Institute for Advanced Studies
in Basic Sciences
Gava Zang, Zanjan, Iran

Multi-agent Systems

Final Project

By: Sina Heydari
Student ID: 14024133

Introduction

Since we are mortal creatures, we tend to escape the reality of death hiding behind mental structures we build. It's not strange that the lines have escaped our minds and came to reality taking shapes into structures we build to overcome chaos although, many has always argued the absurd nature of order in human hands.

To begin with, organizing transportation has always been interesting to human mind. In this project I tried to visualize the very simple and basic features of a transportation system like Uber using A-star algorithm.

First let's take a look at the basic concepts that has been implemented in the project, then I will fully explain the code structure.

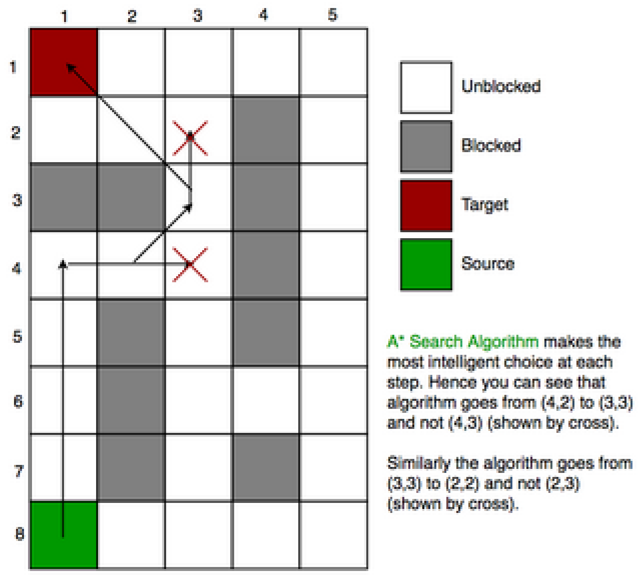
A-star

What is A* Search Algorithm?
A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Why A* Search Algorithm?
Informally speaking, A* Search algorithms, unlike other traversal techniques, it has "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.
And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Explanation
Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.
What A* Search Algorithm does is that at each step it picks the node according to a value-'**f**' which is a parameter equal to the sum of two other parameters – '**g**' and '**h**'. At each step it picks the node/cell having the lowest '**f**', and process that node/cell.
We define '**g**' and '**h**' as simply as possible below
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this 'h' which are discussed in the later sections.

Suppose as in the below figure if we want to reach the target cell from the source cell, then the A* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristics.


1
2
3
4
5
Unblocked
Blocked
Target
Source
A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).
Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

Heuristics

We can calculate **g** but how to calculate **h**?

We can do things.

A) Either calculate the exact value of h (which is certainly time consuming).

OR

B) Approximate the value of **h** using some heuristics (less time consuming).

Here we went down the second road.

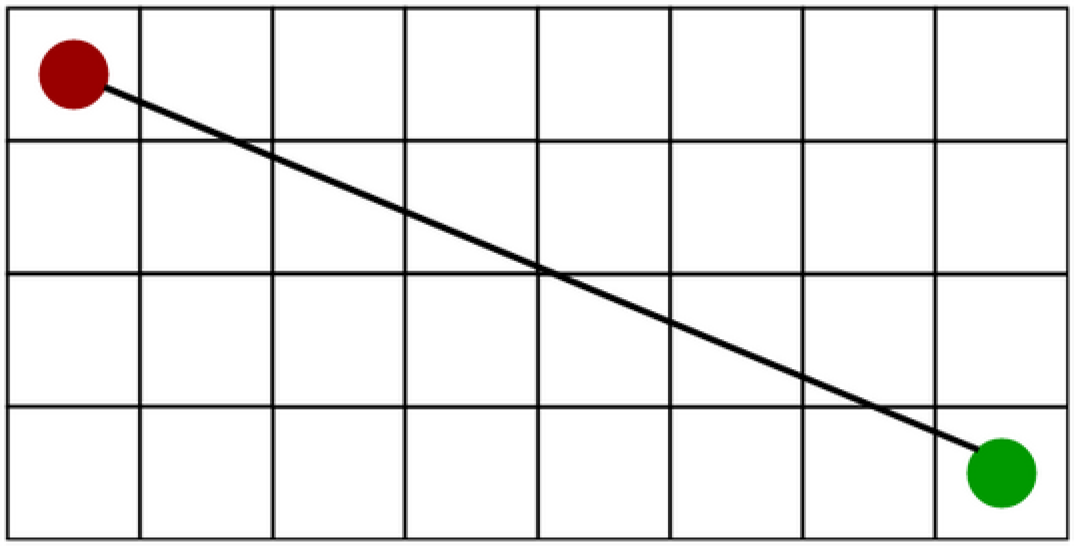
Euclidean Distance

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula

```
  h = sqrt ( (current_cell.x – goal.x)2 +
            (current_cell.y – goal.y)2 )
```

- When to use this heuristic? – When we are allowed to move in any directions.

The Euclidean Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).

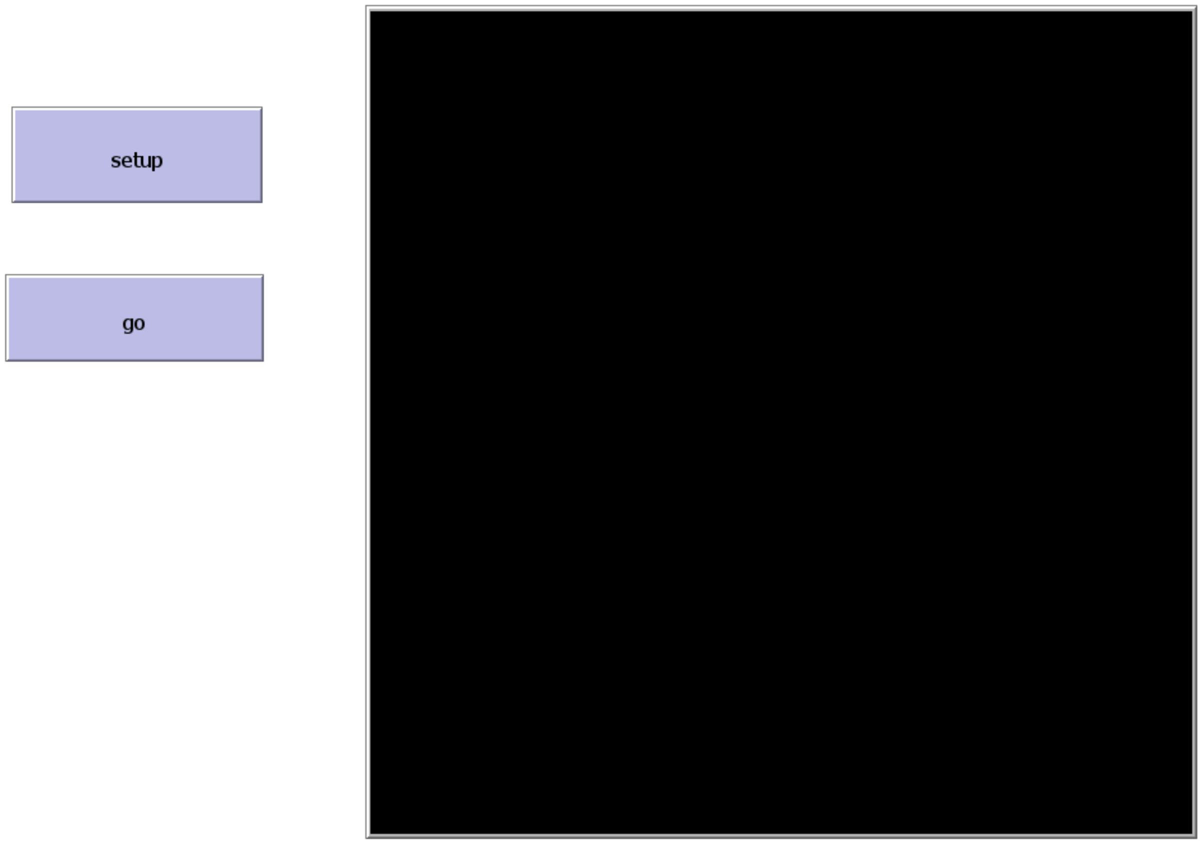


In the code, I also used **Euclidian Distance** to select the nearest cab to the passenger which, we will fully discuss further in the document.

