Comparison of Inference engines in a town traversal problem

AIA ICA 2 – Paper

Team M

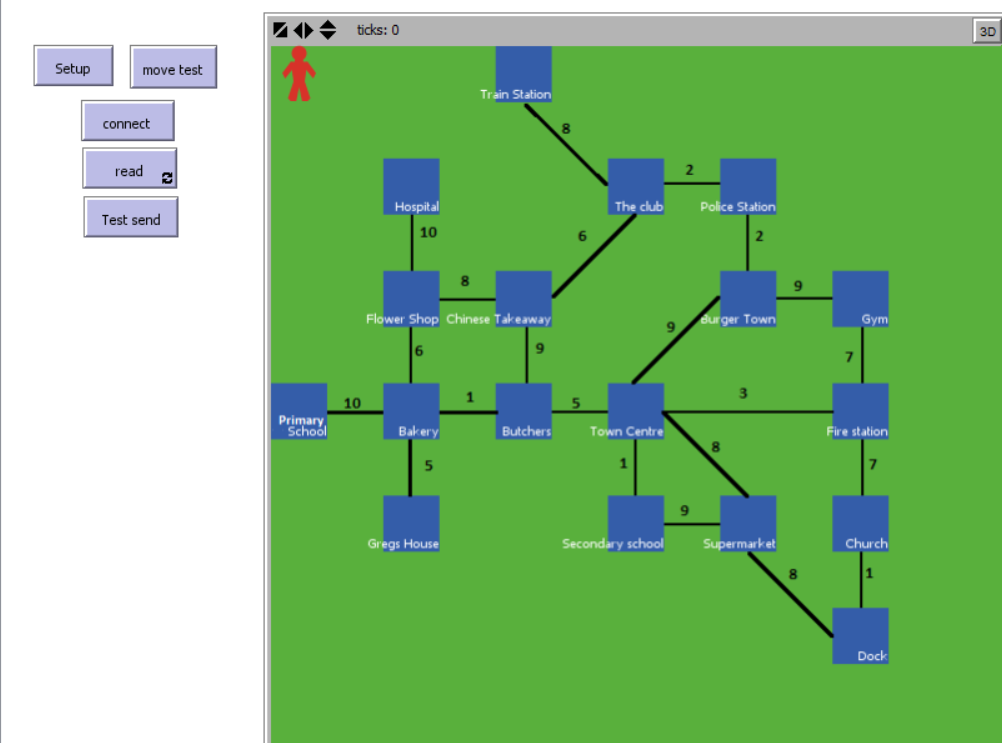
|  |  |  |
| --- | --- | --- |
| P.J.McTeague  School of Computing  Teesside University  p4029206@tees.ac.uk | A.J.Mitchell  School of Computing  Teesside University  p4258660@tees.ac.uk | K.W Wood-Toward  School of Computing  Teesside University  t7235345@tees.ac.uk |

Abstract

The task was to create a intelligence engine with a NetLogo based frontend, natural language processing input through a UI and a backend utilizing various inference engines in order to compare different intelligence engines performing a task. We chose to model a town-traversal (or path finding) problem.

Design

The model was created by combining a Clojure backend, created using Cursive for IntelliJ (Cursive Limited, 2018) for taking user input in the form of text and processing it using some basic morphological processing and inference engines, created by Simon Lynch and hosted on his site ("Clojure tools", 2018). This was followed by sending the result through to a NetLogo agent based frontend via Simons sock2 socket ("NetLogo", 2018), which takes the output of the backend and moves the agent around a representation of our town using that output.   
The backend works by having a user enter a command such as “Move from the town-centre to the club” and then stripping unimportant words from it, transforming it to the sentence “move town-centre club”. This is then used as input for another function which calls one of the inference engines. These inference engines search a list of state definitions for a route, with more complexity added dependent on which inference engine is used, then returns a list of commands, such as “move-to gym”, which when ran in sequence would move an agent from the start location to their goal state. These outputs are then formatted so that the output is common among all inference engines and sent over a socket as a string, to the frontend, with a delay between each send so that the output can be observed.



