s_8, s_9, s_{10}, s_{11}, s_{12}\}$
3. `(state,plan)=t.pop() // (`s_2`, load(cabbage) cross())`
   `m.add(`s_3`)`
   `t.push((`s_3`, load(cabbage) cross() unload()))`

   Content of `t` and `m` at end of iteration:
   `t=[((`s_3`, load(cabbage) cross() unload())]`
   `[(`s_9`, load(cabbage) load(wolf))]`
   `[(`s_{12}`, load(cabbage) load(sheep))]`
   `[(`s_{11}`, load(sheep))]`
   `[(`s_{10}`, load(wolf))]`
   `[(`s_8`, cross())]`
   `m =` $\{s_0, s_1, s_2, s_3, s_8, s_9, s_{10}, s_{11}, s_{12}\}$
4. `(state,plan)=t.pop() // ((`s_3`, load(cabbage) cross() unload())`
   `m.add(`s_4`)`
   `t.push((`s_4`, load(cabbage) cross() unload() cross()))`

   Content of `t` and `m` at end of iteration:
   `t=[(`s_4`, load(cabbage) cross() unload() cross())]`
   `[(`s_9`, load(cabbage) load(wolf))]`
   `[(`s_{12}`, load(cabbage) load(sheep))]`
   `[(`s_{11}`, load(sheep))]`
   `[(`s_{10}`, load(wolf))]`
   `[(`s_8`, cross())]`
   `m =` $\{s_0, s_1, s_2, s_3, s_4, s_8, s_9, s_{10}, s_{11}, s_{12}\}$

5. `(state,plan)=t.pop() // (`s_4`, load(cabbage) cross() unload() cross())`
   `m.add(`s_{13}`)`
   `t.push((`s_{13}`, load(cabbage) cross() unload() cross() load(sheep))`
   `m.add(`s_5`)`
   `t.push((`s_5`, load(cabbage) cross() unload() cross() load(wolf))`

   Content of `t` and `m` at end of iteration:
   `t=[(`s_5`, load(cabbage) cross() unload() cross() load(wolf))]`
   `[(`s_{13}`, load(cabbage) cross() unload() cross() load(sheep))]`
   `[(`s_9`, load(cabbage) load(wolf))]`
   `[(`s_{12}`, load(cabbage) load(sheep))]`
   `[(`s_{11}`, load(sheep))]`
   `[(`s_{10}`, load(wolf))]`
   `[(`s_8`, cross())]`
   `m =` $\{s_0, s_1, s_2, s_3, s_4, s_5, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}\}$

6. `(state,plan)=t.pop() // (`s_5`, load(cabbage) cross() unload() cross() load(wolf))`
   `m.add(`s_{15}`)`
   `t.push((`s_{15}`, load(cabbage) cross() unload() cross() load(wolf) cross())`
   `m.add(`s_6`)`
   `t.push((`s_6`, load(cabbage) cross() unload() cross() load(wolf) load(sheep))`

   Content of `t` and `m` at end of iteration:
   `t=[(`s_6`, load(cabbage) cross() unload() cross() load(wolf) load(sheep))]`
   `[(`s_{15}`, load(cabbage) cross() unload() cross() load(wolf) cross())]`
   `[(`s_{13}`, load(cabbage) cross() unload() cross() load(sheep))]`
   `[(`s_9`, load(cabbage) load(wolf))]`
   `[(`s_{12}`, load(cabbage) load(sheep))]`
   `[(`s_{11}`, load(sheep))]`
   `[(`s_{10}`, load(wolf))]`
   `[(`s_8`, cross())]`
   `m =` $\{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{15}\}$

7. `(state,plan)=t.pop() // (`s_6`, load(cabbage) cross() unload() cross() load(wolf) load(sheep))`
   `m.add(`s_7`)`
   `t.push((`s_7`, load(cabbage) cross() unload() cross() load(wolf) load(sheep) cross())`

   Content of `t` and `m` at end of iteration:
   `t=[(`s_7`, load(cabbage) cross() unload() cross() load(wolf) load(sheep) cross())]`
   `[(`s_{15}`, load(cabbage) cross() unload() cross() load(wolf) cross())]`
   `[(`s_{13}`, load(cabbage) cross() unload() cross() load(sheep))]`
   `[(`s_9`, load(cabbage) load(wolf))]`
   `[(`s_{12}`, load(cabbage) load(sheep))]`
   `[(`s_{11}`, load(sheep))]`
   `[(`s_{10}`, load(wolf))]`
   `[(`s_8`, cross())]`
   `m =` $\{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{15}\}$