User Interface

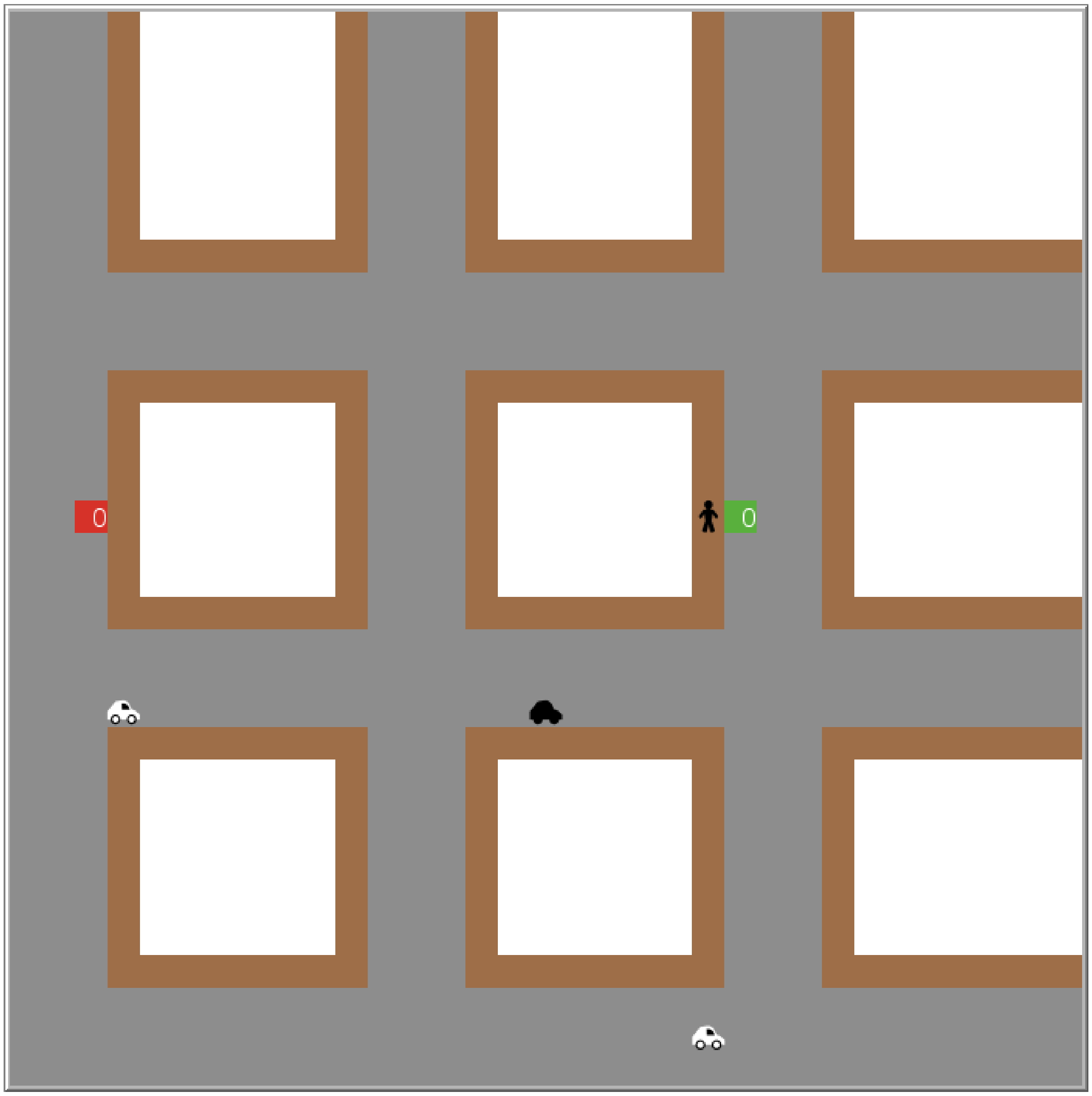
As show in the picture below, the simplicity of the work done might bother the eye in the first sight but as I explained earlier this is just as basic implementation. There are lots of work to be done and feature to include.

Firstly the two button **setup** and **go** are being used to set the environment and go for running the code.



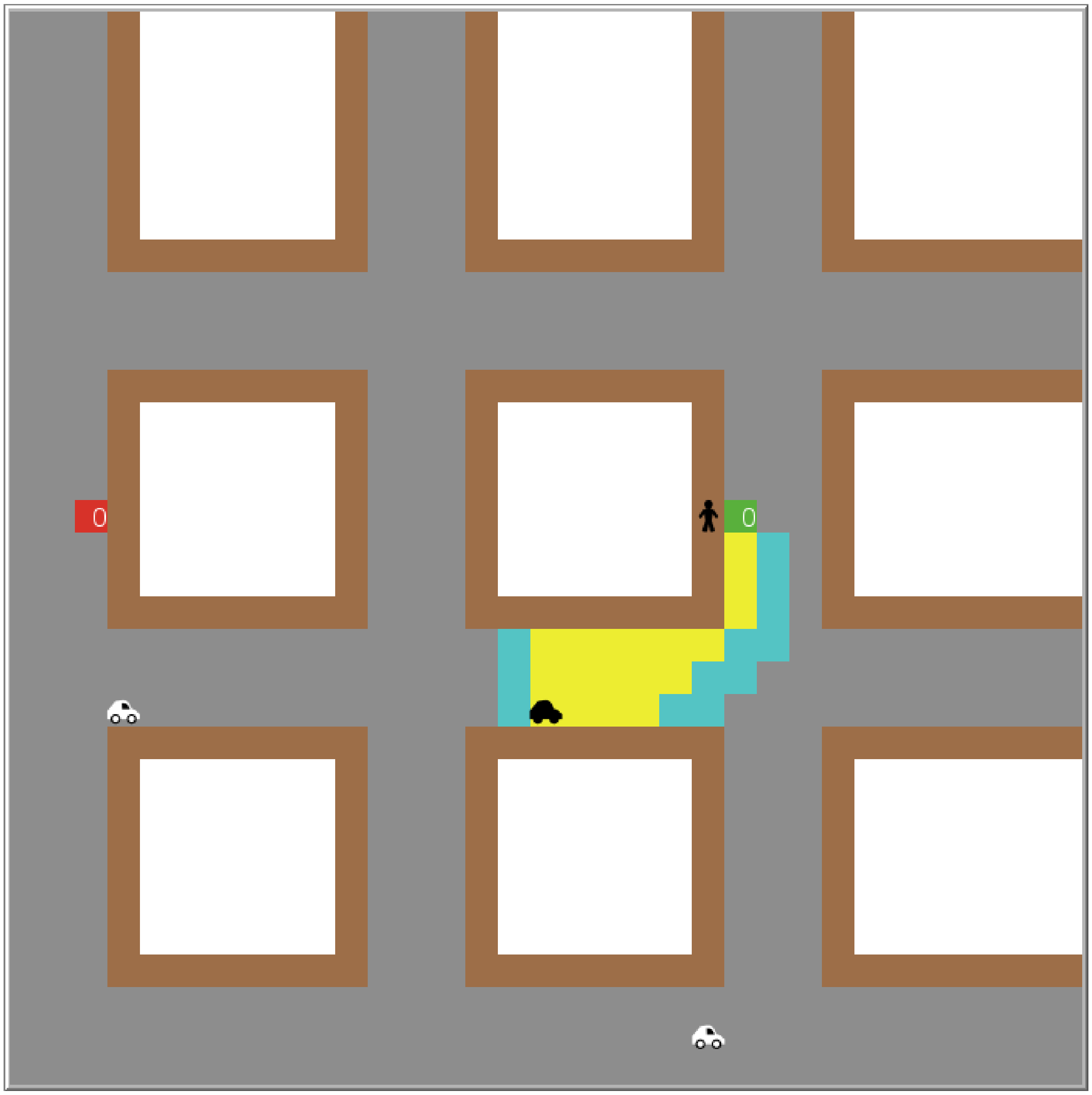
As we first lunch the code in netlogo, we see the picture above.

If we push the **setup** button the world turns to what is shown in the picture below.