The frontend displayed to the user

The frontend contains labels placed at positions in the town, acting as a visual representation of the town and an agent. When setup these are displayed to the user. The connect button must then be pressed to connect via the socket to the running backend. The read button is then pressed and from there anything sent over the socket is read. The frontend checks to see if it understands what has been sent and acts accordingly. For example, “move-to gym” is understood and the agent in the frontend is moved to the label which has the text “gym”. Below two code extracts showing how the frontend processes strings sent through the socket are shown.

let text sock2:read

print (word "reading: " text)

if not (text = "")

[

if member? "move-to" text

[

move-to-node (remove "move-to " text)

]…

to move-to-node [destination]

print destination

print (one-of patches with [plabel = destination])

ask turtle 0

[move-to (one-of patches with [plabel = destination])]

End

Inference engines

The inference engines used are Breadth First search, A\* Search and Ops Search.   
  
Breadth First search was chosen as it’s a simplistic search that will act as a good base case by giving a simple result. It works by taking the start location, and then finding which locations can be reached from there. It then takes the locations that are reached from each one of them in turns and repeats this until it finds the goal location. Breadth first however can’t account for anything more than which places connect though which does make it quite limited as it cannot account for the time taken to move along different roads, meaning it will not necessarily find the best solution (though it may by chance) but will return the first solution it comes across.

(def breadth-first-lmg

'{

primary-school

(

bakery

)

gregs-house

(

bakery

)

A\* search was used as it takes cost into account in order to find the best result. It’s a type of best first search algorithm that searches all possible paths to the goal and returns the one which has the least total cost. This works well with our model as our town can have costs in terms of the time taken to travel each road and, so it’d be interesting to see the difference in result and execution time, accounting for this extra time taken to find the best result – the one with the lowest cost. It uses a legal move generator containing states and what states can be reached from them for what cost.

(defn a\*lmg [state]

(let [n (:state state)

c (:cost state)

]

(case n

"primary-school"

(list

{:state "bakery", :cost (+ c 10)}

)

"gregs-house"

(list

{:state "bakery", :cost (+ c 5)}

)

Ops-search was used as it supports more commands/operations than just moving, as we can write operation definitions for it. Ops-search is an implementation of breadth first search that applies operators to change state, until it gets to the goal state. This search is useful for demonstrating extensibility which we did using a “purchase” operation where the agent can get items from places. It however will not find the best solution like A\* does, as it does not account for cost.

(def ops

'{move {:pre ( (agent ?agent)

(at ?agent ?p1)

(connects ?p1 ?p2)

)

:add ((at ?agent ?p2))

:del ((at ?agent ?p1))

:txt (move ?p1 to ?p2)

:cmd [move ?p2]

}

purchase {:pre ((agent ?agent)

(has ?obj ?place)

(at ?agent ?place)

)

:add ((has ?obj ?agent))

:txt (purchase ?obj from ?place)

:cmd [purchase ?obj]

}

})

;; A world state representing the town in the town traversal problem

(def ops-world-state

'#{(connects primary-school bakery)

(connects gregs-house bakery)

NetLogo Representation

NetLogo was chosen to be used to display a front end for the solution due to its unique ability to quickly produce a simple prototype.

The map was made by creating a function called “draw-location” which labels a certain patch by coordinates and gives it a name. These labels can then be used to refer to these coordinates, meaning that communication between the NetLogo front end and Clojure back end becomes simpler. This is because Clojure can now pass a string to NetLogo without needing to know the coordinates of the locations and NetLogo does not need to interpret the location coordinates in the function. This also allows for easier extensibility in adding, removing, and editing new locations.

The map itself was created in paint, this was done because creating paths with weights as required to illustrate which locations connect is quite an awkward process. Initially it was attempted with patches but it was found that with a location with an order of 5 or more it becomes very messy. Although a very simple method it is still very robust allowing for adjustments to the map, as well as quicker than the alternatives, which creates other issues such as with the clarity of the map.

For the NetLogo front end to display information from the Clojure back end there must be a way for them to communicate. In this instance a 'socket' was used using the 'sock2' extension. The socket must be set up on either side and then they must establish a connection via a local port, upon doing so the Clojure back end can then send a string which the front end listens for at intervals. The string is then manipulated into a format which can then be called to update the front end display.

