Homework 2

Artificial Intelligence For Robotics

Julian Torres

1) Problem 7.7

a) sentence is false iff B and C are false. happens in 4 cases for A and D. → 12

b)False only if A,B,C,D are false, which occurs in 1 case → 15

c) gives a model in which the first conjunct is false → 0

2)

Objects

- Vehicles: cars, trucks, motorcycles, buses, etc.
- Pedestrians: adults, children, disabled individuals
- Road Infrastructure: crosswalks, traffic lights, stop signs, lane markings
- Environment: Obstacles, weather conditions, daytime light levels

Taxonomic Tree

```
World
├── Vehicles
│   ├── Motorized
│   │   ├── Cars
│   │   ├── Trucks
│   │   ├── Motorcycles
│   │   └── Buses
│   └── Non-Motorized
│       └── Bicycles
├── Pedestrians
│   ├── Adults
│   ├── Children
│   └── Disabled
│       └── Wheelchairs
├── Road Infrastructure
│   ├── Crosswalks
│   ├── Traffic Control
│   │   ├── Traffic Lights
│   │   └── Stop Signs
│   └── Lane Markings
└── Environment
    ├── Obstacles
    ├── Weather
    │   ├── Rain
    │   ├── Fog
    │   └── Snow
    └── Light Levels
```

```
│       └── Light Levels
│           ├── Daylight
│           ├── Night
│           └── Shadows
```

Events

1. Vehicle Approaching:

- Detect vehicles approaching the crosswalk.
- Estimate the speed and trajectory of vehicles.
- Identify if vehicles are likely to stop or pose a threat to pedestrians.

---

2. Pedestrian Waiting to Cross:

- Identify pedestrians waiting at the crosswalk.
- Classify types of pedestrians (e.g., children, disabled).

---

3. Crosswalk Clearance:

- Monitor the crosswalk to ensure that it is clear of obstacles and pedestrians.
- Signal when it is safe for pedestrians to cross.

---

4. Traffic Light State Change:

- Detect changes in the state of traffic lights.
- Adjust crossing signals and timing based on the traffic light state.

---

5. Pedestrian Crossing:

- Monitor pedestrians while they are crossing the road.
- Identify any hazards or sudden changes in pedestrian behavior.

---

6. Emergency Situations:

- Detect emergency vehicles and respond accordingly (e.g., halting pedestrian crossings temporarily).
- Respond to sudden obstacles or accidents in the crosswalk area.

---

7. Weather Changes:

- Detect weather conditions that may impact visibility or safety.
- Adapt behavior based on weather (e.g., allow extra time for pedestrians to cross during rain).

3)

i. A $\wedge$ B

ii. A $\implies$ B

iii. (S $\implies$ W) $\wedge$ (W $\implies$ S)

iv. (dry $\wedge$ raining outside) $\implies$ (have umbrella $\vee$ have hoodie) $\wedge$ $\neg$ raining heavily

v. $\neg$ (student handed homework late or incomplete $\implies$ the student will not lose points)

4)

A is true if and only if it is not the case that A, B, and C are all true

5)

Translation of English Sentences into First-Order Logic (Markdown Format)

---

vi. Some students pass English but not Math.

Let:

- (S(x)): (x) is a student.
- (P_E(x)): (x) passes English.
- (P_M(x)): (x) passes Math.

$$\exists x\,(S(x) \wedge P_E(x) \wedge \neg P_M(x))$$

---

vii. Every student is registered in a class and enrolled at a university.

Let:

- (S(x)): (x) is a student.
- (R(x)): (x) is registered in a class.
- (E(x)): (x) is enrolled at a university.

$$\forall x\,(S(x) \implies (R(x) \land E(x)))$$

---

viii. If someone is an aunt or uncle, then someone must be their niece or nephew.

Let:

- (A(x)): (x) is an aunt.
- (U(x)): (x) is an uncle.
- (N(y, x)): (y) is a niece or nephew of (x).

$$\forall x\,((A(x) \lor U(x)) \implies \exists y\,N(y,x))$$

---

ix. The old that is strong does not wither.

Let:

- (O(x)): (x) is old.
- (S(x)): (x) is strong.
- (W(x)): (x) withers.

$$\forall x\,((O(x) \land S(x)) \implies \neg W(x))$$

6)

10.2

Fly(P1, JFK, SFO) Fly(P1, JFK, JFK) Fly(P2, SFO, JFK) Fly(P2, SFO, SFO)

10.9

- negative effects happen in S1, they are mutex with their positive counterparts
- **Fly** and **Load** actions are possible at A0
- planes can fly even if they are empty

7)

8.1

a)

town markers, road symbols, monuments, water, highways

b)

Explicit

1. The city of Boston is located at coordinates (42.3601, -71.0589).
2. There is a road connecting City A and City B.
3. City X is located north of City Y.