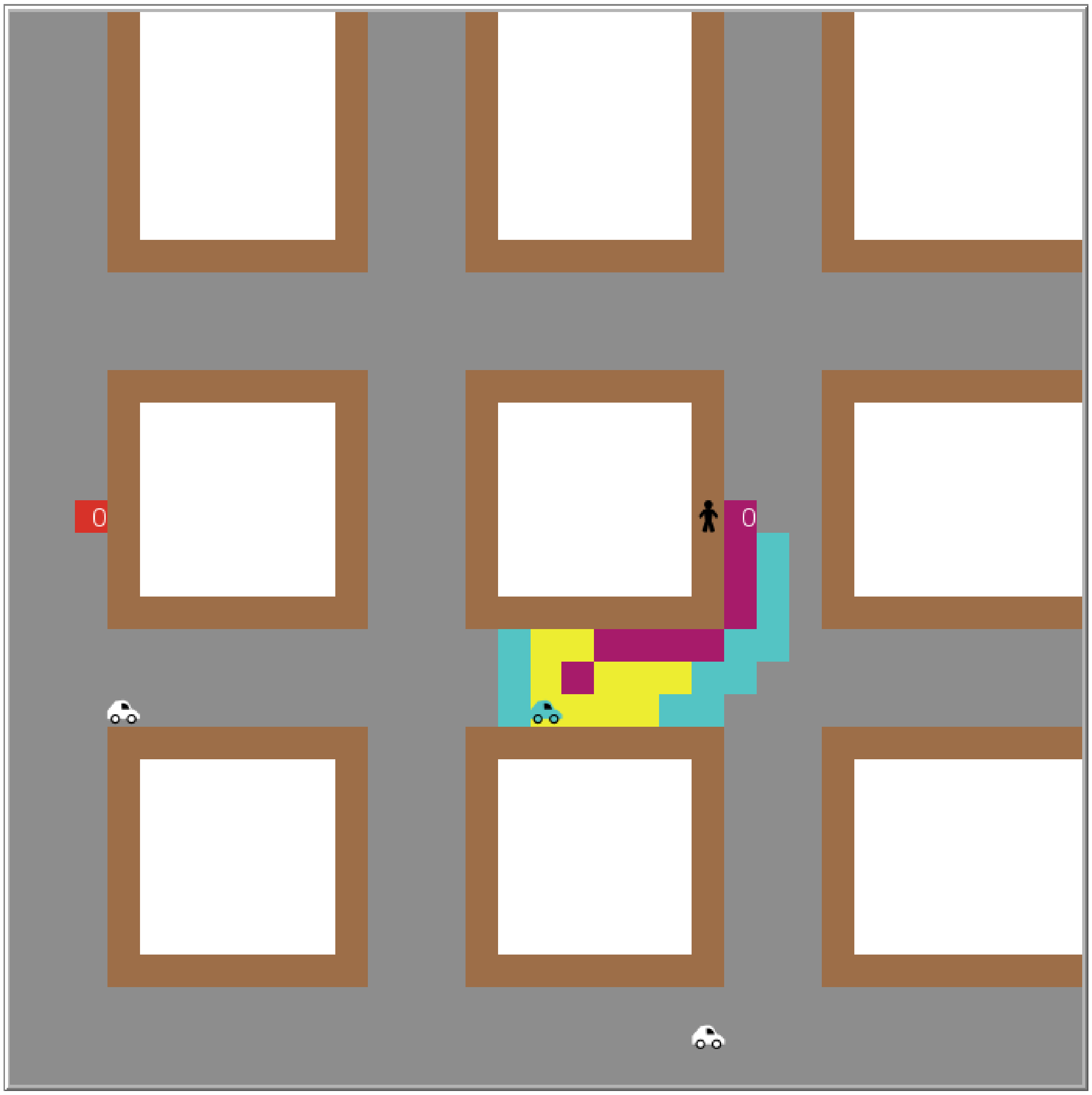


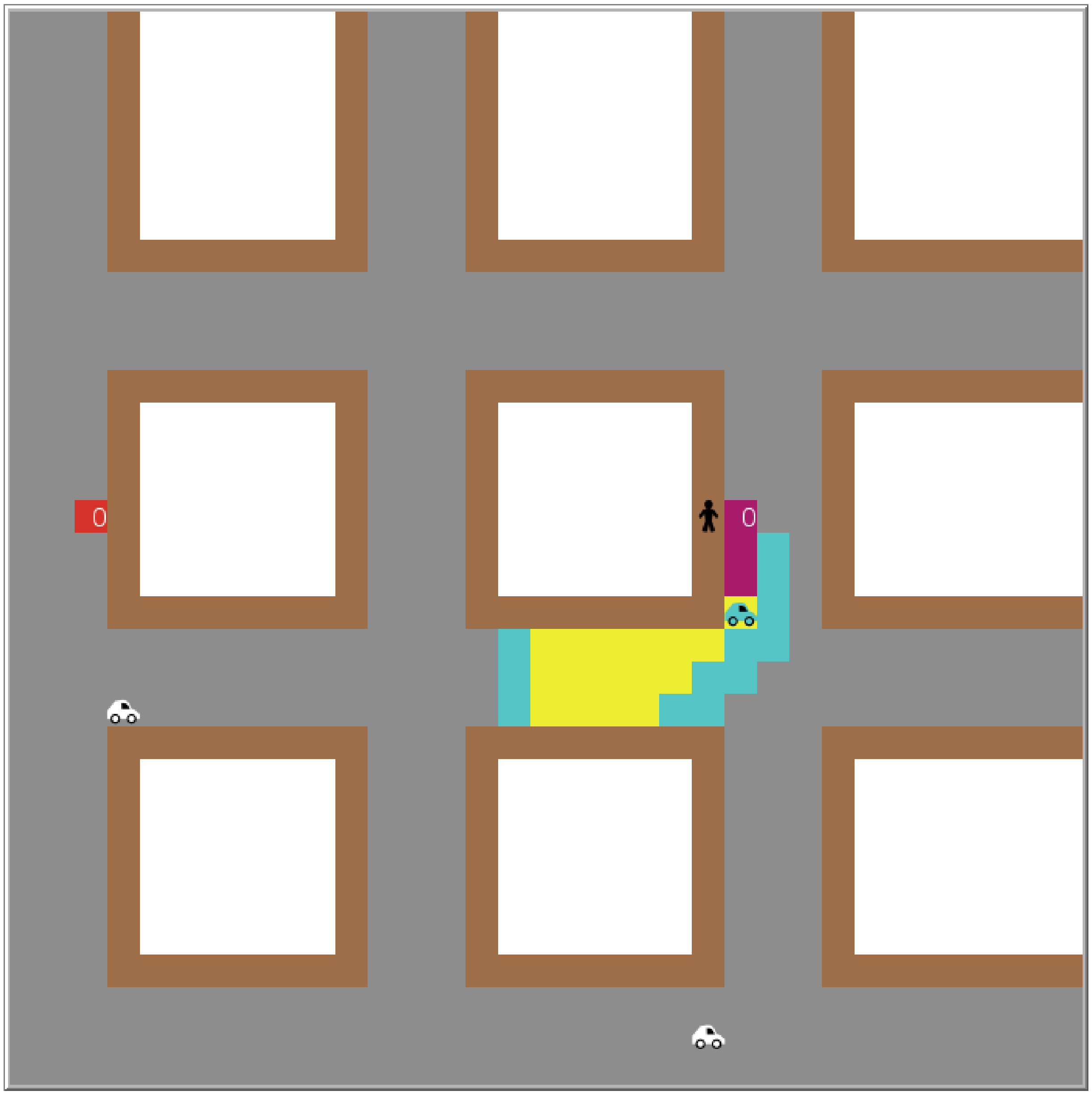
As show here we have three cabs and one passenger. There is a method to find the nearest cab to the passenger called **find-nearest-cab**. And another method to calculate the distances between cabs and the passenger called **cab-passenger-distance.** In **cab-passenger-distance** we have used Euclidian Distance to calculate the distance and in **find-nearest-cab-** we select the cab with the lowest distance and turn its color to black.

By pushing the go button the black cab searches for the nearest route using **A*** algorithm which is shown below.

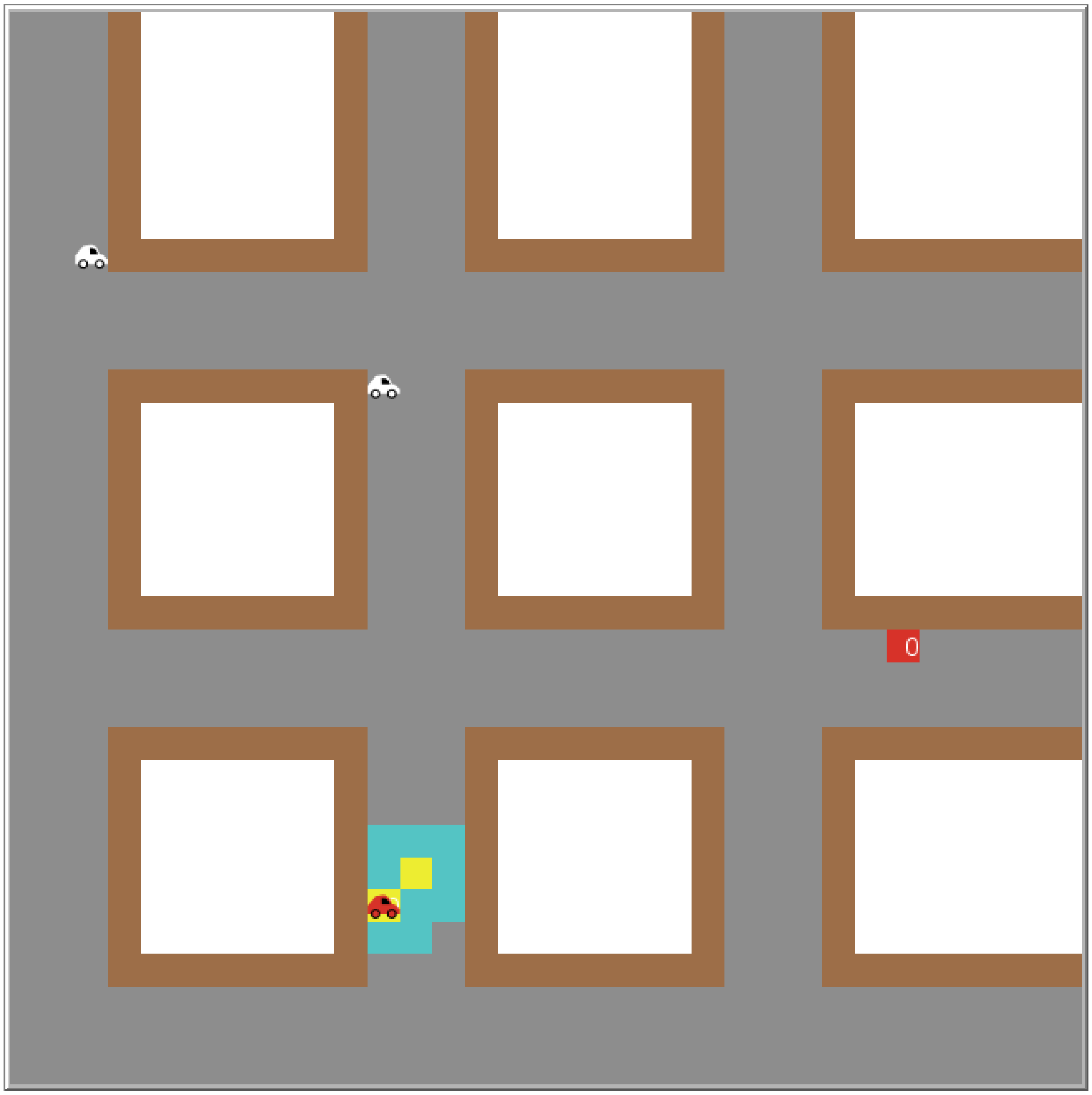


After our search space reaches the patch next to the passenger witch, is marked by the green color the route turns to purple and the cab turns to blue, then start to drive to the pickup point marked in front of the passenger. And as the cab starts to move each patch it travels through turn yellow and fades into the search space discovered.

