# **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

TODO

* Bring in the figure for the edge labels
* Make sure they are numbers and not letters
* Finish rewriting to mirror the names used in the code

Draft:

* 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. E.g., in the figure to the top right for tile T1, we could have edge\_connected(H,C,T1) – this is shown with the red line
* configuration(t, o): tile t is oriented in direction o. The orientation can be NSEW, and we will assume that A,B endpoints are on the N side (again, see the figure above)
* location(t, loc): tile t is at location loc. The location is where on the board that things exist.
* at(p, t, ep): piece p is on tile t, at the edge point ep (e.g., A, B, …).
* can\_reach(p, t, ep): given the path in front of it, piece p can reach tile t’s endpoint ep
* connected(t1, ep1, t2, ep2): there is a direct connection between tile t1’s edge point ep1 and tile t2’s edge point ep2

# Constraints

TODO:

* Go over the proposal constraints and refine for what’s currently done
* Add the new constraints that came into the project since then
  + Just summarize the super simple ones (e.g., no self loops)

Draft:

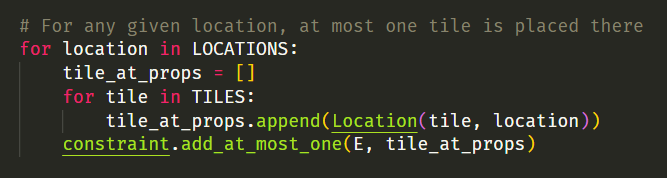
* No tile can be in two locations at the same time. For every tile and pair of unique locations loc1 and loc2, we have,  
   ¬ (location(t, loc1) /\ location(t, loc2))
* No pair of tiles can be at the same location. For every tiles t1 and t2 and location loc, we have,  
   ¬ location(t1, loc) \/ ¬ location(t2, loc)
* If things are connected, then reachability transitively holds. For example,  
  (edge\_connected(x,y,t) /\ can\_reach(p, t, x))  can\_reach(p, t, y)
* I am not yet sure how to force can\_reach(p, t, y) to be false, when there is no way to actually reach it. Halp!!!
* If there is a connection in a tile, then the edge points are connected: edge\_connected(x,y,t)  connected(t, x, t, y)
* Connect neighbouring edge points for tiles next to each other. For loc1 and loc2 that are next to each other,  
  “right/top of loc1” /\ “left/top of loc2”  connected(t1, “right/top”, t2, “left/top”)
* “right/top of loc1” needs to be a logical formula that captures the edge point of the tile that is located at loc1, and needs to take the orientation into account. Not yet sure how to best do this.

# 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

Coming soon…

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

## Forced 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:

A screenshot of a computer

Description automatically generated

## 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

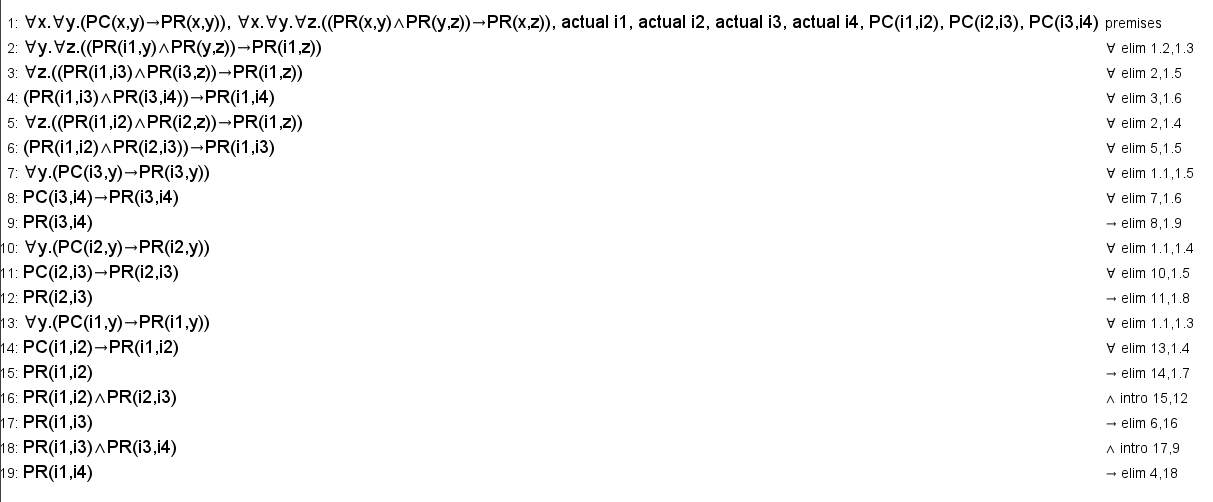
## Application of Repeated Reachability

*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

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

## Locations have at most one tile

TODO

## Connections are symmetric

TODO

## No self-loops

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

## Only starting in 0 hops

TODO

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

TODO

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

TODO

# Useful Notation