MCS IDP Project

simon.marynissen@kuleuven.be

Due before Friday April 3th 23:59 on Toledo.

1 Introduction

For the IDP project of MCS you have to model three specifications

1. Light Up (`https://en.wikipedia.org/wiki/Light_Up_(puzzle)`)
2. Train ticket configuration
3. Nurikabe (`https://en.wikipedia.org/wiki/Nurikabe_(puzzle)`)

which are described further below. On Toledo you can find skeletons for all three specifications containing

- A fixed vocabulary (you cannot add anything to this)
- A student vocabulary extending the fixed vocabulary (you can/will add to this)
- An empty theory over the student vocabulary (you need to add sentences to this)
- An example `main` procedure. Additionally, for Light Up and Nurikabe, visualization procedures are given.

On Toledo, you will also find `..._structures.idp` files containing structures you can use to test your specification.

2 Practical

The output of this part of the project consists of a `.zip` file with file name `LastnameFirstname_StudentNumber` (where Lastname and Firstname are your actual names, and StudentNumber is the number starting with r or s) containing the following

1. `lightup.idp`
2. `traintickets.idp`

3. `nurikabe.idp`

4. A *very concise* report named `report.txt` in which you discuss design decisions.

   - For each predicate symbol `p` and function symbol `f` you introduce, the report should contain its intended interpretation in the following format (which we also used below):
     - `Pred(x,y)` is true if and only if ⟨some condition on x and y here⟩,
     - `Func(x,y,z)` is ⟨some description here⟩.
   - The report should also contain the time you spent on this part of the project. The expected time needed is 10 to 15 hours.

About the code:

- Make it readable!!
- You are obliged to start from the provided `.idp` files. You are **not allowed** to change the fixed vocabularies at all (not even renaming, it will break visualization). Remember that you **are allowed** (and advised) to add new symbols to the student vocabulary.
- Use well-chosen names for introduced symbols and variables.
- Make sure there are no warnings, except for the *Verifying and/or autocompleting structure* warnings and the errors on including visualization and structures files.
- Choose your ontology wisely.
- Write comments! Clearly specify the meaning of every line in your theories!
- Use good indentation.
- Use a consistent form/layout of your specification.
- Specify the type when quantifying a variable.
- Do not add redundant constraints.
- The IDP IDE supports autocompletion (CTRL+SPACE); use this to save yourself some time.

This assignment is graded using automated tests (to see whether your specification matches the rules described in this document). Some points are given for readability of the specification. However, the majority of the grades for

this project will be handed out on the exam, where we will ask you to implement extensions to one the specifications of this project and you are to build further on your preexisting specifications. This makes writing a solid, well-developed project all the more important.

You are allowed to discuss this project with other people, but are not allowed to copy any code from other students. This is not an open source project, i.e. you are also not allowed to put your code openly available on the web. If you want to use git, do not use public GitHub repositories (we will find them).

For any questions, do not hesitate to contact one of the assistants.

2.1 Visualizations

For Light Up and Nurikabe, there are visualizations provided. In order to use them, you need to use the idp-ide (cfr. exercise sessions). Visualization can sometimes be used a tool for debugging purposes. The visualization style is explained below.

3 Descriptions of problems

3.1 Light Up

Description Light Up is played on a rectangular grid of white and black cells, where black cells denote walls, and white cells are 'empty'. The player places lamps in empty cells (so not on walls) with the condition that no two lamps shine on each other. A lamps shines by vertical or horizontal rays, illuminating its entire row and column unless its light is blocked by a wall. The goal of the game is to illuminate all white cells. A wall may have a number from 0 to 4 on it, indicating how many lamps must be placed adjacent to its four sides. For example a wall with a 3 must have exactly three lamps around it, while a wall with a 0 cannot have a bulb next to any of its sides. An unnumbered wall may have any number of lamps adjacent to it. Lamps placed diagonally adjacent to a numbered wall do not contribute to the lamp count. Of course, numbers only occur on walls and a wall does not have multiple numbers on it.