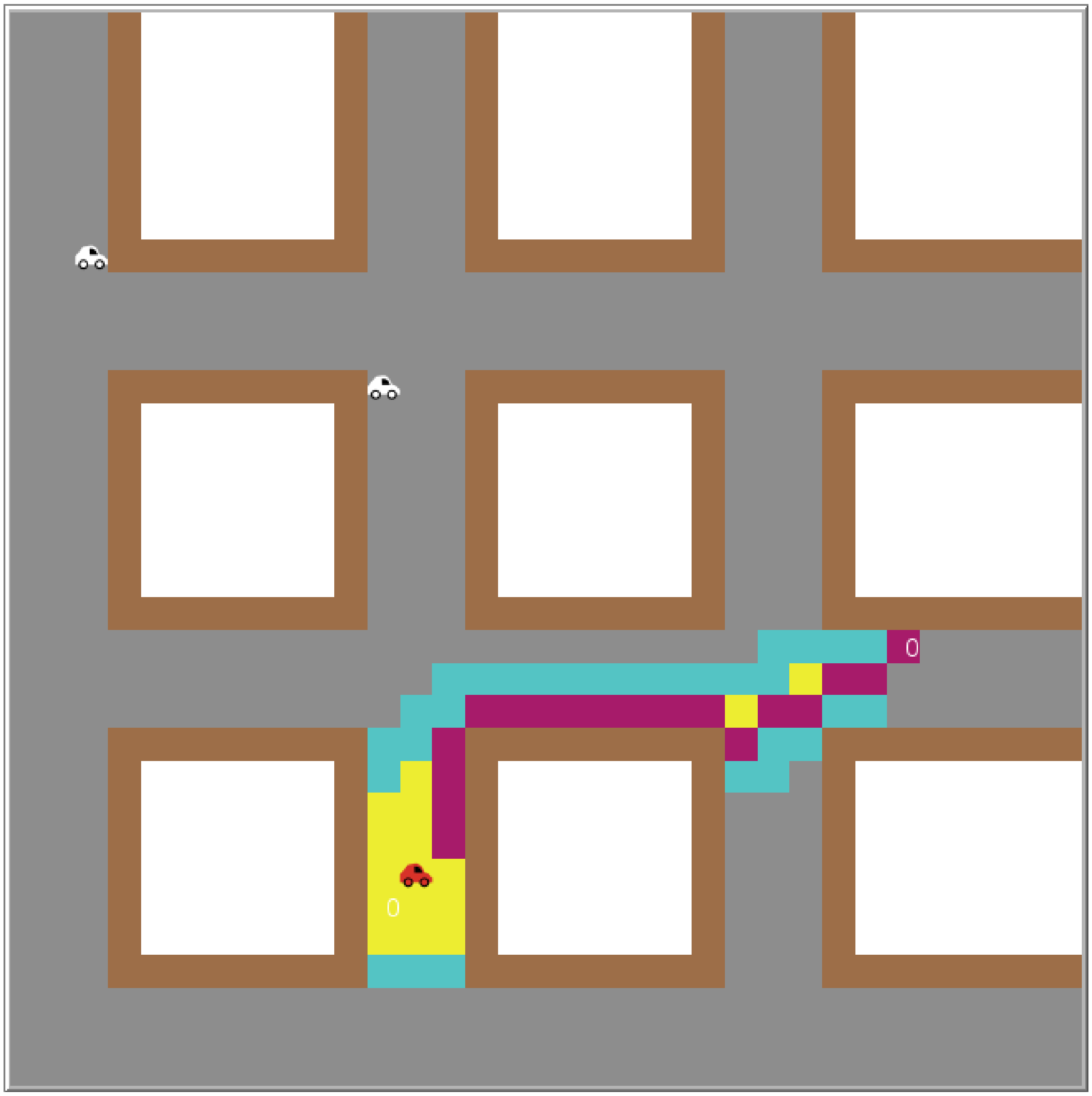


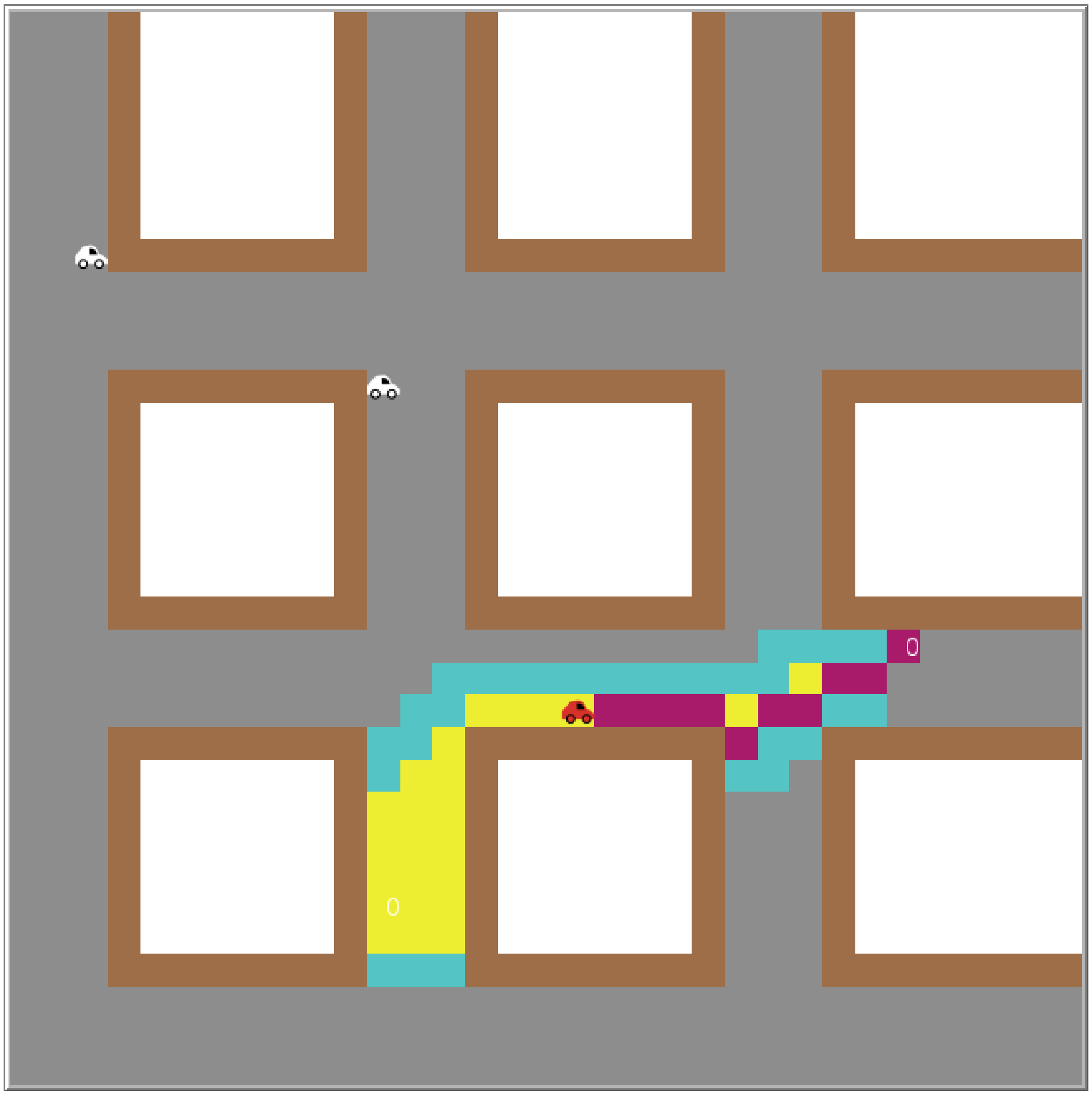


After reaching the pickup point the search space disappears and the cab picks up the passenger and turn into red and start the search to find the destination marked with red.

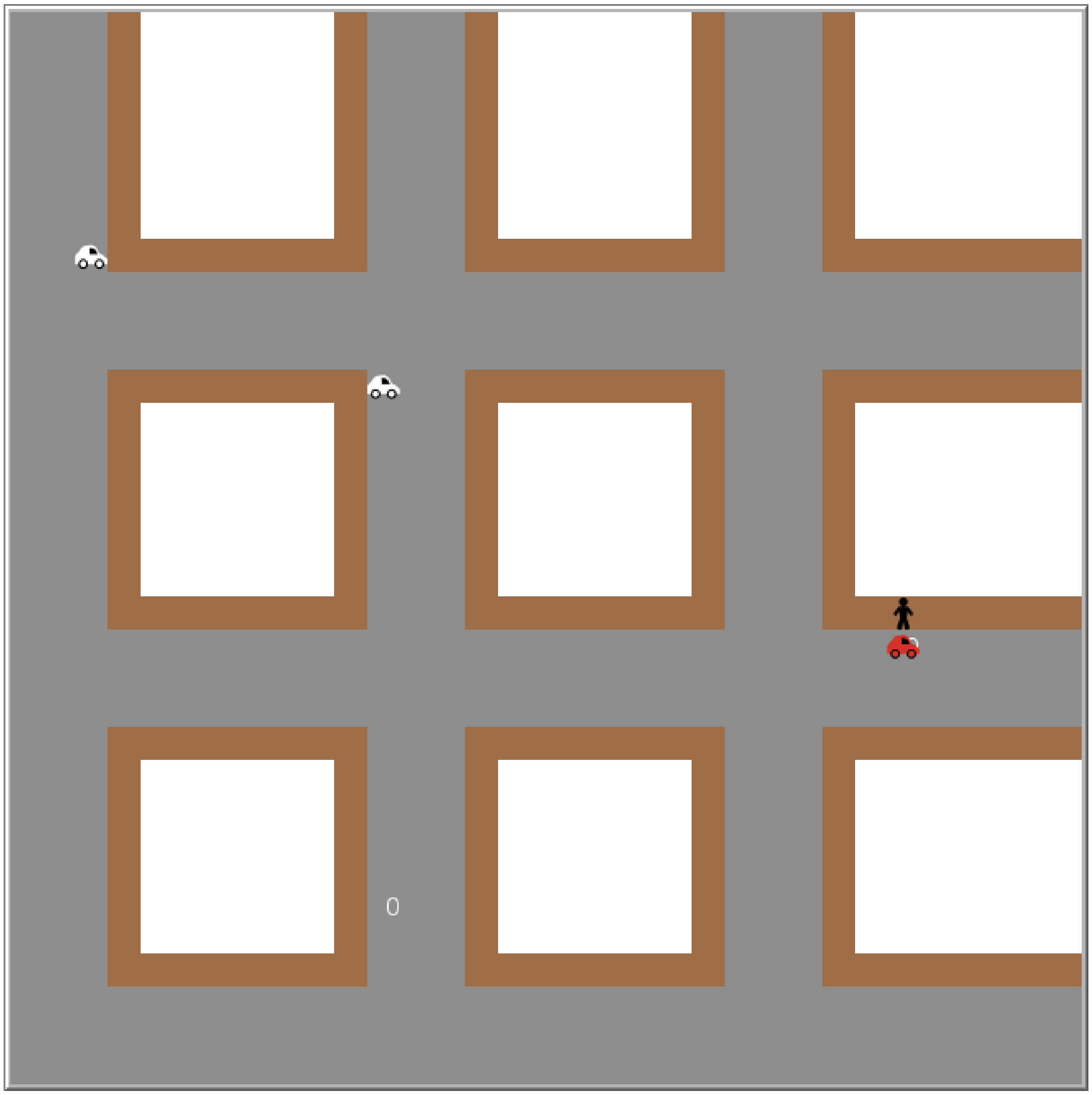


And again the same process continues.


