|  |
| --- |
| AIA Part 2 - pitch |

## Team

Team B:

* P4048819 - Daniel Boden
* P4261930 - Dean Duff
* P4048801 - Jo Humphreys

**Title**

A Comparative Analysis of Inference Mechanisms Applied to Solving the Rush Hour Board Game.

### Focus

A comparative analysis of two or more inference mechanisms, which are utilised to solve a puzzle similar to Nobuyuki “Nob” Yoshigahara’s board game Rush Hour, noting their efficiencies and shortcomings as well as comparing the results of the application of each algorithm to a variety of puzzles, within a wide range of sizes and complexities. From the aforementioned comparison, it may be possible to determine the ideal algorithm of the two for the puzzle format.

#### Inference Mechanisms:

* A\* Search - Provided by agent-domain (<http://s573859921.websitehome.co.uk/pub/clj/tools/Astar-search(2a).clj)>
* Planner Algorithm – Provided by agent-domain (<http://s573859921.websitehome.co.uk/pub/clj/tools/planner(1a).clj>)
* Additional standard Breadth-first and Depth-first search algorithms, time permitting, to support further analysis.

#### Performance Metrics:

* Time taken to find a solution that leads to the goal state.
* Steps involved in solution found (Efficiency).
* Possible RAM usage for each mechanism (Time permitting).

### Experimental Brief

The experiment consists of a slider type puzzle with the theme of a crowded car park, from which a vehicle must exit. However, the crowded nature of the car park has led to several other vehicles blocking the exit. These vehicles most be moved out of the way to allow the player vehicle to leave, but they are in turn obstructed themselves. Thus, the player must rearrange the car park with a number of constraints in mind in order to leave.

#### Netlogo:

* Rendering the puzzle board and its pieces in a manner that makes the position and direction of each car clear.
* Representing the various states that will be reached via each inference mechanism.
* Receipt of metrics recorded by the Clojure code, and displayed to the user, as well as output via BehaviorSpace that can then be passed into graphs for clear representation.

#### Clojure:

* Obtain the world state from the NetLogo model and store it in a format that can be easily modified.
  + Example data format: ((isa 2 car) (colour 2 yellow) (x 2 (3 4)) (y 2 (6)))
* Define the rules of the puzzle
  + Horizontal vehicles can only move left or right.
  + Vertical vehicles can only move up or down.
  + Vehicles cannot pass through one another.
* Record and return available metrics from the process of calculating a solution.

### Key features of experimental scenario

* The player vehicle has to leave via the exit.
  + Only one such exit exists.
* Vehicles will face one of the four cardinal directions on the board.
  + However, these vehicles remain locked in their given orientation. They can only move back and forth, and never sideways. In addition, the vehicles can never turn.
  + Vehicles also obstruct one another – no vehicle may pass through another like a ghost – further restricting each vehicle’s range of movement at any one time.
  + As such, the player vehicle will always line up with and face the exit.
* Vehicles can vary by length, based on the board’s grid, to a minimum of 2 grid squares, but not width (which remains at a size of 1 grid square).

#### Extending the problem:

* Most implementations of this puzzle begin with a board with no static obstacles. However, the introduction of such obstacles could potentially be implemented.
* Variable board sizes can also be considered, as a means to testing each algorithm against larger and smaller boards and comparing the results.
* Variation of car layouts within a specific sized board, which not only includes new positions but also variation in vehicle density, can also be considered.

### Inference Mechanisms

LOOK AT THIS AS A GROUP

* Operators:
  + (move-x <vehicle number> <destination-x>)
    - Pre: car has only one y co-ordinate.
    - Pre: empty space between origin-x and destination-x
    - Pre: empty space between destination (front) and destination (rear)
    - Del: (x <vehicle number> (<origin-front-x> <origin-rear-x>))
    - Add: (x <vehicle number> (<dest-front-x> <dest-rear-x))
  + (move-y <car number> <destination-y>)
    - Pre: car has only one x co-ordinate.
    - Pre: empty space between origin-y and destination-y
    - Pre: empty space between destination (front) and destination (rear)
    - Del: (y <vehicle number> (<origin-front-y> <origin-rear-y>))
    - Add: (y <vehicle number> (<dest-front-y> <dest-rear-y))
  + (exit)
    - Pre: (exit (x y))
    - Pre: Front or rear co-ordinate matches exit co-ordinate
    - Del: None
    - Add: (solved)?
* World state tuples:
  + (map (<x1> <y1>) (<x2> <y2>))
  + (exit (x y))
  + (isa <vehicle number> car)
  + (colour <vehicle number> blue)
  + (x <vehicle number> (<x>))
  + (x <vehicle number> (<front-x> <rear-x>))
  + (y <vehicle number> (<y>))
  + (y <vehicle number> (<front-y> <rear-y>))

***Algorithm Usage:***

* A\*
  + Each movement would have a cost based on an as-yet undetermined factor.
    - Current considerations include basing the cost on a vehicle destination’s proximity to the exit or the likelihood of obstructing another vehicle.
    - Ideally, the algorithm would find a solution by seeking out the least-costly moves. However, it may be possible that the solution found may not be the most ideal one, experimentation with a number of cost applications may be necessary.
* Planner
  + The planner starts from a simplified goal state (for example, the player vehicle finds the exit, but the rest of the world state at this point will be irrelevant), and works its way back to the original starting world state, at which point, it will then apply the moves it has chosen in the correct order.

### Workload

* Construction of the user interface and the model representing the game board.
  + 2 days
  + Boards will be pre-designed, rather than procedurally generated, to allow for consistent experiments.
* Implementation of data transfer between NetLogo and Clojure.
  + 2 days
  + Data regarding initial board layout will be sent from NetLogo to Clojure in tuple format.
  + Instructions for individual steps required to solve the puzzle, as well as recorded metrics, will be sent from Clojure to NetLogo.
* Implementation of the A\* algorithm and necessary rules in Clojure.
  + 4 days
  + Implementation of cost still needs to be determined at the time of writing.
* Implementation of the Planner algorithm and necessary rules in Clojure.
  + 4 days
  + The operators listed above are subject to expansion or change.
* Implementation of BehaviorSpace for collection of metrics.
  + 1 day
* Gathering of results from each experiment.
  + 1 days
  + While collection raw results may take a matter of minutes, they will need to be converted to graphs for better readability.
* Completion of the paper.
  + 5 days
  + Considerable proofreading, discussion and analysis of gathered data and adequate citation (including finding additional credible sources) required.