Chalmers Tekniska Högskola

CHALMERS
UNIVERSITY OF TECHNOLOGY

Constraint programming and applied optimization (EEN025)

Assignment 3

Group 10

Himanshu Sahni
sahni@chalmers.se
MPCAS

Prajwal Prashant Shetye
shetye@chalmers.se
MPSYS

We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part in other solutions.

Task 1

Problem Description

The Tower of Hanoi is a classic puzzle that consists of three rods and a number of disks of different sizes, which can be slid onto any rod. The puzzle starts with the disks in ascending order of size on one rod, with the smallest disk at the top.

The goal is to move the entire stack to another rod, obeying the following rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or an empty rod.

The problem is to determine the minimum number of moves (or time steps) needed to solve the Tower of Hanoi puzzle for a given number of disks, starting with all disks on one rod and ending with them all on another rod.

Implementation

We implemented the Tower of Hanoi problem using the Z3 solver, a powerful tool for solving logical formulas and constraints. The implementation involves defining Boolean variables and arrays to represent the state of the problem at each time step. Here's what the variables represent:

- `on[d][tw][t]`: A Boolean variable that is true if disk d is on tower tw at time step t.
- `obj[d][t]`: A Boolean variable that is true if disk d is the one being moved at time step t.
- `start[tw][t]`: A Boolean variable that is true if tower tw is the starting tower for the move at time step t.
- `to[tw][t]`: A Boolean variable that is true if tower tw is the target tower for the move at time step t.

These variables are used to model the state of the Tower of Hanoi puzzle and the movement of disks from one tower to another over a series of time steps. The constraints defined using these variables ensure a valid solution to the problem while minimizing the number of time steps required to solve it.

Constraints

The Tower of Hanoi problem is constrained by various conditions that ensure a valid solution while minimizing the number of time steps required to solve it. Here are the key constraints:

1. **Precondition I**: If a disk is on a tower at a given time, it cannot be the object being moved at that time.
2. **Precondition II**: If a disk is on a tower at a given time, it cannot be moved to another tower where it's not on top of a larger disk.
3. **Uniqueness of Start Variable**: If a disk is on a tower and is being moved, there must be only one tower as the starting tower at that time.

4. **Uniqueness of To Variable**: If a disk is being moved, there must be only one target tower at that time.

5. **Uniqueness of Obj Variable**: At any time step, only one disk can be the object being moved.

6. **Non-moving Disks**: If a disk is not the object being moved and is on a tower, it will remain on that tower in the next time step. Other towers cannot have the same disk at the next time step.

7. **Distinct Start/To**: If a tower is the starting tower, it cannot be the target tower at the same time step.

8. **Update**: If a disk is an object being moved from one tower to another, the state should be updated to reflect that the disk is now on the new tower and not on the previous one.

9. **Initial/Final State**: The initial state has all disks on the first tower, and the final state has all disks on the last tower.

These constraints collectively ensure a valid and minimal solution to the Tower of Hanoi problem by defining the movement of disks from one tower to another over a series of time steps.

Results

Number of Disks	**Time Steps**
2	3
3	7
4	15
5	31
6	63
7	127

Table 1: Number of Time Steps for Solving the Tower of Hanoi Problem

```
sat
Minimum time steps needed to solve is 7
Solution for the Tower of Hanoi with 3 discs:
Time step 0:
Tower 0: [ Disk 0 Disk 1 Disk 2 ]
Tower 1: [ ]
Tower 2: [ ]
Time step 1:
Tower 0: [ Disk 1 Disk 2 ]
Tower 1: [ ]
Tower 2: [ Disk 0 ]
Time step 2:
Tower 0: [ Disk 2 ]
Tower 1: [ Disk 1 ]
Tower 2: [ Disk 0 ]
Time step 3:
Tower 0: [ Disk 2 ]
Tower 1: [ Disk 0 Disk 1 ]
Tower 2: [ ]
Time step 4:
Tower 0: [ ]
Tower 1: [ Disk 0 Disk 1 ]
Tower 2: [ Disk 2 ]
Time step 5:
Tower 0: [ Disk 0 ]
Tower 1: [ Disk 1 ]
Tower 2: [ Disk 2 ]
Time step 6:
Tower 0: [ Disk 0 ]
Tower 1: [ ]
Tower 2: [ Disk 1 Disk 2 ]
Time step 7:
Tower 0: [ ]
Tower 1: [ ]
Tower 2: [ Disk 0 Disk 1 Disk 2 ]
```

Task 2

Moving Bricks on the same platform

We have a robotic crane that is responsible for moving bricks on a designated platform. This specific scenario focuses on the robot handling tasks on the same platform, which was previously defined by the user. To maintain consistency with the encoding used in the previous task and to address the constraints relevant to this scenario, we will define variables in a manner similar to what was done in the earlier task:

- $On(b, pos, t)$ A Boolean variable that represents the state where a brick (referred to as 'b') is located at a particular position ('pos') on the platform at a given time-step ('t').
- $Obj(b, t)$ A Boolean variable that is true if brick b is the one, moved at time-step 't'.
- $From(pos, t)$ A Boolean variable that signifies the starting position at time-step 't' when moving a brick.
- $To(pos, t)$ A Boolean variable that indicates the destination position at time-step 't' during the brick movement.