0
0



And finally after reaching the destination, it drops off the passenger and clears the search space.


0

Code Documentation

Global Variables:

```
globals [
  passengers-dist-list  ; List to store distances between cabs and passengers
  openlist              ; List of open patches (nodes) to be evaluated for pathfinding
  closelist             ; List of closed patches (nodes) already evaluated for pathfinding
  goal                  ; Goal patch for pathfinding
  x-goal                ; x-coordinate of the goal patch
  y-goal                ; y-coordinate of the goal patch
  start                 ; Starting patch for pathfinding
  x-start               ; x-coordinate of the starting patch
  y-start               ; y-coordinate of the starting patch
  go-stop               ; Flag to control stopping
  finish?               ; Flag to indicate if the pathfinding process is finished
  delivered?            ; Flag to indicate if a passenger is delivered
  q                     ; Temporary variable for pathfinding
]
```

These global variables are used to manage various aspects of the simulation, such as storing distances, managing open and closed lists for pathfinding, keeping track of the goal and starting points, and controlling the flow of the simulation.

Agent Breeds:

```
breed [ passengers passenger ]  ; Define passenger breed
breed [ cabs cab ]              ; Define cab breed
```

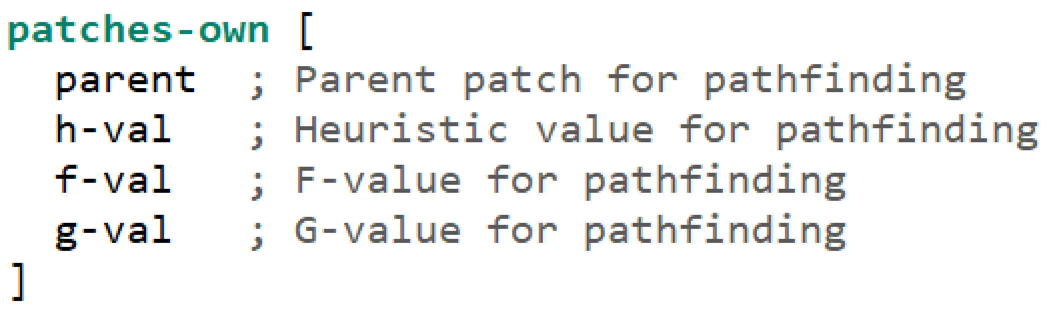
Defines two agent breeds: "passengers" for representing passengers and "cabs" for representing taxi cabs in the simulation.

Passenger and Cab Attributes:

```
passengers-own [
  pickuppoint  ; Patch where passenger is picked up
  dist         ; Distance between cab and passenger
]

cabs-own [
  min-dist        ; Minimum distance between cab and passenger
  cab-pass-dist   ; List of distances between cab and passengers
  pass-on-board?  ; Flag to indicate if a passenger is on board
]
```

Defines the attributes specific to each passenger and cab. For passengers, it includes the pickup point and distance from the cab. For cabs, it includes the minimum distance, list of distances to passengers, and a flag to indicate if a passenger is on board.

Patch Attributes:

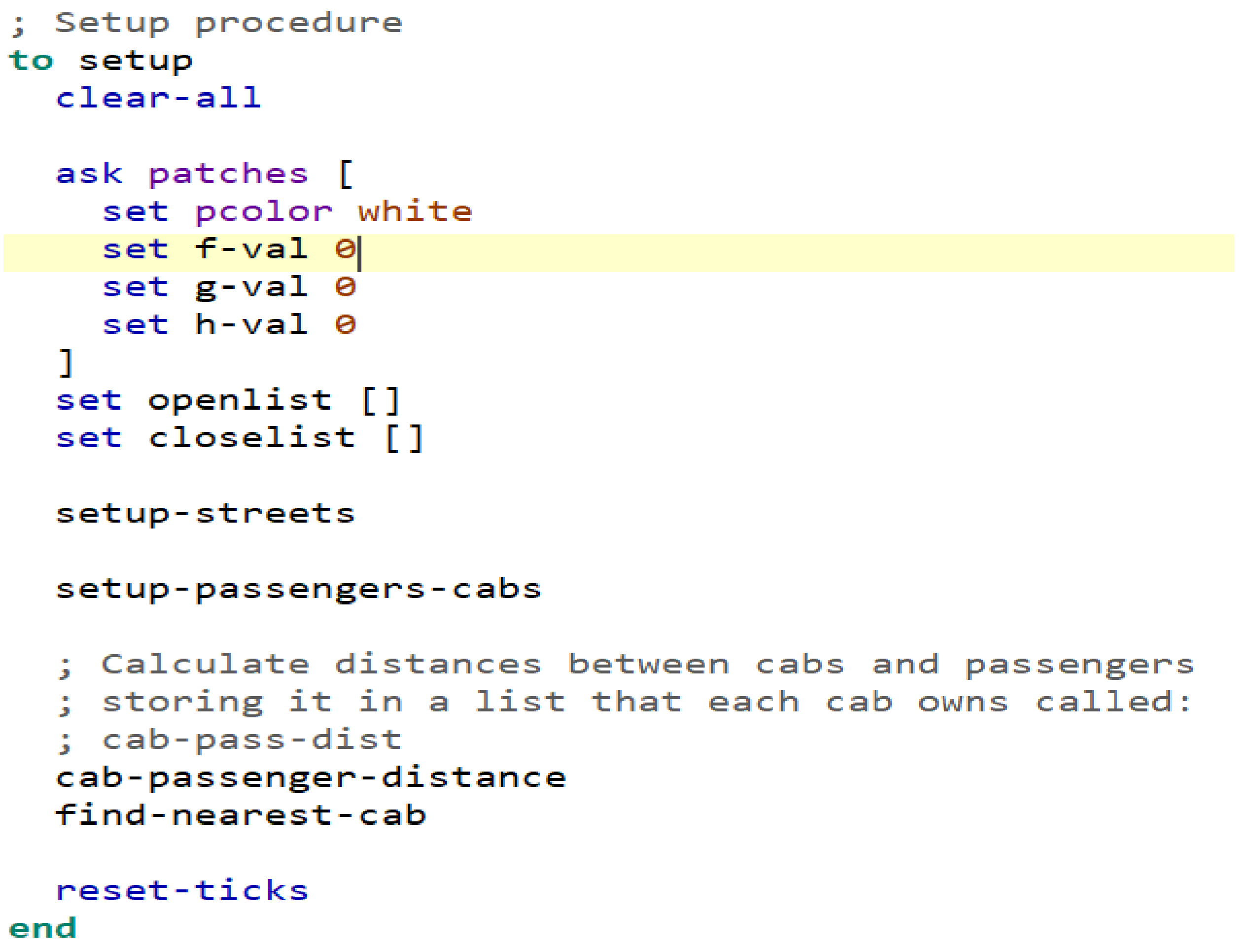


```
patches-own [
  parent   ; Parent patch for pathfinding
  h-val    ; Heuristic value for pathfinding
  f-val    ; F-value for pathfinding
  g-val    ; G-value for pathfinding
]
```

Defines patch attributes used for pathfinding, including the parent patch (used in A* algorithm), heuristic value, F-value (total estimated cost from start to goal), and G-value (actual cost from the start to the current patch).

Setup Procedure:
This procedure initializes the simulation environment. It clears all patches, sets their colors, initializes lists, sets up the streets, creates passengers and cabs, calculates distances between cabs and passengers, and resets ticks.

- **Clears All:** clear-all clears the entire world, removing all agents and resetting all patches to their default state.
- **Sets Patch Attributes:** ask patches [...] iterates over all patches in the world and sets their color to white, and initializes the f-val, g-val, and h-val attributes to 0. This step effectively resets the patches to their default state for the start of a new simulation.
- **Initializes Lists:** set openlist [] and set closelist [] initialize the open and closed lists used for pathfinding.
- **Sets Up Streets:** setup-streets sets up the street layout in the world.
- **Sets Up Passengers and Cabs:** setup-passengers-cabs creates passengers and cabs, placing them in the world at random locations.
- **Calculates Cab-Passenger Distances:** cab-passenger-distance calculates distances between cabs and passengers and stores them in the cab-pass-dist list.
- **Resets Ticks:** reset-ticks resets the tick counter to 0, preparing the simulation to start from the beginning.



```
; Setup procedure
to setup
  clear-all

  ask patches [
    set pcolor white
    set f-val 0
    set g-val 0
    set h-val 0
  ]
  set openlist []
  set closelist []

  setup-streets

  setup-passengers-cabs

  ; Calculate distances between cabs and passengers
  ; storing it in a list that each cab owns called:
  ; cab-pass-dist
  cab-passenger-distance
  find-nearest-cab

  reset-ticks
end
```

Go Procedure:
The main procedure responsible for the simulation's execution. It controls the flow of the simulation, including stopping conditions, picking up passengers, driving cabs, delivering passengers, and advancing the simulation by one tick.