Implicit

1. City A is east of City B.
2. There is a path from City X to City Z via City Y.
3. The distance between Point P and Point Q is greater than the distance between Point P and Point R.

---

c)

1. Cultural/Historical siginificance of locations (i.e. Gettysburg)
2. Demographic Information
3. Climate

8.6

a) Valid

b) Valid

c) Valid

8)

A CSP-Based Integrated Task and Motion Planning for Assembly Robots

This document explains how to formulate and solve a task and motion planning problem for assembly robots using a Constraint Satisfaction Problem (CSP) approach, as discussed in "CSP-Based Integrated Task & Motion Planning for Assembly Robots."

---

1. Formulating Task and Motion Planning as a CSP

A CSP for assembly robots can be defined as a triple `(X, D, C)` where:

- **X**: Set of variables representing different aspects of the robot and assembly process.
- **D**: Set of domains specifying possible values for each variable
- **C**: Set of constraints that define valid relationships between variables.

Variables (X)

Key variables include:

- `loc`: Represents the configuration of each arm (such as left arm, right arm) over time.
- `slotOcc`: Represents which part occupies which slot.
- `pos`: Represents the location of each part (e.g., held by an arm or placed in a specific slot).

---

2. Domains (D)

- `loc`: All possible configurations for each arm.
- `slotOcc`: A set of possible parts or an empty value for each slot.
- `pos`: All possible locations for each part (e.g., in a slot or held by a gripper).

---

3. Constraints (C)

These constraints ensure that the task and motion planning are valid:

- **State Constraints**: Define allowable configurations of the robot to prevent collisions.
- **Initial State Constraints**: Define the starting conditions of the robot and the assembly process.
- **Goal Constraints**: Define the desired final state, such as all workpieces being assembled.
- **Dynamic Constraints**: Ensure that transitions between successive states are valid (e.g., ensuring collision-free movement).

Ex):

```
minizinc
constraint forall(c in Cells, r in Arms)(
  blocking(c, loc(r)) ↔ occupant(c) == r
);
```

---

4. Cost Function

Should be defined such that it minimizes metrics such as execution time and energy expended Ex)

```
minizinc
cost = |Arms| * k^2 + sum(t in 1..(k-1), r in Arms)(
  if loc[r][t] != loc[r][t-1] then 1 else 0 endif
);
```

where k is number of steps, loc[r][t] is location of arm r at time t,

5. Designing a CSP Solver

First, I would define the structure of a CSP with the following components:

- Variables: The entities that need to be assigned values.
- Domains: The set of possible values for each variable.
- Constraints: Rules that specify which value combinations are allowed.

I would also implement a class that holds the CSP's variables, domains, and constraints, with methods to:

- Add constraints.
- Check consistency for variable assignments.
- Perform backtracking search for a solution.

Here is partially implemented code that uses the github repo as a reference

```python
from abc import ABC, abstractmethod
from typing import Dict, List, Generic, TypeVar

V = TypeVar('V')  # Variable type
D = TypeVar('D')  # Domain type

class AbstractConstraint(Generic[V, D], ABC):
    def __init__(self, variables: List[V]) -> None:
        self.variables = variables
```

```python
    @abstractmethod
    def satisfied(self, assignment: Dict[V, D]) -> bool:
        ...
class CSP(Generic[V, D]):
    def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:
        self.variables = variables
        self.domains = domains
        self.constraints = {var: [] for var in variables}

    def add_constraint(self, constraint: AbstractConstraint[V, D]) -> None:
        for variable in constraint.variables:
            self.constraints[variable].append(constraint)

    def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
        return all(constraint.satisfied(assignment) for constraint in self.constraints[variable])

    def backtracking_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:
        if len(assignment) == len(self.variables):
            return assignment

        unassigned = [v for v in self.variables if v not in assignment]
        first = unassigned[0]

        for value in self.domains[first]:
            local_assignment = assignment.copy()
            local_assignment[first] = value
            if self.consistent(first, local_assignment):
                result = self.backtracking_search(local_assignment)
                if result is not None:
                    return result
        return None

class UniqueConstraint(AbstractConstraint):
  def __init__(self, variables):
      super().__init__(variables)

  def satisfied(self, assignment):
      values = list(assignment.values())
      return len(values) == len(set(values))
```