# Assignment 4 - Version 1.0

# Domain-Independent Planning

## *Domain Modelling*

**Worth:** *6%*
**Due:** *Monday 5th October Noon*
**Late Penalty:** If you submit your assignment after the due date but before two (2) days after the due date, then it will have 20% of its marks deducted. Assignments will not be accepted after that.

## *Important!*

The work done on this assignment *must be your own work*. Under no circumstances should you work together with another student on any part that ***you or they write*** for the assignment. Copying from online sources is also *not* allowed. You can *borrow ideas* as long as you cite the source of those ideas in your assignment files.

*It is allowable and even encouraged to discuss, on the class forum, exactly what is being expected for this assignment and general ideas on how to proceed.*

## *Goal of Assignment:*

In domain-independent planning, instead of writing code that computes the *neighbours* or the *is_goal* relationships, one simply describes the goal and the operators that represent the actions that can be taken in that domain. In this assignment, you will be given a planner and the problems, and you will write the domain operators and the ontology. Your domain description must follow the constraints laid out in the Domain Modelling lectures in week 7. *N.B.: In this document both "action schema" and "operator description" mean the same thing.*

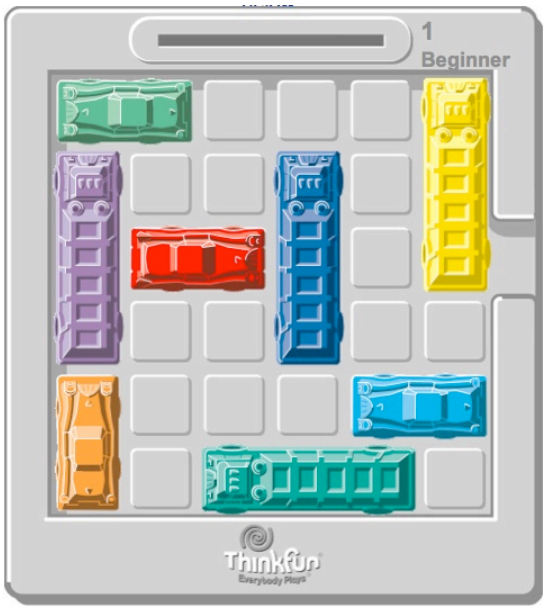
## *Introduction*

The domain will be a variation of RushHour® (an example of a RushHour® puzzle is shown below). We have extended its rules. Our rules are:

1. The traffic grid can be any size.
2. Vehicles can only move one square in the direction they are facing.
3. Vehicle face the direction shown by going from their trunk towards their hood.
4. Vehicles can reverse their direction by swapping the locations of their hood and trunk.
5. The goal is specified by the problem (i.e., it can vary from problem to problem).
6. No vehicle can move into a square already occupied by a vehicle.

7. A vehicle encompasses all the squares between its hood and its trunk inclusively.
8. No vehicle can move outside the grid.

The specific problem will be determined by the specific initial state and the goal description. Unlike the original RushHour® puzzle which always had the same goal (to get the red card to exit the grid through the opening on the right side of the grid), the rushHour problems given to your planner can have any valid goal!!! We will see examples of possible goals in the next paragraph.




In our rushHour domain, there are locations and there are vehicles. While the rushHour layout is a rectangular grid, the location in both *atHood/2* and *atTrunk/*2 is specified as a single number. (It is specified as a single number so that you will actually want to create additional predicates.) The problem specification will specify in the initial state the width and the height of the rushHour grid, for example in the picture above we would have both *width(6)* and *height(6)* in the initial state. The problem specification will also state the goal. For example, a problem may specify the goal as [*hoodAt(redCar, 17), trunkAt(greenTruck, 35)*], which specifies the location of the hoods of *redCar* and of *greenTruck*.

While the RushHour® puzzle only had 6 x 6 grids, rushHour problems can have grids of any size and your domain encoding MUST handle any rectangular grid! Essentially the location views the each row in the grid as additional segments of a line. The first row is the first line segment, the 2nd row is the 2nd segment and so on. Each segment is the width of the grid, and there are obviously "height" many segments. So, if the width is 6 and the height is 6 there will be 36 possible locations (0 through 35). Given a location L and a grid width of W, location L represents the (L // W) row and (L mod W) column in the grid (for both we assume we start numbering from zero).