- **Stops if Flag is True:** if go-stop = true [stop] checks if the go-stop flag is true. If it is, the simulation stops.
- **Checks Cabs with Color Black:** if any? cabs with [color = black] [...] checks if there are any cabs with a color of black, indicating they are available for picking up passengers.
    - **Handles Finish:** If finish? is true, it stops the simulation and marks the path from goal to start with magenta color.
    - **Calls Pickup Procedure:** If there are available cabs, it calls the pickup-passenger procedure.
- **Checks Cabs with Color Cyan:** if any? cabs with [color = cyan] [...] checks if there are any cabs with a color of cyan, indicating they are driving.
    - **Calls Drive Procedure:** If there are cabs driving, it calls the drive procedure.

- **Checks Cabs with Color Red:** if any? cabs with [color = red] [...] checks if there are any cabs with a color of red, indicating they are delivering passengers.
    - **Calls Deliver Procedure:** If there are cabs delivering passengers, it calls the deliver procedure.
    - **Calls Drive-to-Deliver Procedure:** It calls the drive-to-delivere procedure to handle driving to deliver the passenger.
- **Ticks:** tick advances the simulation by one tick.

```
to go
  if go-stop = true[stop]


  if any? cabs with[color = black]
  [
    if finish? = true
    [
      ask patch-set closelist [set pcolor yellow]

      let x goal

      while[x != start]
      [
        ask x [ set pcolor magenta ]
        set x [parent] of x
      ]

      ask cabs with[color = black]
      [
        if any? cabs with[color = cyan][stop]
        set color cyan
      ]
      stop
    ]
    pickup-passenger
  ]
  if any? cabs with[color = cyan]
  [
    drive
  ]

  if any? cabs with[color = red]
  [
    deliver
    drive-to-delivere


  ]




  tick
end
```

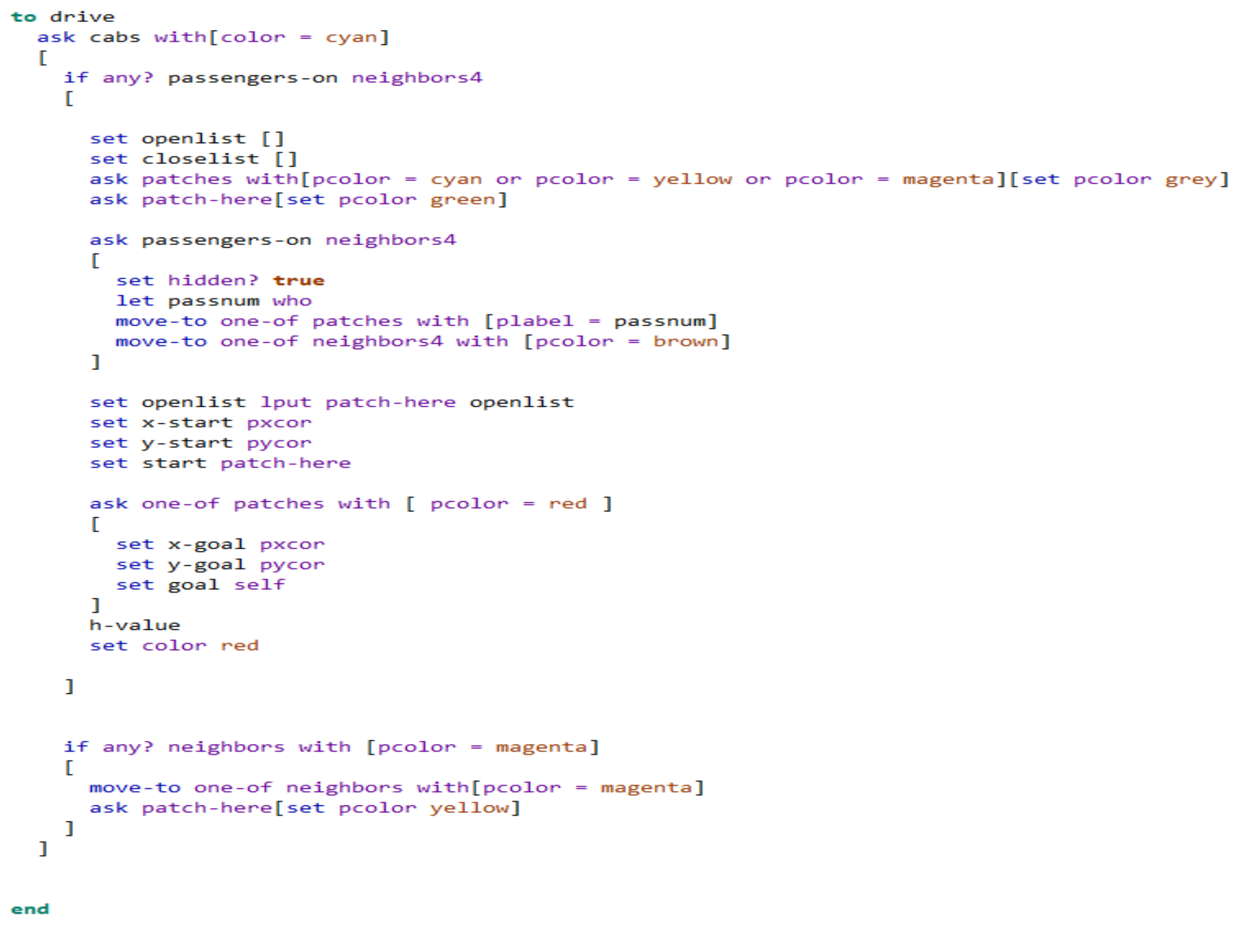
Sub-procedures:

There are sub-procedures called within the main procedures to perform specific tasks, such as driving cabs, picking up passengers, delivering passengers, calculating heuristic values, finding the nearest cab, calculating distances, setting up streets, and setting up passengers and cabs.

Drive

This procedure handles the behavior of cabs that are currently driving. Here's what it does:

- **Checks if Passengers are Nearby:** if any? passengers-on neighbors4 [...] checks if there are any passengers adjacent to the cab.
    - **Handles Pickup:** If there are passengers nearby, it initializes the open and close lists, sets patches' colors, hides the passengers, sets the pickup location, calculates heuristic values, and sets the cab's color to red.
- **Moves to Magenta Patches:** if any? neighbors with [pcolor = magenta] [...] checks if there are neighboring patches with a color of magenta, indicating a path to the destination.
    - **Moves to Magenta:** If such patches exist, the cab moves to one of them and sets the current patch's color to yellow.

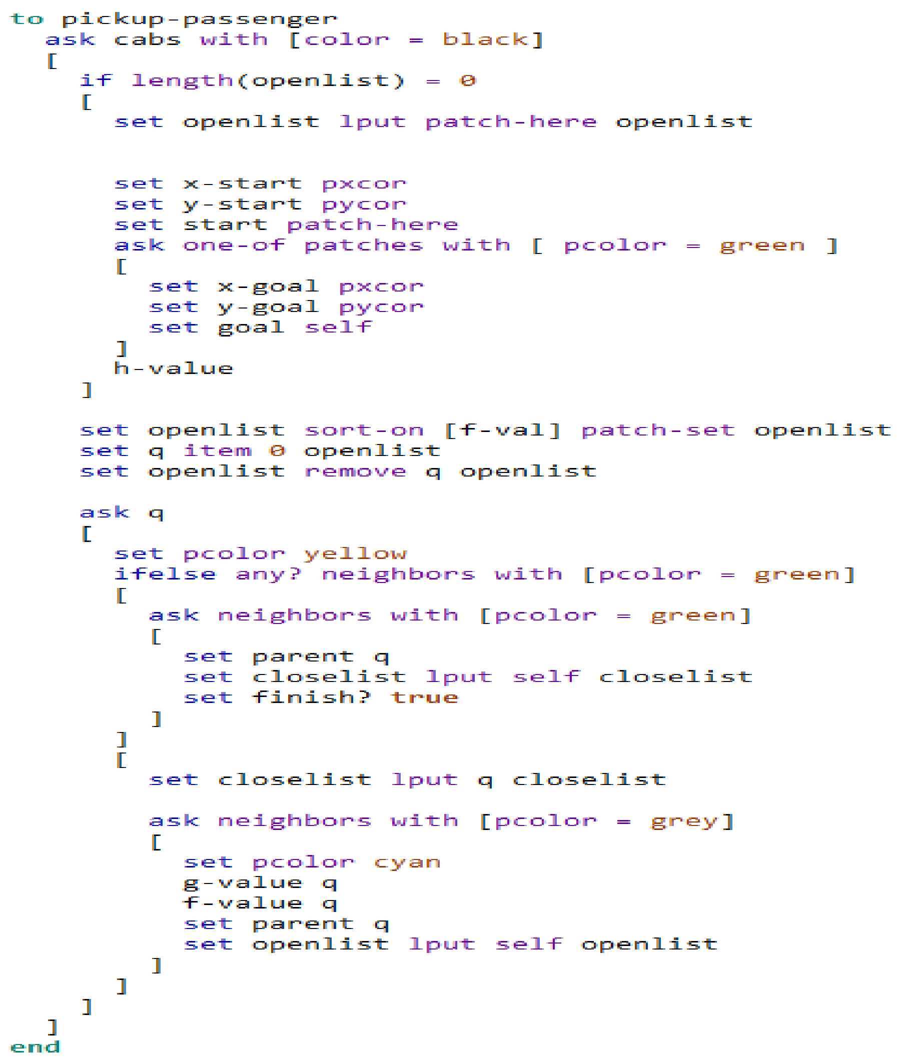


```
to drive
  ask cabs with[color = cyan]
  [
    if any? passengers-on neighbors4
    [

      set openlist []
      set closelist []
      ask patches with[pcolor = cyan or pcolor = yellow or pcolor = magenta][set pcolor grey]
      ask patch-here[set pcolor green]

      ask passengers-on neighbors4
      [
        set hidden? true
        let passnum who
        move-to one-of patches with [plabel = passnum]
        move-to one-of neighbors4 with [pcolor = brown]
      ]

      set openlist lput patch-here openlist
      set x-start pxcor
      set y-start pycor
      set start patch-here

      ask one-of patches with [ pcolor = red ]
      [
        set x-goal pxcor
        set y-goal pycor
        set goal self
      ]
      h-value
      set color red

    ]


    if any? neighbors with [pcolor = magenta]
    [
      move-to one-of neighbors with[pcolor = magenta]
      ask patch-here[set pcolor yellow]
    ]
  ]

end
```