The functions in the NetLogo front end are as follows: setup, setup-world, draw-location, connect, read, move-to-node.

Setup clears the graphics, calls setup-world , imports the map, resets the ticks and initialises the agent.

to setup

clear-all

setup-world

import-drawing "Map.PNG"

create-turtles 1 [set shape "Person" set color red]

reset-ticks

end

Setup-world initialises the locations and labels them by calling the draw-location function.

to setup-world

set-patch-size 45

ask patches [ set plabel-color white ]

draw-location 0 -6 "primary-school"

end

Draw-location takes three inputs, two referring to the coordinate of the location and the last one labelling it using plable, which is built in to netlogo.

to draw-location [x y name]

ask patch x y [ set plabel name ]

end

Connect is used to establish a connection with a port allowing for communication to take place.

to connect

sock2:connect-local 2222

set n 0

end

Read checks what the string contains and manipulates it to be in a format where the move-to-node function can understand it.

to read

let text sock2:read

print (word "reading: " text)

if not (text = "")

[

if member? "move-to" text

[

move-to-node (remove "move-to " text)

]

if member? "purchase" text

[

print "Agent has purchased "

print (remove "purchase " text)

]

]

end

Move-to-node takes in an input as a destination name and then moves to the location via the plabel.

to move-to-node [destination]

print destination

print (one-of patches with [plabel = destination])

ask turtle 0

[move-to (one-of patches with [plabel = destination])]

end

Results & Analysis

A number of benchmarks were carried out, using Criterium for Clojure ("hugoduncan/criterium", 2018) measuring the time the inference engines took to complete a traversal under different circumstances. By doing so conclusions can be drawn as to how certain factors can affect the algorithms differently allowing us to see which they are best suited.

Each benchmark was carried out in Clojure over many iterations with extreme outliers being removed from the data set, the averages were then used as comparisons.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Execution time mean / µs | Execution time std-deviation / µs | Execution time lower quantile / µs | Execution time upper quantile / µs |
| Breadth Traversal from Gym to Primary School | 83.676647 | 1.809285 | 82.532246 | 89.190184 |
| A\* Traversal from Gym to Primary School | 82.198822 | 0.15629099 | 81.940021 | 82.521481 |
| Ops-Search Traversal from Gym to Primary School | 21060.16 | 255.179251 | 20956.25 | 21234.292 |
| Breadth Traversal from Bakery to Supermarket | 13.088846 | 0.434261053 | 12.867077 | 13.9059 |
| A\* Traversal from Bakery to Supermarket | 39.697932 | 0.697366348 | 39.392735 | 41.66082 |
| Ops-Search Traversal from Bakery to Supermarket | 8685.747 | 574.617398 | 8156.073 | 10147.621 |
| Breadth Traversal from Dock to Fire-station | 3.921824 | 0.1503082 | 3.774827 | 4.256188 |
| A\* Traversal from Dock to Fire-station | 4.982522 | 0.187473788 | 4.846649 | 5.570302 |
| Ops-Search Traversal from Dock to Fire-station | 2648.658 | 83.81062 | 2595.574 | 2885.31 |

When comparing the mean execution time of all searches between the gym and the primary school, the fastest outcome is the A\* search, which is then very closely followed by the Breadth First Search, and finally the Ops-Search which took a magnitude of 250 times as long. This is most likely due to the nature of the Ops-Search which must look at the goal states available to it, which although slowing it down does allow for extensibility. For example in the task the agent can pick up fast food from certain locations and drop it off at others, or could be told to go somewhere via a certain location.

Although one search may be the fastest it may be worth considering other factors. For example when comparing finding the quickest route in terms of distance A\* will always come out on top as it takes into account cost of each path, whereas BFS does not. Ops-Search will also always find the quickest route in terms of distance as well as the fact it is extensible.

