Yachay Tech University

Mathematical and Computational Logic

Prolog Lab 6: Prolog Tutorial & Lab: Constraint-Based Scheduling with CLPFD

1. Introduction

In this lab, you will learn how to use Constraint Logic Programming over Finite Domains (CLPFD) to solve scheduling problems in Prolog. Scheduling is a classic AI and optimization challenge — assigning tasks to time slots and resources under given constraints, such as precedence and non-overlap.

2. Objectives

- Represent scheduling problems declaratively.
- Use CLPFD to define temporal and resource constraints.
- Prevent overlapping tasks using logical constraints or disjoint1/1.
- Optimize total schedule time (makespan) using labeling with minimize.

3. Background: Scheduling and CLPFD

Constraint Logic Programming is ideal for scheduling because constraints can directly express temporal relations. In Prolog's CLPFD library, each task can have variables representing its start time, duration, and end time. Constraints ensure that dependent or concurrent tasks are ordered properly without overlaps.

Load the CLPFD module before using its predicates:

```
:- use_module(library(clpfd)).
```

4. Modeling a Simple Schedule

Each task can be represented as a tuple: Start, Duration, End.
The basic constraint is: End #= Start + Duration.

Example:
StartA in 0..10, EndA #= StartA + 3.
StartB in 0..10, EndB #= StartB + 2.

5. Defining Constraints

Common constraints include:
- **Precedence**: Task A must finish before B starts → EndA #=< StartB.
- **Non-overlap**: Tasks sharing the same resource must not overlap →
  StartA + DurA #=< StartB #\/ StartB + DurB #=< StartA.
- **Total Time**: The total project duration is represented by Makespan = max(EndA, EndB, ...).

6. Example: Two Tasks

```
:- use_module(library(clpfd)).

schedule(Starts, Ends, Makespan) :-
   Starts = [S1, S2], Ends = [E1, E2],
   Durations = [3, 2],
   E1 #= S1 + 3, E2 #= S2 + 2,
   S1 + 3 #=< S2,  % Task 1 finishes before Task 2 starts
   Makespan #= max(E1, E2),
   labeling([min(Makespan)], Starts).
```

7. Non-overlapping Tasks with disjoint1/1

SWI-Prolog's CLPFD library includes disjoint1/1 to express non-overlapping intervals easily:

```
Tasks = [task(S1, 3, E1, 1, _), task(S2, 2, E2, 1, _)],
disjoint1(Tasks).
```

This ensures that tasks using the same resource (1) do not overlap.

8. Optimization

We can minimize total schedule time using labeling/2:

```
labeling([min(Makespan)], [S1, S2, Makespan]).
```

9. Lab Exercise: Multi-Task Scheduler

In this lab, you will create a scheduler that assigns start times to a list of tasks with the following properties:

1. Each task has a duration.
2. Tasks that share a resource must not overlap.
3. Some tasks depend on others finishing first.
4. The total makespan is minimized.

Part 1 – Setup

1. Create a file named `scheduler.pl`.
2. Add the following directive: :- use_module(library(clpfd)).
3. Define your task facts as: task(Name, Duration, Resource).

Part 2 – Core Logic

1. For each task, define Start and End variables.
2. Apply the End #= Start + Duration constraint.
3. Use disjoint1/1 to ensure non-overlapping tasks.
4. Apply precedence constraints for dependent tasks.

Part 3 – Optimization

1. Compute Makespan as the maximum of all end times.
2. Use labeling([min(Makespan)], Vars) to find the optimal schedule.
3. Print task schedules and total project time.

10. Example: Three Tasks

```
tasks([task(a, 3, 1), task(b, 2, 1), task(c, 4, 2)]).

schedule(Tasks, Starts, Ends, Makespan) :-
   Tasks = [task(a,3,1), task(b,2,1), task(c,4,2)],
   Starts = [Sa, Sb, Sc], Ends = [Ea, Eb, Ec],
   Sa in 0..10, Sb in 0..10, Sc in 0..10,
   Ea #= Sa + 3, Eb #= Sb + 2, Ec #= Sc + 4,
   disjoint1([task(Sa,3,Ea,1,_), task(Sb,2,Eb,1,_)]),
   Makespan #= max(Ea, Eb, Ec),
   labeling([min(Makespan)], [Sa,Sb,Sc,Makespan]).
```

11. Deliverables

- A `.pl` file with your scheduler implementation.
- A printed schedule of tasks with their start and end times.
- A screenshot of the optimized result.

12. Summary

This lab introduced constraint-based scheduling with CLPFD. You learned to model tasks as variables, apply constraints for non-overlap and precedence, and use labeling/2 to optimize total time. This approach extends easily to complex real-world scheduling, timetabling, and resource allocation problems.