Pickup-passenger

This procedure handles the behavior of cabs that are picking up passengers. Here's what it does:

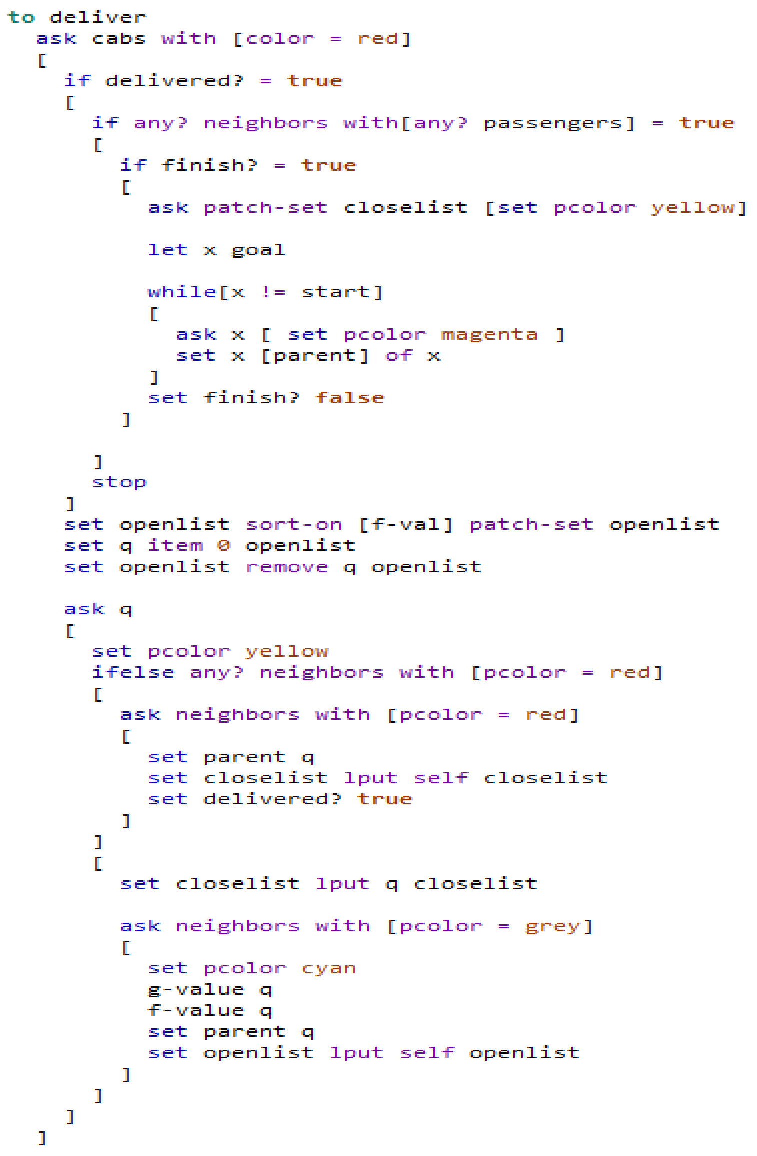
- **Initializes Open List:** If the openlist is empty, it initializes it with the current patch.
- **Sorts Open List:** It sorts the openlist based on the F-value of patches.
- **Selects Patch:** It selects the patch with the lowest F-value from the openlist as the current patch (q).
- **Checks if Goal is Reached:** If the current patch is adjacent to the goal, it sets the goal as reached.
- **Expands Search:** If the goal is not reached, it expands the search by considering neighboring patches.
- **Calculates G-Value and F-Value:** For each neighboring patch, it calculates the G-value (actual cost from the start) and F-value (total estimated cost from start to goal) and updates the lists accordingly.



```
to pickup-passenger
  ask cabs with [color = black]
  [
    if length(openlist) = 0
    [
      set openlist lput patch-here openlist


      set x-start pxcor
      set y-start pycor
      set start patch-here
      ask one-of patches with [ pcolor = green ]
      [
        set x-goal pxcor
        set y-goal pycor
        set goal self
      ]
      h-value
    ]

    set openlist sort-on [f-val] patch-set openlist
    set q item 0 openlist
    set openlist remove q openlist

    ask q
    [
      set pcolor yellow
      ifelse any? neighbors with [pcolor = green]
      [
        ask neighbors with [pcolor = green]
        [
          set parent q
          set closelist lput self closelist
          set finish? true
        ]
      ]
      [
        set closelist lput q closelist

        ask neighbors with [pcolor = grey]
        [
          set pcolor cyan
          g-value q
          f-value q
          set parent q
          set openlist lput self openlist
        ]
      ]
    ]
  ]
end
```

This procedure handles the behavior of cabs that are delivering passengers. Here's what it does:

- **Checks if Passenger is Delivered:** If the passenger is delivered, it checks if there are other passengers nearby.
- **Sorts Open List:** It sorts the openlist based on the F-value of patches.
- **Selects Patch:** It selects the patch with the lowest F-value from the openlist as the current patch (q).
- **Checks for Neighboring Cabs:** If there are neighboring patches with cabs (pcolor = red), it updates the parent, closes the list, and marks the passenger as delivered.
- **Expands Search:** If no neighboring cabs are found, it expands the search by considering neighboring patches.
- **Updates Lists:** It updates the open and close lists and patches colors based on the search.



```
to deliver
  ask cabs with [color = red]
  [
    if delivered? = true
    [
      if any? neighbors with[any? passengers] = true
      [
        if finish? = true
        [
          ask patch-set closelist [set pcolor yellow]

          let x goal

          while[x != start]
          [
            ask x [ set pcolor magenta ]
            set x [parent] of x
          ]
          set finish? false
        ]

      ]
      stop
    ]
    set openlist sort-on [f-val] patch-set openlist
    set q item 0 openlist
    set openlist remove q openlist

    ask q
    [
      set pcolor yellow
      ifelse any? neighbors with [pcolor = red]
      [
        ask neighbors with [pcolor = red]
        [
          set parent q
          set closelist lput self closelist
          set delivered? true
        ]
      ]
      [
        set closelist lput q closelist

        ask neighbors with [pcolor = grey]
        [
          set pcolor cyan
          g-value q
          f-value q
          set parent q
          set openlist lput self openlist
        ]
      ]
    ]
  ]
```

Drive-To-Delivere

This procedure handles the behavior of cabs that are driving to deliver passengers. Here's what it does:

- **Checks for Passengers Nearby:** If there are passengers nearby, it hides patches and sets go-stop to true.
- **Moves to Magenta Patches:** If there are neighboring patches with a color of magenta, indicating a path to the destination, it moves to one of them and sets the current patch's color to yellow.

```
to drive-to-delivere
  ask cabs with[color = red]
  [
    if any? passengers-on neighbors4
    [
      ask patches with[pcolor = cyan or pcolor = yellow or pcolor = magenta][set pcolor grey]
      ask passengers-on neighbors
      [

        set hidden? false
      ]
      set go-stop true
    ]
    if any? neighbors with [pcolor = magenta]
    [
      move-to one-of neighbors with[pcolor = magenta]
      ask patch-here[set pcolor yellow]
    ]
  ]
end
```

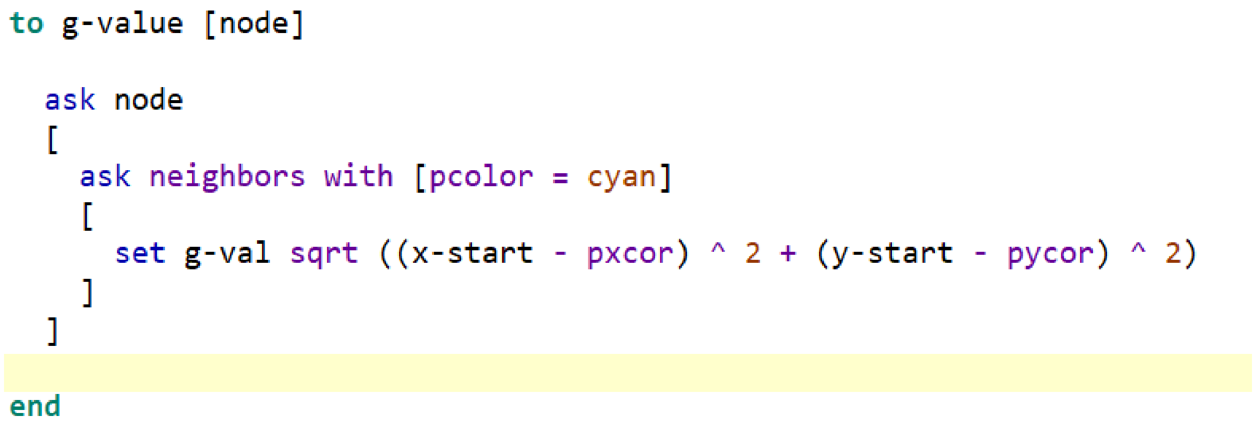
H-value

This procedure calculates the heuristic value (H-value) for each gray patch. It calculates the Euclidean distance between the patch and the goal and sets the h-val attribute accordingly.

```
to h-value

  ask patches with [pcolor = grey]
  [
    set h-val sqrt ((pxcor - x-goal) ^ 2 + (pycor - y-goal) ^ 2)
  ]

end
```

