Harita Shroff
CS-156
11/21/2017

Homework-4

1.Express each of the following as first order logic formula: (a) *ε* is a binary string, (b) if x is a binary string, then so are One(x) and Zero(x), (c) the length of the *ε* is 0, (d) the length of One(x) and the length of Zero(x) is 1 plus the length of x.

a.	Binary(*ε*)
b.	∀x(Binary(x) => (Binary(One(x)) ^ Binary(Zero(x))))
c.	Length(*ε*) = 0
d.	∀x(Binary(x) => (Length(One(x)) = 1 + Length(x)) ^ (Length(Zero(x)) = 1 + Length(x)))

2.Using our Natural Deduction system extended by rules for First-order logic, assuming (a)-(d) of problem 1 as our knowledge base, give a formal proof of the formula *α* := there exists an *x* that is a binary string of length 1+1+0.

α = ∃x (Binary(x) ^ Length(x) = 1 + 1 + 0)

1. Binary(*ε*) (From KB)
2. Length(*ε*) = 0 (From KB)
3. Binary(*ε*) => Binary(One(*ε*)) ^ Binary(Zero(*ε*)) (Replacing x with *ε* in KB)
4. Binary(One(*ε*)) ^ Binary(Zero(*ε*)) (Applied Modus Ponen on 1 & 3)
5. Binary(One(*ε*)) (And Elimination)
6. Binary(*ε*) => Length(One(*ε*)) = 1 + Length(*ε*) ^ Length(Zero(*ε*)) = 1 + Length(*ε*) (Replacing x with *ε* in KB)
7. Length(One(*ε*)) = 1 + Length(*ε*) ^ Length(Zero(*ε*)) = 1 + Length(*ε*) (Applied Modus Ponen on 1 & 6)
8. Length(One(*ε*)) = 1 + 0 ^ Length(Zero(*ε*)) = 1 + 0 (From KB)
9. Length(One(*ε*)) = 1 + 0 (And Elimination)
10. Binary(One(*ε*)) => Binary(One(One(*ε*))) ^ Binary(Zero(One(*ε*))) (Replacing x with One(*ε*) in KB)
11. Binary(One(One(*ε*))) ^ Binary(Zero(One(*ε*))) (Applied Modus Ponen on 5 & 10)

12. Binary(One(One(*ε*))) (And Elimination)
13. Binary(One(*ε*)) => Length(One(One(*ε*))) = 1 + Length(One(*ε*)) ^ Length(Zero(One(*ε*))) = 1 + Length(One(*ε*))
14. Length(One(One(*ε*))) = 1 + Length(One(*ε*)) ^ Length(Zero(One(*ε*))) = 1 + Length(One(*ε*)) (Applied Modus Ponen on 5 & 13)
15. Length(One(One(*ε*))) = 1 + Length(One(*ε*)) (And Elimination)
16. Length(One(One(*ε*))) = 1 + 1 + 0 (From 8)
17. Binary(One(One(*ε*))) ^ (Length(One(One(*ε*))) = 1 + 1 + 0)
18. ∃x (Binary(x) ^ Length(x) = 1 + 1 + 0) (From #17)

3.Let the formulas of Problem 1 be our KB and *α* be as in Problem 2. Skolemize the formulas in KB and ¬*α*, convert the result to CNF, and then clauses. Finally, find a resolution refutation. For at least one place where you needed to do unification carefully show the steps the algorithm from class would use.

1. KB: Binary(*ε*)
2. KB: ∀x(Binary(x) => (Binary(One(x)) ^ Binary(Zero(x))))
3. KB: Length(*ε*) = 0
4. KB: ∀x(Binary(x) => (Length(One(x)) = 1 + Length(x)) ^ (Length(Zero(x)) = 1 + Length(x)))
5. ¬*α* : ¬(∃x (Binary(x) ^ Length(x) = 1 + 1 + 0))

Skolemization of Formulas in KB and ¬*α*

1.	Binary(*ε*)	From KB
2.	∀x(Binary(x) => (Binary(One(x)) ^ Binary(Zero(x))))	From KB
3.	Length(*ε*) = 0	From KB
4.	∀x(Binary(x) => (Length(One(x)) = 1 + Length(x)) ^ (Length(Zero(x)) = 1 + Length(x)))	From KB
5.	¬(∃x (Binary(x) ^ Length(x) = 1 + 1 + 0))	From ¬*α*
6.	∀x(¬Binary(x) V (Binary(One(x)) ^ Binary(Zero(x))))	From #2 Skolemization
7.	(¬Binary(x) V Binary(One(x))) ^ (¬Binary(x) V Binary(Zero(x)))	From #6
8.	**¬Binary(x) V Binary(One(x))**	**From #7**
9.	∀x(¬Binary(x) V (Length(One(x)) = 1 + Length(x)) ^	From #4