Looking at the standard deviation it can be noted that the Ops-search has the largest deviation, this is most likely due to the complexity of the algorithm and that it will check all legal move states to reach its goal. A\* generally had negligible standard deviation most likely due to the fact that is the best of both worlds in that it takes into account cost, unlike BFS, but is also less complex than the Ops-Search as well as stopping as soon as it finds a solution. The BFS had very low standard deviation, although it did have a factor of almost 7 more than the A\*. This is because it does not take into account cost, and just blindly cycles through locations in a set manner until a solution is found.

Summary

In summary the recommendation is that Ops-Search is the most complete search and therefore should be used when time is not a factors and extensibility is desired. Although an A\* search will work fine when trying to find the fastest route quickest. BFS should only be used when the others are not an option.

References

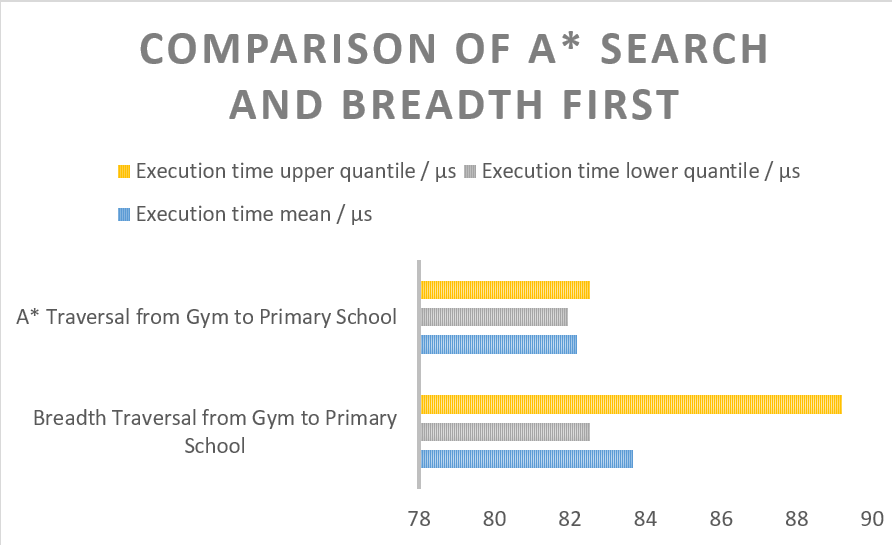
Cursive Limited, h. (2018). *Cursive: The IDE for beautiful Clojure(Script) code*. *Cursive-ide.com*. Retrieved 11 January 2018, from <https://cursive-ide.com/>

Clojure tools. (2018). S573859921.websitehome.co.uk. Retrieved 11 January 2018, from <http://s573859921.websitehome.co.uk/pub/clj/tools_index.htm>

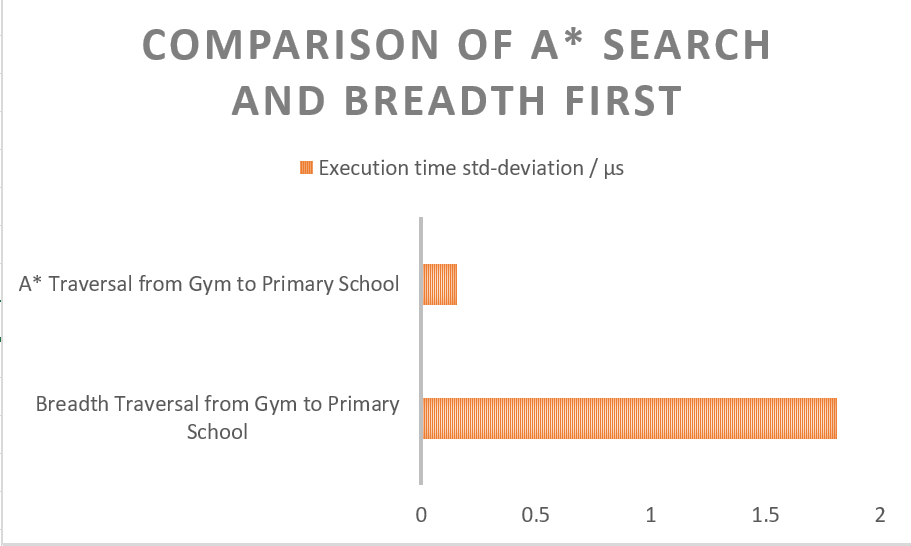
NetLogo. (2018). S573859921.websitehome.co.uk. Retrieved 11 January 2018, from <http://s573859921.websitehome.co.uk/pub/netlogo/cogs_nl_tools.htm>