Vocabulary The supplied fixed vocabulary is as follows:

```
vocabulary V_lightup {
  type X isa int
  type Y isa int
  type Number isa int
  type Pos constructed from { P(X, Y) }
  wall(Pos)
  lamp(Pos)
```

```
    lit(Pos)
    tip(Pos, Number)
}
```

with the following intended meaning

X is the set of x-coordinates.

Y is the set of y-coordinates.

Number is the set of numbers (that appear on walls).

Pos is the set of positions (tuples of x and y coordinates denoted $P(x, y)$).

wall(p) holds iff at position p there is a wall.

lamp(p) holds iff at position p there is a wall.

lit(p) holds iff position p is lit.

tip(p, n) holds iff at position p there is a number n stating the number of lamps around it.

Visualization Walls are shown by black squares. Non-lit non-wall cell by white squares. Lit non-wall cells by yellow squares. Lit walls squares are shown by red squares (indicating an error). A lamp in a non-wall cell is indicated by a black 'L' and a lamp in a wall cell is indicated by a red 'E' (indicating an error).

3.2 Train Tickets

Description We are given a network of stations and tracks as a graph. Each station is a node in the graph and an edge between two nodes denotes a track (tracks have direction). You can see an example in Figure 1.

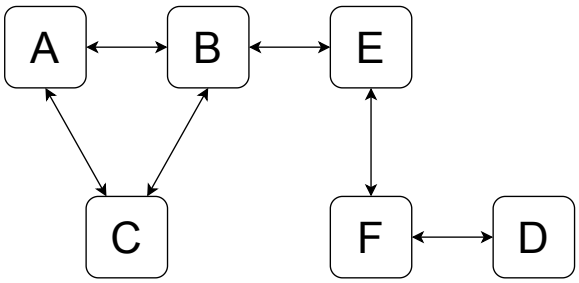

Figure 1: Example of a station and track network

Trains ride on this track. Trains are given by consecutive stations it passes and by consecutive stops it makes. Each stop is also a pass. Each stop

(respectively pass) is given by a number to indicate which stop (respectively pass) it is. For example if we have a train $IC1$ that starts in A, then drives further to pass B, but not stop at B, then continues to stop at E, then the numbers are as follows:

Station	pass	stop
A	0	0
B	1	
E	2	1

Figure 2: Passes and stops for $IC1$

For each train if it passes stations s_1 and s_2 at pass number n respectively $n+1$, then there should be a track from s_1 to s_2. If a train has an nth stop, then it also has a mth stop for each $0 \leq m \leq n$. If a train has an nth pass, then it also has a mth pass for each $0 \leq m \leq n$.

Tickets for trains are given by four things:

1. A ticket type, one of Standard, Youth (age should be less than 26), and Senior (age should be above 65).
2. The identifier of a specific train[1]
3. The station where you hop on the train.
4. The station where you hop off the train.

The prize of a ticket is determined as follows. For a Youth ticket, the prize is fixed at 5. For a Senior ticket, the prize is fixed at 6. For a Standard ticket, the prize is 3 times the number of tracks you ride on. For example, on the train $IC1$, going from A to E will cost 6.

Suppose now a person wants to travel from a given station s_1 to another given station s_2. The problem is calculate a set of tickets such he can validly travel from s_1 to s_2 and the set does not contain two different tickets for the same train.

In the IDP file, there is one structure given and in Figure 3 you can see how it is visualized.

[1]In Belgium this is not customary, but for example in the UK this type of tickets exist.

