Inference Mechanisms

For this model, the use of three inference mechanisms were employed in an attempt to compare each one's efficiency in a number of aspects. Each mechanism depends on a collection of possible actions and their rules, defining the necessary requirements for execution and the contraints by which they can modify the model. For the purposes of this paper, existing Clojure-based implementations — or adaptations of those implementations — of the following algorithms were aqcuired from the website Agent Domain (https://agent-domain.org)

As an example, an east or west facing car positioned at grid co-ordinates `(3, 2)` can only move to the adjacent grid co-ordinate `(4, 2)` if that sector is actually clear. This would be reflected in a an action called `move-x` with a precondition set that expects `(at 4 2 none)`. This is, of course, a somewhat simplified example, as ultimately the model would provide a level of complexity for which additional preconditions would be required, such as checking whether a co-ordinate is actually adjacent. Failure to consider these additional restraints would likely result in a car somehow teleporting across the map to the first sector it finds.

An example of the operators used in the algorithms employed can be found below:

```
:move-horiz
{:name move-horiz
 :when (
    (at ?vx1 ?y ?v)
    (at ?vx2 ?y ?v)
    (isa ?v vehicle)
    (at ?nx ?y none)
    (:guard (= 1
               (abs (- (? vx1) (? vx2)))
               (abs (- (?nx) (? vx1))))))
  :add {:state ((at ?nx ?y ?v)
                (at ?vx2 ?y none))}
  :del {:state ((at ?nx ?y none)
                (at ?vx2 ?y ?v))}
  :cmd ((move ?v to ?nx ?y))
  :txt ((?v moves to ?nx ?y))
  :cost 1
  }
```

The `:when` property of the operator defines the preconditions required for this operator to be employed, with the parts preceded by a question mark substituted with qualifying values using a pattern matcher (an implementation of which has also been obtained from Agent Domain). The `:guard` property serves to apply additional constraints that don't exist directly within the world state but instead must be calculated.

Upon executing an operator, the contents of the `:del` property, also filled in by the pattern matcher, are removed from the world state and replaced by the contents of the `:add` property.

The `:cmd` property provides an additional sequence of commands to be passed into another function or even language. In this case the contents of this property would be eventually passed into the NetLogo model that visually represents the world state, executing functions that would manipulate that visible model. The `:txt` property, for the most part, serves to provide a textual description of the property that has been executed, usually one that would be easy for a human reader to understand to some degree.

Breadth-first

A breadth-first search mechanism, quite simply, iterates through the various branches of the tree of potential decisions that can be made on the path toward devising a solution. While it remains likely that a breadth-first search could find the most efficient solution, this will, unfortunately, take some time to complete. Potentially, anything from a few minutes to a few hours, depending on the size of the world state, the available operators and the potential branches at each step, of which the search would be traversing every possible branch encountered In addition, expanding the world state by even the smallest amount can serve to expand that tree significantly.

A* (A-star)

A*, or A-star, operates similarly to breadth-first searches, with a key difference in that an additional critera for execution is the cost of each operator, the lowest-costing action being the one that is attempted first. This cost, however, depends on the requirements of the model itself, and assigning a cost to an action could either be arbitrary or, if the developer is feeling brave, have the cost calculate itself with a formula that takes advantage of a number of factors. An example of such a formula could include heuristics such as Hamming Distance (the number of incorrectly-placed objects based on the intended goal state), or Manhattan Distance (the number to "steps" an object must make in order to reach its destination).1

A collection of operators with suitably defined costs or formulae can potentially find the most efficient path to a solution. However, the efficiency of that path isn't always guaranteed.

At the time of writing, the operators developed for this model did not include such formulae for its cost, as time constraints forced the mere implementation of arbitrary 1 or 2 as costs for horizontal movement and vertical movement, respectively.

Goal-based planner

A goal-based planner works similarly to a breadth-first search or A*, however, there exists a key difference in that this inference mechanism begins from a goal state and works its way backwards. As such, an additional few sets of rules are required for each operator to assist and constrain this reverse search. However, if and when the planner succeeds in finding a path, then all that is required for the planner to do is perform each action in the resulting stack.

Developing the ops for a goal-based planner can be extremely difficult, as the goal state it works with is only partial (that is to say, it only contains the actual conditions required for a world state to qualify as a goal state, with none of the remaining world state included), and can thus only take the original state into account at all times, or until it reaches a point at which it can work through the stack and complete some operators. In addition, the developer desigining the operators must learn to work backwards with these restrictions in account. This can be alleivated somewhat by including operators that mark targets as "protected", but this still requires determining what needs protecting and when.

An example of an operator with the aforementioned additional rules can be seen below:

```
:move-horiz
{
  :name move-horiz
  :achieves (at ?x ?y ?v)
  :when (
         (at ?x ?y none)
         (at ?nx1 ?y ?_)
         (at ?nx2 ?y ?_)
         (at ?vx1 ?y ?v)
         (at ?vx2 ?y ?v)
         (clear-me ?vx ?y ?v)
         (isa ?v vehicle)
         (:guard (and
                   (not= (? x) (? nx1) (? nx2))
                   (= 1
                      (abs (- (? nx1) (? nx2)))
                      (abs (- (? x) (? nx1)))
                      (abs (- (? vx1) (? vx2))))
                   (or (= (? vx1) (? vx))
                       (= (? vx2) (? vx)))
                   (> (abs (- (? x) (? nx2)))
                        (abs (- (? x) (? nx1))))
                     (> (abs (- (? x) (? vx1)))
                        (abs (- (? nx1) (? vx1)))))))
  :post (
         (protected ?x ?y ?v)
         (at ?nx1 ?y ?v)
         )
  :pre (
         (at ?nx1 ?y ?v)
         (at ?x ?y none)
         (protected ?x ?y ?v)
         )
  :del (
        (at ?nx2 ?y ?v)
        (at ?x ?y none)
        )
  :add (
        (at ?x ?y ?v)
        (at ?nx2 ?y none)
        )
  :cmd ((move ?v ?x ?y))
  :txt ((?v moves to ?x ?y))
  }
```

In this case, `:when` has become the preconditions for adding this operation to the stack that would eventually be executed in the correct sequence, with the preconditions for actually executing the operation now under the `:pre` property.

An additional property, `:post` adds some postconditions for prior steps in the stack to achieve, which will be compared with the `:acheives` property as a means of more quickly determining operators to consider. All other properties work in the exact same manner as their A* counterparts.

Of additional note is the concept of "protecting" items in the world state. In this case, protection serves to establish where a vehicle should exist by that poinnt, and as such serve as a guiding precondition. An example of the operator for protecting a position can be seen below.

```
:protect
{
 :name protect
 :achieves (protected ?x ?y ?v)
 :add ((protected ?x ?y ?v))
 }
```

REFERENCES