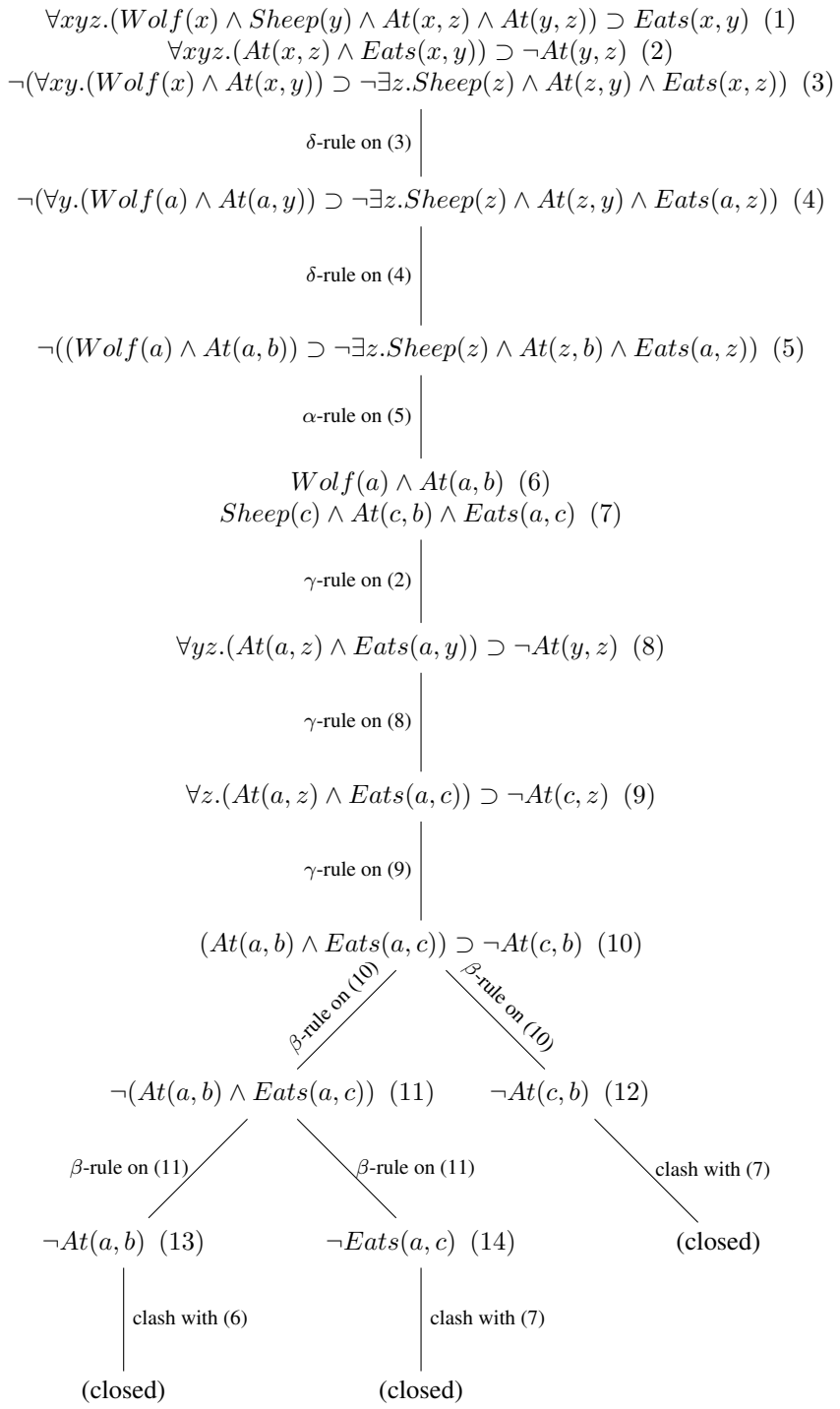
8. `(state,plan)=t.pop()`
   `// (`s_7`, load(cabbage) cross() unload() cross() load(wolf) load(sheep) cross())`
   `return plan`

Returned plan: `load(cabbage) cross() unload() cross() load(wolf) load(sheep) cross())`

Tableau:

Checking whether $\{\phi_1, \phi_2\} \models \phi_3$, is equivalent to checking whether the KB $\mathcal{K} = \{\phi_1, \phi_2, \neg\phi_3\}$ is unsatisfiable.

Let's construct the tableau for $\mathcal{K}$:



Since the tableau closes, it follows that $\{\phi_1, \phi_2\} \models \phi_3$. Observe, in particular, that ϕ_2 alone logically implies ϕ_3 (ϕ_1 is never used in the construction).