	(Length(Zero(x)) = 1 + Length(x)))	Skolemization
10.	(¬Binary(x) V (Length(One(x)) = 1 + Length(x))) ^ (¬Binary(x) V (Length(Zero(x)) = 1 + Length(x)))	From #9
11.	**¬Binary(x) V (Length(One(x)) = 1 + Length(x))**	**From #10**
12.	∀x(¬(Binary(x) ^ Length(x) = 1 + 1 + 0))	From #5
13.	**¬Binary(x) V ¬(Length(x) = 1 + 1 + 0)**	**From #12**

CNF Clauses:

1. Binary(*ε*) (From KB)
2. Length(*ε*) = 0 (From KB)
3. ¬Binary(x) V Binary(One(x)) (#8 in table above)
4. ¬Binary(x) V (Length(One(x)) = 1 + Length(x)) (#11 in table above)
5. ¬Binary(x) V ¬(Length(x) = 1 + 1 + 0) (#13 in table above)
6. Binary(One(*ε*)) **(From R1 and R3)**
7. Length(One(*ε*)) = 1 + Length(*ε*) **(From R1 and R4)**
8. Length(One(*ε*)) = 1 + 0 **(From R2)**
9. Binary(One(One(*ε*))) **(From R3 and R6)**
10. Length(One(One(*ε*))) = 1 + Length(One(*ε*)) **(From R4 and R6)**
11. Length(One(One(*ε*))) = 1 + 1 + 0 **(From R8)**
12. ¬(Length(One(One(*ε*))) = 1 + 1 + 0) **(From R5 and R9)**
13. {} **(From R11 and R12) (Resolution Refutation as clause is empty)**

Using Unifying Algorithm for (Binary(*ε*), Binary(x)):

1. Unify (x, y, S)
2. Unify (Binary(*ε*), Binary(x), ())
    a. x = Binary(*ε*), y = Binary(x), S = ()
3. As x and y are both terms,
    a. Return (Unify(args(x), args(y), Unify (op(x), op(y), S))
        i. Unify(*ε*, x, Unify (Binary(t), Binary(t), ()))
            1. x = *ε*, y = x, S = Unify (Binary(t), Binary(t), ())
            2. Return Unify (Binary(t), Binary(t), ())
                a. Binary(t) = Binary(t) so it will return S = ()
        ii. Unify(*ε*, x, ())
            1. return Unify-var(x, *ε*, ()) (as x is a variable)
            2. Unify-var(x, *ε*, ())
                a. All if and else if conditions will fail as mapping does not exist in S.
                b. In the else loop, return (append((x ->*ε*), S))
                c. return (x -> *ε*)
            3. return (x -> *ε*)

4.Pretend you are going all out to get dressed for a night out. Imagine all the different things you might put on or have to choose between. Model this as a PDDL problem. Then use the GraphPlan algorithm to find a solution.

Literals:

- Top(t) - A predicate for whether t is a top.
- Bottom(b) - A predicate for whether b is a bottom.
- Shoe(s) - A predicate for whether s is a shoe.
- Accessory(a) - A predicate for whether a is an accessory.
- MatchBottomTop(b, t) - A predicate for whether the selected bottom matches top
- MatchShoeTop(s, t) - A predicate for whether the selected shoe matches top
- MatchShoeBottom(s, b) - A predicate for whether the selected shoe matches bottom
- MatchAccessorytop(a, t) - A predicate for whether the selected accessory matches top
- MatchAccessoryBottom(a, b) - A predicate for whether the selected accessory matches bottom
- MatchAccessoryShoe(a, s) - A predicate for whether the selected accessory matches shoe
- SelectedTop(t) - A top was selected
- SelectedBottom(b) - A bottom was selected
- SelectedShoe(s) - A shoe was selected
- SelectedAccessory(a) - An accessory was selected
- Dressed(t, b, s, a) - A predicate for whether we are dressed matched top, bottom, shoe and accessory or not.

Actions:

- ChooseTop(t) - Select a top
- ChooseBottom(b) - select a bottom
- ChooseShoe(s) - select a shoe
- ChooseAccessory(a) - select an accessory
- PutOn(t, b, s, a) - put on selected top, bottom, shoe and accessory

Init: (Top(t) ^ Bottom(b) ^ Shoe(s) ^ Accessory(a) ^ SelectedTop(t) ^ ¬SelectedTop(t) ^ ¬SelectedBottom(b) ^ ¬SelectedShoe(s) ^ ¬SelectedAccessory(a))

Goal: (SelectedTop(t) ^ SelectedBottom(b) ^ SelectedShoe(s) ^ SelectedAccessory(a) ^ Dressed(t, b, s, a))