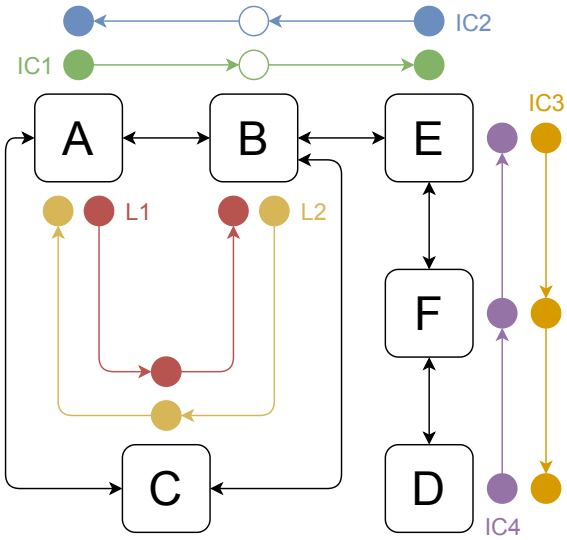

Figure 3: Provided structure visualized. Open circle are passes, but not stops. Filled circles are stops. Black arrows denote tracks.

Vocabulary The supplied fixed vocabulary is as follows:

```
vocabulary V_tickets {
  type Station
  type Train
  type Pass isa int
  type Stop isa int
  type Prize isa int
  type Age isa int
  type TicketType constructed from
    {Standard, Youth, Senior}
  type Ticket constructed from {
    T(TicketType, Train, Station, Station) }
  track(Station, Station)
  passes(Train, Station, Pass)
  stopsAt(Train, Station, Stop)
  hasTicket(Ticket)
  paid(Ticket, Prize)
  start : Station
  end : Station
  age : Age
}
```

with the following intended meaning

Station is the set of stations.

Train is the set of train identifiers, such as $IC1$.

Pass is the set of numbers that are the pass number of a train in a station, see Figure 2.

Stop is the set of numbers that are the stop number of a train in a station, see Figure 2.

Prize is the set of number that can be a prize of a train ticket.

Age is the set of possible ages.

TicketType is the set consisting of Standard, Youth and Senior.

Ticket is the set of possible tickets, where $T(t, tr, s, e)$ is the ticket of type t for train tr to hop on at s and to hop off at e.

track(s,t) holds iff there is a track from s to t.

passes(t, s, n) holds iff s is the nth pass of the train t.

stopsAt(t, s, n) holds iff s is the nth stop of the train t.

hasTicket(t) holds iff the person has ticket t.

paid(t, p) holds iff the person has ticket t and the prize of t is p.

start is the station to start travelling from.

end is the station to travel to.

age is the age of the person that wants to travel from start to end.

3.3 Nurikabe

Description Nurikabe is a Japanese logic puzzle played on a rectangular grid of cells, some of which contain numbers. The goal of the puzzle is to colour each cell blue (water) or beige (land). Two same-colored cells are considered connected if they are adjacent horizontally or vertically, but not diagonally. Connected land cells form islands, while connected water cells form seas. Each numbered cell is a land cell, the number in it is the number of cells in that island. Moreover, each island must contain exactly one numbered cell. All water cells are connected to form a single sea, with the restriction that the sea cannot contain 2x2 areas of water.

Vocabulary The supplied fixed vocabulary is as follows:

```
vocabulary V_nurikabe {
  type X isa int
  type Y isa int
  type Number isa int
  type Pos constructed from { P(X, Y) }
```

```
    land(Pos)
    water(Pos)
    islandSize(Pos , Number)
}
```

with the following intended meaning

X is the set of x-coordinates.

Y is the set of y-coordinates.

Number is the set of numbers (that appear on cells).

Pos is the set of positions (tuples of x and y coordinates denoted $P(x, y)$).

land(p) holds iff at position p there is land.

water(p) holds iff at position p there is water.

islandSize(p, n) holds iff at position p there is a number n indicating that this island has n cells of land.

Visualization Water tiles are visualized by blue squares, land tiles by beige squares. If a cell is both water and land, it will be shown by a red square. If a tile is not land and not water, it will be shown by a grey square.

Good luck!