G-value

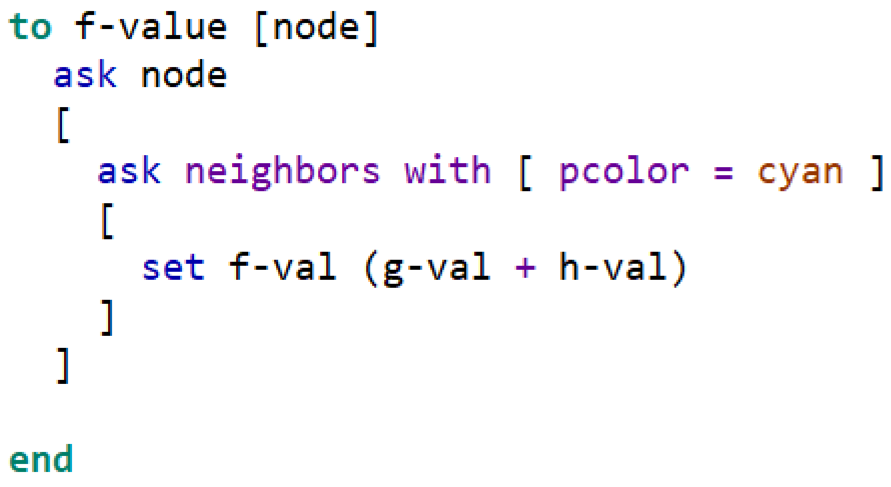
This procedure calculates the actual cost (G-value) from the start to the current patch (node). It calculates the Euclidean distance between the starting patch and the current patch and sets the g-val attribute accordingly.



```
to g-value [node]

  ask node
  [
    ask neighbors with [pcolor = cyan]
    [
      set g-val sqrt ((x-start - pxcor) ^ 2 + (y-start - pycor) ^ 2)
    ]
  ]

end
```

F-value

This procedure calculates the total estimated cost (F-value) from the start to the goal passing through the current patch (node). It adds the G-value and H-value of the patch and sets the f-val attribute accordingly.



```
to f-value [node]
  ask node
  [
    ask neighbors with [ pcolor = cyan ]
    [
      set f-val (g-val + h-val)
    ]
  ]

end
```

Find-Nearest-Cab

This procedure finds the nearest cab to each passenger and marks it with a black color. It calculates the minimum distance between each cab and passengers and marks the nearest cab with a black color.

```
to find-nearest-cab
  if any? cabs with [color = black or color = cyan][stop]
  let nearest min [min-dist] of cabs with[color = white]
  ask cabs with[color = white and min-dist = nearest]
  [
    set color black
  ]
end
```

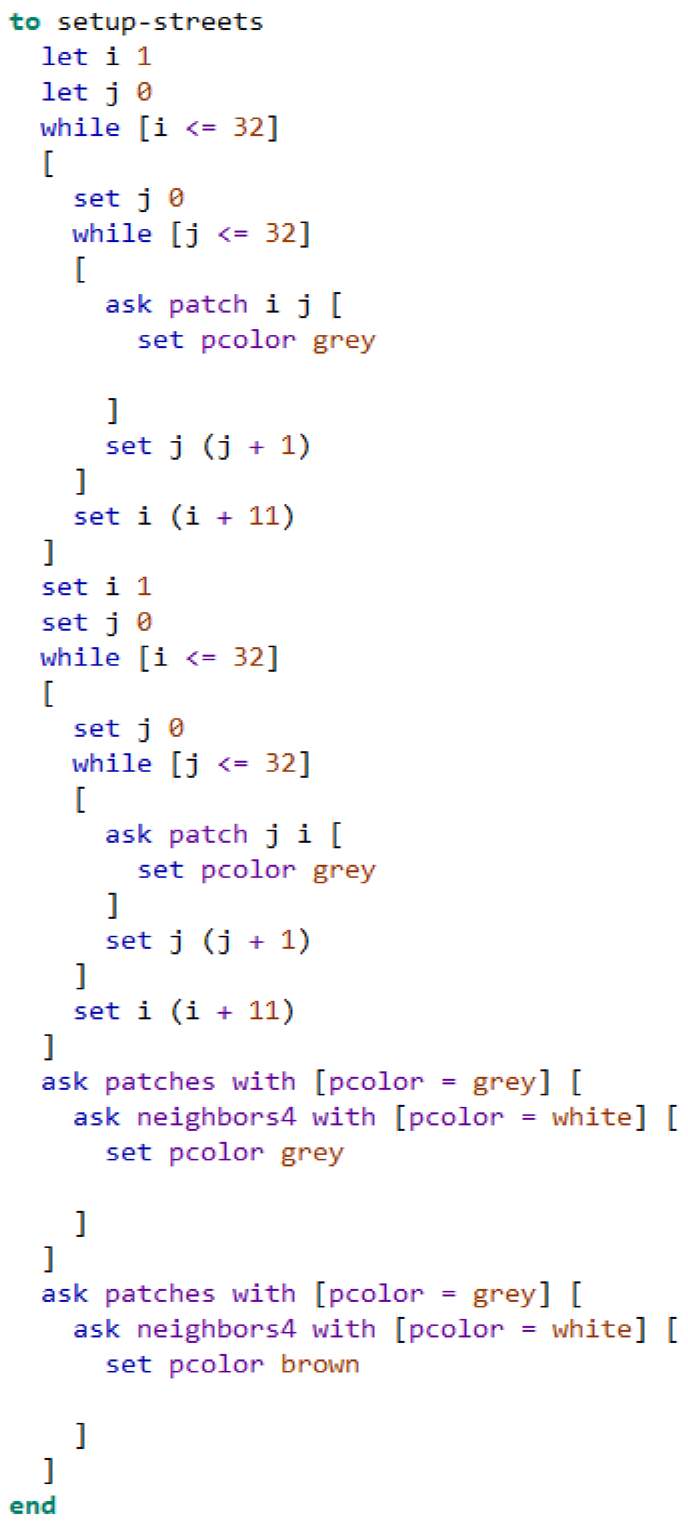
cab-passenger-distance

This procedure calculates the distance between each cab and passengers. It iterates over each cab, calculates the distance to each passenger, and stores it in the cab-pass-dist list.

```
to cab-passenger-distance

  let px 0
  let py 0
  let i 1


  while [ i <= 3 ]
  [
    set passengers-dist-list []
    ask cab i
    [
      set px pxcor
      set py pycor

      ask passengers
      [
        set dist sqrt ((px - pxcor) ^ 2 + (py - pycor) ^ 2)
        set passengers-dist-list lput dist passengers-dist-list
        ]

      let unsorted passengers-dist-list
      let sorted (sort-by < unsorted)
      set cab-pass-dist sorted
      set min-dist item 0 cab-pass-dist
      set cab-pass-dist []
    ]

    set i ( i + 1 )
  ]

end
```

setup-streets

This procedure sets up the street layout in the world. It creates a grid-like street layout using gray patches and marks some patches as brown to indicate potential pickup points.



```
to setup-streets
  let i 1
  let j 0
  while [i <= 32]
  [
    set j 0
    while [j <= 32]
    [
      ask patch i j [
        set pcolor grey

      ]
      set j (j + 1)
    ]
    set i (i + 11)
  ]
  set i 1
  set j 0
  while [i <= 32]
  [
    set j 0
    while [j <= 32]
    [
      ask patch j i [
        set pcolor grey
      ]
      set j (j + 1)
    ]
    set i (i + 11)
  ]
  ask patches with [pcolor = grey] [
    ask neighbors4 with [pcolor = white] [
      set pcolor grey

    ]
  ]
  ask patches with [pcolor = grey] [
    ask neighbors4 with [pcolor = white] [
      set pcolor brown

    ]
  ]
end
```

setup-passengers-cabs

This procedure sets up passengers and cabs in the world. It creates passengers and cabs and places them at random locations on brown patches, representing pickup points and streets.

```
to setup-passengers-cabs

  create-passengers 1 [
    move-to one-of patches with [pcolor = brown]

    set shape "person"
    set color black
    let passenger-number who

    ask one-of patches with[pcolor = brown]
    [
      ask one-of neighbors4 with [pcolor = grey]
      [

        set plabel passenger-number
        set pcolor red

      ]
    ]

    ask one-of neighbors4 with [pcolor = grey]
    [
      set pcolor green
      set plabel passenger-number

    ]
    set pickuppoint one-of neighbors4 with [pcolor = green]
  ]
  create-cabs 3 [
    move-to one-of patches with [pcolor = grey]
    set shape "car"
    set color white
    set pass-on-board? false

  ]
end
```

Each of these sub-procedures plays a crucial role in the simulation, handling specific aspects such as pathfinding, agent behavior, environment setup, and distance calculations. Together, they contribute to the overall functionality and behavior of the simulation.

Overall:

- The code simulates a taxi service environment where cabs pick up passengers from specific locations and drop them off at their destinations.
- A* algorithm is used for pathfinding, with heuristic values calculated to estimate the distance from a patch to the goal.
- Each cab evaluates the nearest passenger, picks them up, and delivers them to their destination.
- The simulation environment is set up with streets, passengers, and cabs.
- The code is organized into procedures and sub-procedures for clarity and modularity.