Case 1:

Every brick is unique and the operations are carried on the same platform
The constraints essential for solving this problem are as follows:

1. The variable **From** is only true for the position from which a brick is being moved at a given time step. This constraint ensures that **From** accurately reflects the starting point of each brick's movement.
2. The variable **To** is only true for the position to which a brick is being moved at a specific time step. This constraint ensures that **To** accurately reflects the destination of each brick's movement.
3. At any given time step, only one brick **b** can be moved. This constraint enforces that only one brick is selected to be moved at each time step.
4. Bricks that are not being moved will remain in the same position in the next time step. This constraint ensures that non-moving bricks do not change positions.
5. Once a brick has been moved to a new position, it will remain in that new position in the next time step. This constraint handles the continuity of the position of a moved brick.
6. A brick cannot be moved to the same position in a particular time step. The initial and final positions of a brick must be different at a specific time step.
7. Bricks start at their initial positions at time step 1 but reach their goal positions at the final time step. The user defines these initial and final positions.
8. At a given time step, two or more bricks cannot occupy the same position. Each position is uniquely occupied by one brick.
9. Initially, each brick is at its predefined initial position and cannot be found at any other location. This constraint ensures that bricks start in their designated initial positions.

Case 2:

Brick can be categorized and the operations are carried on the same platform

If we introduce classes of objects, such as objects of the same color that can be considered identical, into the model, it would require an extension to the existing model to handle these classes. The primary change would be in how we represent and track objects of the same class, ensuring that they are treated as identical for constraints related to uniqueness and movement. Furthermore, when categorizing the bricks, it's important to note that the final position of each brick can be within the overall goal area associated with a specific category of bricks. Importantly, the order or index of each brick remains unchanged during this process.

1. **Object Classes** We need to define classes of objects based on their shared attributes, such as color. Each class can be represented by a unique identifier, and objects within the same class will have the same identifier.

2. **Object Identifiers** Instead of representing each object with a unique identifier, we will represent them by their class identifier. For example, if there are three red objects, they will all have the same class identifier "red."

3. **Object Movement** When moving objects, we need to ensure that objects of the same class can be moved interchangeably. For instance, if we have two red objects, we should be able to move either one in a particular step.

4. **Constraints** Adjusting the constraints to reflect the new representation of object classes. For example, uniqueness constraints will ensure that only one object from a class is moved at a given time step, and no two objects, regardless of class, occupy the same position.

5. **Initial and Goal States** Defining the initial and goal states with respect to object classes. For instance, you may specify that all objects of the "red" class should be in a particular configuration at the goal state. The bricks are initially located at their starting positions at time-step 1. However, by the final time-step, these bricks should have reached their specified goal positions, which are determined by the user.

Case 3:

Brick can be moved using multiple robots and platforms

In the more complex scenario with multiple robots and platforms, there are several changes and additional constraints required to model the problem accurately. Here are the key modifications and constraints to consider:

Multiple Platforms and Robots

- Define sets of platforms, robots, and adjacent platforms.
- Each robot is associated with a specific platform.

Object Location

- Introduce variables to represent the location of objects, including platforms and positions on platforms.
- Extend the domain for object positions to include different platforms.

Object Transfer

- Allow objects to be moved between different platforms.
- Introduce variables to represent the transfer of objects between platforms.

Constraints

- Ensure that robots can only move objects on the platform they are associated with.
- Extend uniqueness constraints to ensure that objects or classes of objects are in unique positions across all platforms.
- Specify the rules for transferring objects from one platform to another.

The additional constraints can be expressed as follows:

Robot-Platform Association

For each robot r, specify the platform it is associated with:

$$\text{robot\_platform}[r] = p, \forall r \in \text{Robots}$$

Object Location and Transfer

Introduce variables to represent the location of objects and their transfer between platforms. Constraints must ensure that objects are moved between platforms correctly:

on_platform$[o, p, t]$: Binary variable indicating object o is on platform p at time t.

transfer[o, p_1**,**p_2**, t]**: Binary variable indicating the transfer of object **o** from platform p_1 to platform p_2 at time **t**.

Robot Object Movement

Constraints to specify that a robot can only move objects on its associated platform:

For each robot r, object o, platform p, and time t:

If robot r is associated with platform p and p is the current location of the object o, then the robot can move object o at time t.

$$robot\_platform[r] = p \land on\_platform[o, p, t] \Rightarrow can\_move[r, o, t]$$

Inter-Platform Transfer Constraints

Constraints to specify when and how objects are transferred between platforms:

- Rules for transferring objects between adjacent platforms via robots.
- Ensure that an object can only be transferred to an adjacent platform accessible by a robot.
- Rules for timing the transfer, ensuring objects do not disappear or appear magically.
- Appropriate initial and goal states for objects across platforms.

These constraints are more complex than in the previous models, reflecting the increased complexity of the scenario with multiple platforms and robots. The specific details of the constraints may vary based on the characteristics of the problem and the rules governing object transfer and robot operations.

The objective function, if applicable, should consider factors such as minimizing the number of moves, energy consumption, or other optimization goals while adhering to the defined constraints.

Case 4:

Implementing Distance as a cost function

Now, we need to build a cost function that takes into account the distance. To achieve this, we can utilize an array d with the same length as the number of robots, which will help us keep a record of the distance traveled by each robot.

Let's consider a scenario where a brick b is moved from coordinates $(x1, y1)$ to $(x2, y2)$ within an $n \times n$ grid at a specific time t_i by a robot r_j. Taking into consideration both Manhattan and Cartesian distance between the two points, the $d(r_j, t_i)$ element, which represents the movement of robot r_j, can be calculated as follows:

$$d(r_j, t_i) = d(r_j, t_i) + |x2 - x1| + |y2 - y1| + \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

This will serve as our new cost function.