Action: ChooseTop(t)
PreCond: Top(t) ^ ¬SelectedTop(t)
Effect: SelectedTop(t)

Action: ChooseBottom(b)
PreCond: SelectedTop(t) ^ Bottom(b) ^ MatchBottomTop(b, t) ^ ¬SelectedBottom(b)
Effect: SelectedBottom(b)

Action: ChooseShoe(s)
PreCond: SelectedBottom(b) ^ Shoe(s) ^ MatchShoeTop(s, t) ^ MatchShoeBottom(s, b) ^ ¬SelectedShoe(s)
Effect: SelectedShoe(s)

Action: ChooseAccessory(a)
PreCond: SelectedShoe(s) ^ Accessory(a) ^ MatchAccessorytop(a, t) ^ MatchAccessoryBottom(a, b) ^ MatchAccessoryShoe(a, s) ^ ¬SelectedAccessory(a)
Effect: SelectedAccessory(a)

Action: PutOn(t, b, s, a)
PreCond: SelectedTop(t) ^ SelectedBottom(b) ^ SelectedAccessory(a) ^ SelectedShoe(s)
Effect: Dressed(t, b, s, a)

GraphAlgorithm:

```
function GraphPlan(problem) returns solution or failure:
    graph := Initial-Planning-Graph(problem)
    goals := Conjuncts(problem, Goal)
    nogoods := empty hash table
    for t= 0 to infty do:
        if goals all present and non-mutex in S_t of graph then
            solution := Extract-Solution(graph, goals, NumLevels(graph), nogoods)
            if solution != failure then return solution
        if graph  and nogoods have not changed since t-1 then return failure
        graph := Expand-Graph(graph, problem)
```

1. Express the goal as conjunct of literals

- The goal can be expressed as:
  Goal: SelectedTop(t) ^ SelectedBottom(b) ^ SelectedShoe(s) ^ SelectedAccessory(a) ^ Dressed(t, b, s, a)

2. For t=0, Check if goals are all and non-mutex in S0

- In S0, none of the goal literals are present. Expand-Graph will called to add the actions from A0. In this case, ChooseTop(t) will be the action. All other Choose actions will be updated due to mutex links.

3. For t=1, Check if goals are all and non-mutex in S1

- In S1, not all goals are present. Expand-Graph will be called to add the action from A1. At this stage, ChooseBottom(b) will be the action and all other actions with mutex links will be updated.

4. For t=2, Check if goals are all and non-mutex in S2

- In S2, not all goals are present, Expand-Graph will be called to add the action from A2. At this state, ChooseShoe(s) will be the action and all other actions with mutex links will updated.

5. For t=3, Check if goals are all and non-mutex in S3

- In S3, not all goals are present so Expand-Graph will be called to add the action from A3. At this stage, ChooseAccessory(a) will be the action and all other actions with mutex link will be updated.

6. For t=4, Check if goals are all and non-mutex in S4

- In S4, not all goals are present so Expand-Graph will be called to add the action from A4. At this stage, putOn(t, b, s, a) will be called.

7. For t=5, Call Extract-Solution

- There are two ways to extract-solution in this case. In CSP, the variables are actions at each state. All the variables have domain of either in or out. This represents whether action is included or not. The constraints are the mutex relations in this approach. So in this problem, variables will be {ChooseTop(t), ChooseBottom(b), ChooseShoe(s), ChooseAccessory(a), PutOn(t, b, s, a)}. This will solve the problems without violating any mutex relationships due to them being constraints.

5.Briefly explain how PDDL solves the frame problem. Give some disadvantages to formulating problems in PDDL.

The frame problem involves being able to calculate what stays the same or remains unchanged as the result of an action on a state. PDDL solves this problem by

“specifying the result of an action in terms of what changes; everything that stays the same is left unmentioned” (Russell and Norvig, 367). Take the following PDDL action schema, for example:

Action(Fly(p, from, to),
PRECOND: At(p, from) ^ Plane(p) ^ Airport(from) ^ Airport(to)
EFFECT: ~At(p, from) ^ At(p, to))

The EFFECT statement tells us what has changed from PRECOND. Therefore, we know that everything else that is not mentioned in EFFECT (such as Plane(p), Airport(from), and Airport(to)) remains unchanged. Although this is a significant benefit and advantage to PDDL, there are some disadvantages to formulating problems in PDDL as well. For instance, certain fluents are not allowed to be expressed in a state: non-ground fluents, negations, and those that contain function symbols. Another disadvantage is that if the problem is very large and complex, it will be difficult to keep track of all of the action schemas. For example, the air cargo transport and spare tire problem are relatively small examples with three action schemas. However, a problem with twenty action schemas may take longer and/or be more complicated to represent and solve. Similarly, another disadvantage is if there are a lot of different variables in the PDDL.