There are vehicles of various sizes and various orientations. Each vehicle has a unique id and the only information given about a vehicle is the location of its hood (the front of the car) and of its trunk (the back of the car). For example, given the *blueTruck* in the above figure, we would specify *hoodAt(blueTruck, 9)* and *trunkAt(blueTruck, 21)*. Assuming a 6 x 6 grid, this would tell us that the front of the truck was located in row 1 column 3 and the back was located in row 3 column 3. All vehicles are exactly 1 grid square wide and are between 2 to 3 grid squares long.

There are four primitive predicates: *hoodAt/2, trunkAt/2, width/1,* and *height/1*. These will be used to define a problem's initial state and its goal.

There are only two actions: *move* and *reverse*. The *move* action can only move a vehicle one grid square in the direction it is facing. In our example above, the *blueTruck* is facing upwards and can only move up, at least until we execute an action to reverse its direction. Additionally, a vehicle is NOT allowed to "move" into a square already occupied by another vehicle!!! In our example above, the *redCar* cannot move forward because the middle of the truck is blocking it.

The *reverse* action can only swap the locations of the front and back of the vehicle, thus if *blueTruck,* in the figure above, were "reversed" it would then be facing downwards and could then "move" in that direction. A vehicle can always be reversed. A vehicle cannot move outside of the specified grid, i.e., it can never go left when its hood is in the $0^{th}$ column or go right when its head is in the (width – 1) column, never go up when its hood is in the $0^{th}$ row or go down when its head is in the (height – 1) row. So, for example, the yellow truck in the above figure cannot move.

The effects of the 2 actions are expressed using the primitive predicates described above (*hoodAt/2,* and *trunkAt/2*). The preconditions will be where all the fun is. While it may be possible to correctly describe the preconditions for *reverse* using the primitive predicates, you should **not** do that for the preconditions of *move*. To get full marks for this assignment, you need to create the ontology for this domain, which includes ensuring that there are static, fluent, derived, object-level, and meta-level predicates. Not only that but you will need to make "correct" preconditions and effects for the operators. By correct I mean no operator can cause anything that has been forbidden above or that obviously contravenes the rules of the "game".

## *What you will need to do*

In this assignment you will be:

1. Creating the rushHour action schemas.
2. Creating the rushHour domain ontology which supports your action schemas.

## *rushHour Domain Ontology*

The basic description of the ontology can be found in the "Introduction" section above.

For every derived and/or metaLevel predicate you will need to add a separate Prolog directive ":- dynamic *name*/*arity*.", where *name* is the name of the predicate and *arity* is

the number of arguments of that predicate. So, if you define a *smaller/2* derived predicate and a *lessThan/2* metaLevel predicate you would need to add both *":- dynamic lessThan/2."* and *":- dynamic smaller/2."*.

## Calling the Planner

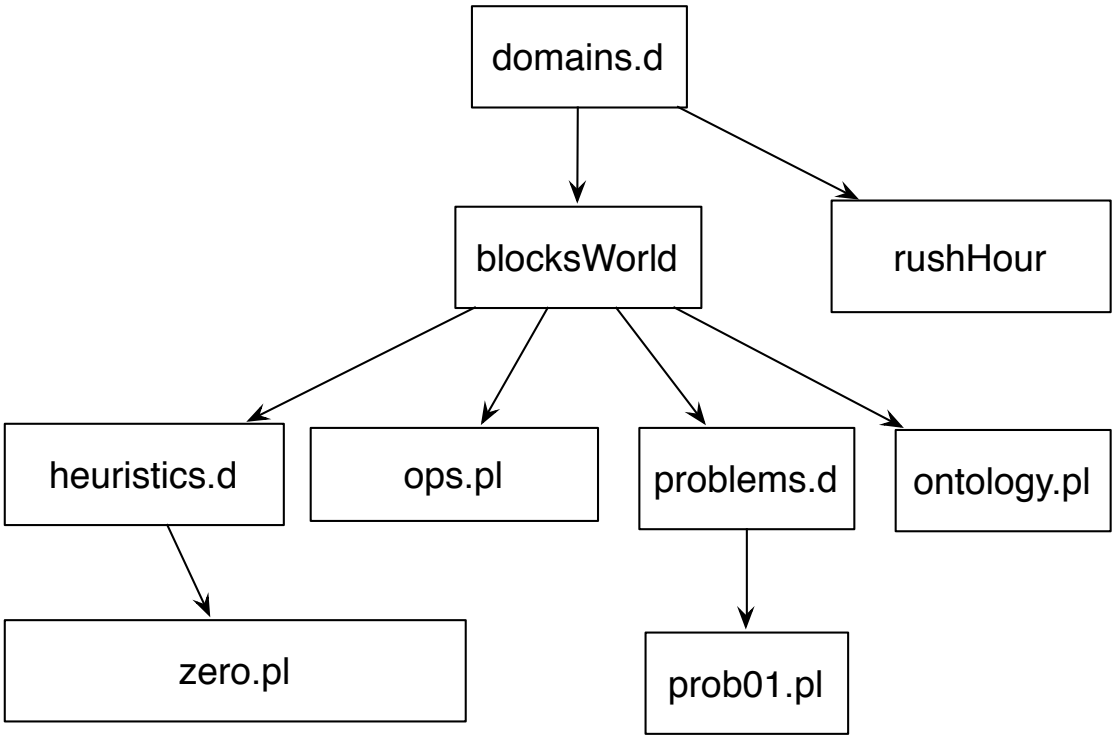
To call the planner to solve a problem:

1. Go to the directory containing ***idaStar.pl***
2. Execute SWI Prolog
3. Load ***solve.pl***
4. Enter ***solve(Domain, Problem, Solution),*** where ***Domain*** is the name of the domain directory, ***Problem*** is the name of problem file, and ***Solution*** is the unbound variable where the solution will be returned.
   For example, "***solve(blocksWorld, prob001, Solution).***".

Look at trace.txt to see an example session.

## Directory Structure

For the above example, the directory structure would be:




## What You Will Be Given

You will be given the planner (idaStar.pl, etc.). You will also be given the domains.d directory structure show above. You should look through the blocksWorld domain,

particularly ops.pl and ontology.pl for examples of how to write such files for the rushHour domain.

There may also be some files in the rushHour directory, and if there are the files ontology.pl and ops.pl, you will need to extend them.

## *Action Schemas File: ops.pl*

Your action schemas must be stored in the file *ops.pl* in the *rushHour* domain directory. The action schemas must follow the following guidelines:

- The action schemas are facts asserted into the knowledge base.
- An action schema is a data structure with the following layout:
  ***op(OpName, Parameters, Preconditions, Effects)***

For example, in the blocksWorld domain, the ***move/3*** operator schema is written as:

```
op(move,
  %% move B1 off of B2 and onto B3
  [B1, B2, B3],
  [cube(B1), cube(B2), cube(B3),
   neq(B1, B2), neq(B1, B3),
   neq(B2, B3), clear(B1), clear(B3),
   on(B1,B2)],
  [on(B1, B3), clear(B2), not(clear(B3)),
   not(on(B1,B2))]).
```

## *Domain Ontology File: ontology.pl*

The domain ontology file contains contains both a set of domain-general rules such as "*fluent(Pred/Arity) :- not(static(Pred/Arity))",* which should be in every ontology file, and a set of domain-specific facts (e.g., which predicates are static/fluent, object-level/meta-level, or primitive/derived) and rules defining the meaning of derived and metaLevel predicates. The name of the domain ontology file is *ontology.pl* and is found in the domain direction (e.g., the *blocksWorld* directory). For example, in the blocksWorld domain, we could have the following facts and definitions in its domain ontology file:

```
static(cube/1).
fluent(Pred/Arity) :- not(static(Pred/Arity)).

derived(clear/1).
primitive(Pred/Arity) :- not(derived(Pred/Arity)).
%%%% the table is always clear
clear(table).
%%%% cubes are clear if nothing is on top of them
clear(Cube) :- cube(Object), not(on(_, Object)).
%%%% nothing else is clear

metaLevel(neq/2).
```

objectLevel(Pred/Arity) :- not(metaLevel(Pred/Arity)).

## *Problem Files*

Problem files can be called whatever you like as long as it has a "***.pl***" at the end. Problem files must be in the ***problems.d*** directory. In a problem file, there is one problem data structure: **problem(InitialState, Goal)**. The initial state (**InitialState)** is a complete state description (i.e., it contains both the static and the fluent literals (predicates) that describe the initial state of the problem). The problem's file name is how you refer to a specific problem.

## *Marking Schedule*

Still being constructed.

The output from the planner is the sequence of the states from the solution path. It is this sequence of states that will be used to determine whether you have correctly encoded this domain. If your encoding allows the planner to generate an incorrect plan then you have a bug in your encoding.

Sample problems and solutions will be provided soon.