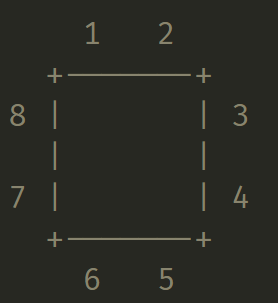
# **Tsuro**

# Tsuro | A battle of flying dragons. | Louis Oliveira | FlickrProject Summary

The idea behind the game of Tsuro is that you place tiles (from your hand of 3 tiles) onto a cell in front of your piece. Then you must follow the path in front of you, until you can follow no further. Also, any other piece in front of your tile must move as well.

There are multiple players, and the last one standing, wins. We will model the idea of what happens when you place a tile, and determine which pieces remain and where they end up on the board.

# Propositions



Location and tile propositions will  
  
both have 8 edge locations associated  
  
with them (pictured on the right).

These are the propositions used in the encoding:

* TileConnection(t,x,y): says that edge point x is connected to edge point y, on tile t. There are 7 possible target points for every individual entry point.
* LocationConnection(loc,x,y): there is a connection between x and y edge points on location l
* CrossLocationConnection(loc1, loc2, x, y): edge point x on loc1 is touching edge point y on loc2
* Location(t, loc): tile t is at location loc. The location is where on the board that things exist.
* Reachable(loc, x, k): edge point x on location loc is reachable in k hops

# Constraints

There are three core types of constraints for the model: those primarily involving Tiles, Locations, and Reachability propositions.

## Tiles

* A tile is placed in exactly one configuration somewhere on the board:  
  For every orientation of a tile (e.g. t in {t1N, t1E, t1S, t1W}) and location loc, exactly one of the Location(t, loc) propositions are true. The “exactly one” constraint was implemented by Bauhaus using a disjunction and a number of binary constraints saying any pair couldn’t both be true.
* For any given location, at most one tile is placed there:  
  For every pair of unique tiles t1 and t2, we have
* Tiles we have, have their connections set: these were just unit constraints that added TileConnection(t, x, y) as a constraint when x was connected to y for tile t in our example. All others are manually set to false.
* Tile connections are symmetric:  
  For every tile t and pair of edges x and y,
* Tile connections are to exactly one other  
  An “exactly one” constraint was used for all of the edgepoints y, given a tile t and edge point x:
* Make sure no self-loops are allowed for tiles:  
  For every tile t and edge point x, we have

## Locations

* Location connections are symmetric  
   *Similar to the Tile constraint above*
* For every location and edge on it, there is at most one connection:  
  For a location loc and 3 unique edge points x,y,z, we have
* If a tile is placed at a location, then the tile connections force the location connections to be the same. I.e., for every location loc, tile t, and pair of edge points x,y, we have
* If there is no tile at a location, then there are no connections on that location  
  The first part is a big disjunction of all the Location(t,loc) propositions (given a location loc, and disjoined over various tiles t). Let’s call this SomeTilePlaced(loc). Further, let’s use SomeConnection(loc) to be the disjunction over all LocationConnection(loc,x,y) propositions, given the location loc. Then the constraint would be:
* Make sure no self-loops are allowed for locations  
   *Similar to the Tile constraint above*
* Neighbouring locations have their matching edges connected
* Create all the other location/edge pairs to say that they are /not/ connected  
    
  These two types of constraints were just several unit clauses (propositions that must be true or false) to connect up neighbouring locations with the CrossLocationConnection propositions. This figure shows an example of two tiles side by side (connections between 3-8 and 4-7):

A black background with white numbers and symbols

Description automatically generated

## Hops

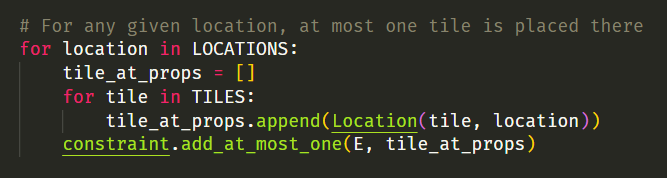
* You can get to the starting spot in 0 hops, but nowhere else in 0 hops: this is simple a number of unit clauses that fixes the starting location/edge point loc/x to be reachable in 0 (i.e., Reachable(loc, x, 0)), and all others (y) not be reachable in 0 (i.e., Reachable(loc, y, 0)).
* If you can get to a location in k hops, then you can get to a neighbour of it in k+1 hops
* If you set Reachable(l, e, k) to be true, then there must be something connected to it that is reachable in k-1 hops  
    
  Both of these constraints required careful quantification over certain objects, and so the constraint is left to the first-order logic section.

# Model Exploration

## Fixing the Bug With More Tiles Than Locations

When I started adding more tile possibilities, I found that 5 tiles happened to be placed on only four locations (2x2 grid). This *shouldn’t* be possible. This came about with 5 tiles, and just 4 locations, and looking at the final solution, I found that two separate tiles were assigned to the same location.