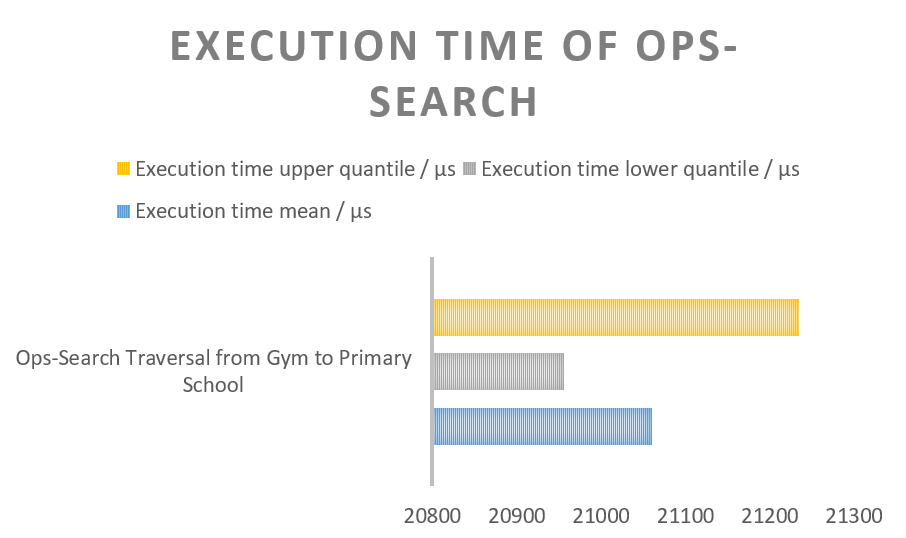
hugoduncan/criterium. (2018). GitHub. Retrieved 11 January 2018, from https://github.com/hugoduncan/criterium

Appendices

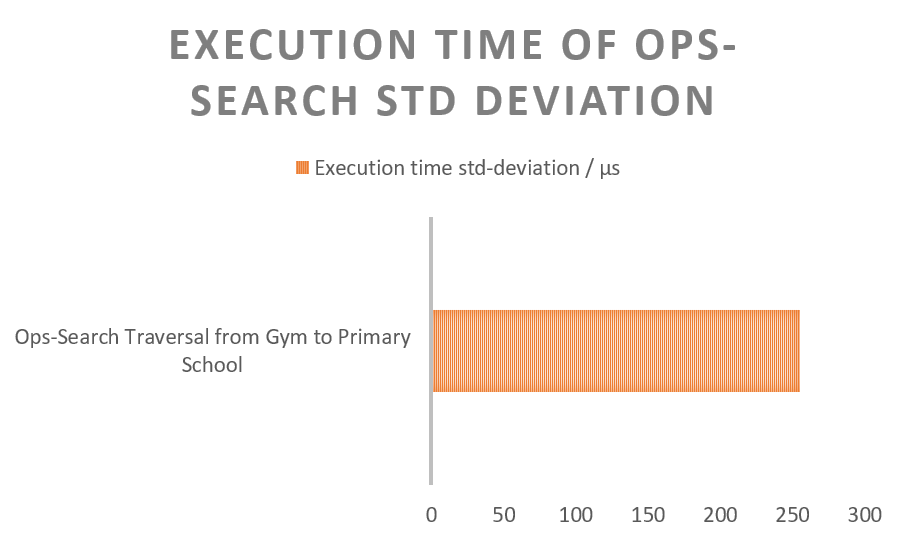
****

A graph showing a comparison of A\* and Breadth First search processing times



A graph showing a comparison of the standard deviation of A\* and Breadth First search

A graph showing the execution time of ops-search



A graph showing the standard deviation of ops-search execution time