This immediately raised the fact that I was missing a constraint saying that at most one tile appears at a location. I fixed this by adding a constraint that looks like:



So for every location, we have this constraint added that says “at most one tile can be there”. Originally, I thought this might have to be “exactly one”, but then decided that I want to allow for some locations to have no tile placed.

After adding this constraint, I confirmed that having 5 tiles and 4 locations was unsolvable.

## Ensuring no Location Connections When No Tile

We wanted to make sure that locations don’t require a tile to be there. But leaving the connections on a location be free to be true/false, means that we might have phantom connections created. To test this theory out, I created an example that I would hope leads to no solution:

## A screen shot of a computer code Description automatically generated

Note that the 3 tiles in the setup are placed at 11, 12, and 21, and that we’re trying to have a location connection on the 22 location. This happened to be solvable, because there was nothing forcing these propositions to be false when no tile was put down. The way I resolved this was to use the following constraints:

A screen shot of a computer program

Description automatically generated

This was also the first time that I had to start using the And and Or functionality. After this constraint, the test case led to an unsolvable theory, just as I had hoped.

## Bug With Viz

For this one, I had an issue with some of solutions being visualized. In the end, it turned out to be a problem with the visualization code rather than the model, but the exploration to figure that out still required going through both:

1. For some reason, this created the following edges: (1,6) (2,7) (3,8) (6,7)
2. I confirmed these are the tiles created for t3  
   t3N: [(1, 8), (2, 5), (3, 6), (4, 7)]  
   t3E: [(3, 2), (4, 7), (5, 8), (6, 1)]  
   t3S: [(5, 4), (6, 1), (7, 2), (8, 3)]  
   t3W: [(7, 6), (8, 3), (1, 4), (2, 5)]
3. So for t3S, the 5,4 is turned into 6,7
4. Inspecting the propositions in the solution, we have:  
   (l11: 1 -> 6)  
   (l11: 2 -> 7)  
   (l11: 3 -> 8)  
   (l11: 4 -> 5)  
   (l11: 5 -> 4)  
   (l11: 6 -> 1)  
   (l11: 7 -> 2)  
   (l11: 8 -> 3)
5. So it looks like it's just the viz that is getting it wrong. Testing the various draw methods, it doesn't look like the problem is there. All of these rendered correctly:  
   draw\_tile([(1,2), (3,4), (5,6), (7,8)])[0].show()  
   draw\_tile([(2,3), (4,5), (6,7), (8,1)])[0].show()  
   draw\_tile([(1,4), (2,7), (3,6), (5,8)])[0].show()  
   draw\_tile([(1,3), (1,7), (2,4), (2,8), (3,5), (4,6), (5,7), (6,8)])[0].show()
6. Drilled down into trying the links one by one
7. Eventually found it was a bad function some draw methods were swapped), and found with this:  
   draw\_tile([(4,1), (4,2), (4,3), (4,5), (4,6), (4,7), (4,8)])[0].show()
8. **Solution** was to swap the implementation of draw45 and draw67.

This was the final output of testing the draw functions sourced from every individual edge position.

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

## Long Path Forced

Here, I wanted to see if I could create a set of tiles, and required reachability distances, so that it had the longest possible path on the 2x2 grid. This is a path that goes around the outside of the tiles and then loops back around to return to the same edge as where it began. The solution looks like the image on the right.

This required two reachability propositions to be set:

E.add\_constraint(Reachable('l11', 2, 17))  
E.add\_constraint(Reachable('l11', 3, 8))

# Jape Proofs

## A screenshot of a computer Description automatically generatedForced Configuration of Tile

*If a tile is not oriented NSE, then it must be W.*

Following assumptions are used:

* Not facing NSE:  
   ¬PN, ¬PS, ¬PE
* Must face one of the directions:  
   PN ∨ PS ∨ PE ∨ PW

Want to prove PW holds. The full Jape proof:

## Forced Connection Given Other Connections

*For a single tile, if there are links 1-2, 3-4, and not 5-6, then 5 must be connected to 7 or 8.*

The following assumptions are placed into Jape to set things up.

* Existing propositions:  
   P12, P21, P34, P43, ¬P56
* 5 is connected to at least one of 1-8 (not including 5 itself):  
   P51 ∨ P52 ∨ P53 ∨ P54 ∨ P56 ∨ P57 ∨ P58
* If there is a connection, 5 can’t be connected to it:  
   P12→¬P51, P21→¬P52, P34→¬P53, P43→¬P54

What we want to prove: P57 ∨ P58

The complete Jape Proof:

A screenshot of a computer

Description automatically generated

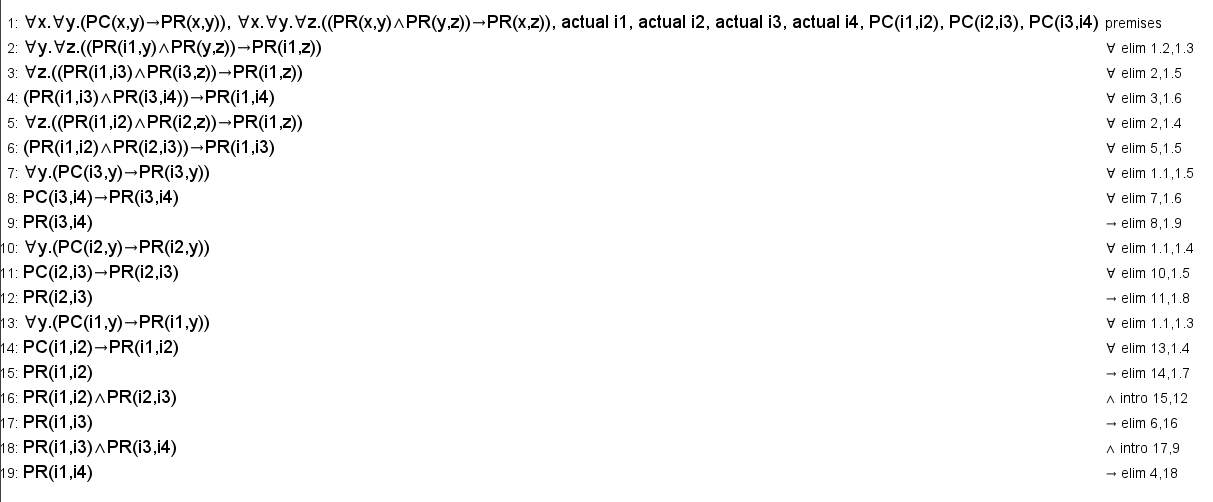
## Application of Repeated Reachability

*If we can reach l3 in 3 hops, then it is reachable*

* If two locations are connected (PC), then you can reach (PR) the destination from the source:  
   ∀x.∀y.(PC(x,y)→PR(x,y))
* If you can reach lj from li and lk from lj, then you can reach lk from li:  
   ∀x.∀y.∀z.((PR(x,y)∧PR(y,z))→PR(x,z))
* Set up the initial configuration:  
   actual i1, actual i2, actual i3, actual i4, PC(i1,i2), PC(i2, i3), PC(i3,i4)

What we want to prove: PR(i1, i4)

The full proof in Jape:



# First-Order Extension

To start with, we’ll define some of the predicates that we might have, following the propositions above:

* **TileConnection(t, e1, e2)**: The tile **t** has a connection between edges **e1** and **e2**.
* **LocationConnection(l, e1, e2)**: Similarly, a location **l** has a connection between edges **e1** and **e2**.
* **CrossLocationConnection(l1, l2, e1, e2)**: The edge **e1** at location **l1** is touching edge **e2** at location **l2**.
* **Placed(t, l)**: Tile **t** is placed at location **l**.
* **Reachable(l, e, k)**: The edge **e** at location **l** is reachable in **k** hops.

In addition, we will also need some new predicates to help with the formulae:

* **Used(t)**: Tile **t** is used on the board somewhere.
* **Orientations(t1, t2)**: Tile **t1** is the same physical tile as **t2** (i.e., they are just different orientations).
* **(\* = \*’)**: Just a notion of equality for comparing two objects. Similarly defined for

We will also need predicates that let us talk about the type of an object:

* **Tile(x)**: **x** is a tile
* **Loc(x)**: **x** is a location
* **Edge(x)**: **x** is an edge
* **Num(x)**: **x** is a number

Finally, we will assume that we have a single function allowing us to do addition. Specifically, (k+1) will be used to refer to the object one higher than the object k. Similarly for (k-1).

In what remains, we go through some of the constraints that can now be represented in first-order logic.

## Tile is Placed Once

A tile is placed in exactly one configuration on the board -- best to break this down into multiple formulae: (1) every tile is used in at least one configuration; (2) every tile is used in at most one configuration; and (2) if a tile is used, it’s placed on the board somewhere.

* {1}
* {2}
* {3}

The final constraint would just be

In contrast, this is what the Bauhaus implementation looked like to capture this constraint:

A screen shot of a computer code

Description automatically generated

## Connections are symmetric

This constraint would be the same for tile and location constraints. As an example, this is the tile version

## No self-loops

For both the tile connections and location connections, we don’t want any self-loops. An example in the tile case:

## If reachable in k, then all outgoing neighbours in k+1

If we can get to a tile’s edge point in k steps, then we should be able to get to everything it’s connected to in k+1 steps. First of all, we will use Connection(l1, l2, e1, e2) to be the proposition representing a direct connection between edge point e1 at location l1 to edge point e2 at location l2:

With this, we can now specify the full constraint:

Note that for simplicity, I didn’t include the obvious typing of the objects (i.e., that l1 and l2 are Locations, etc). This is also the case with the next constraint.

## If reachable in k, then there exists incoming neighbour at k-1

This constraint uses the same notation as the previous, but is subtly different in its final form.

A red line in a circle

Description automatically generated