New Mexico SuperComputing Challenge

Final Report

April 2021

Project Title:Traveling on the Road

Team Number: 57

School Name(s): Saturday Science and Math Academy

Team Members: Isaac Rankin, McLight Emma-Asonye, Kingsley Walker

Sponsor(s): Caia Brown

Project Mentor: Wayne Witzel

Problem Statement

In middle school, my friend (let's call him Jimmy) was always late to class, so much so that he got suspended from the school dance. He lived far away from school so traffic always made the drive long. This is a common experience for many, and sometimes the consequences are greater than getting suspended from the school dance. There are many problems on the road including: traffic, fuel consumption, and safety issues. In 2018, approximately 3,520 minutes, or 58.6 hours per person were spent a year at red lights. Due to traffic, someone can be late or miss an important event. At red lights, cars are still using gas which wastes fuel. The average price of gasoline is around $2.50. About 6 million car accidents occur a year, car crashes are the leading cause of death among young people ages 15-29. These are problems on the road that we can manage computationally in order to improve the driving experience. In this project we seek to analyze these problems using NetLogo.

Description Problem Solving Method

To reduce fuel consumption and decrease travel time we used a NetLogo model that simulates traffic and changed it so that only one car was on the model. We then implemented the A* pathfinding algorithm and added our own algorithm that finds the best path and sees if the lights are red or green. Our algorithm looks at the light ahead and estimates if it will be red or green by the time the car arrives at the intersection. So

if the light will be green, then the car will go to that intersection, so this way you use less fuel and get to your destination faster. We used one-way streets (going West and South), so if the light was red, the car would turn and continue pathfinding on the other road. We chose to use the A* pathfinding algorithm because it is generally said that A* is the best algorithm to solve pathfinding problems among the coding community. This would allow people to be on time for work, school, and other events.

Discussion of how we Verified and Validated our Model

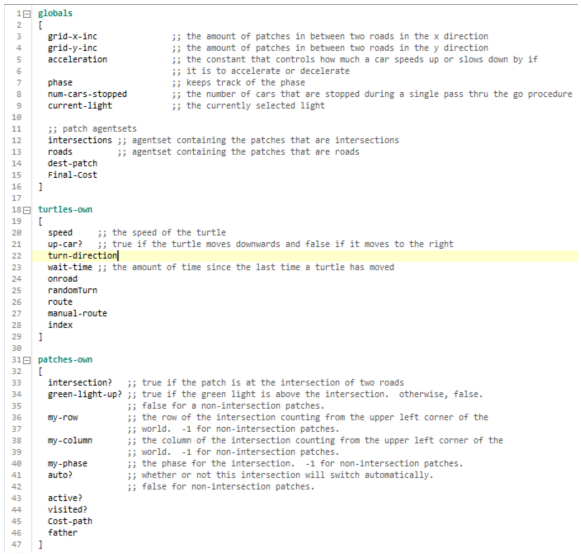
To test our model we chose random spots on the board and made our algorithm and the normal A* algorithm go to the chosen spot. To find how long it took to get to the chosen spot we looked at how many ticks passed until the program ended. Sometimes the normal A* algorithm reached the destination faster than our algorithm, which wasn't supposed to do that, so we had to fix that. This issue was caused by a bug in our algorithm; the paths it took were random if it couldn't find the best path. To fix that, we changed when the next path was determined. After we fixed the bug, our algorithm showed great improvement from before and beat the normal A* algorithm's time every time. Here are our results.

STARTING COORDINATE: (-18, 14)

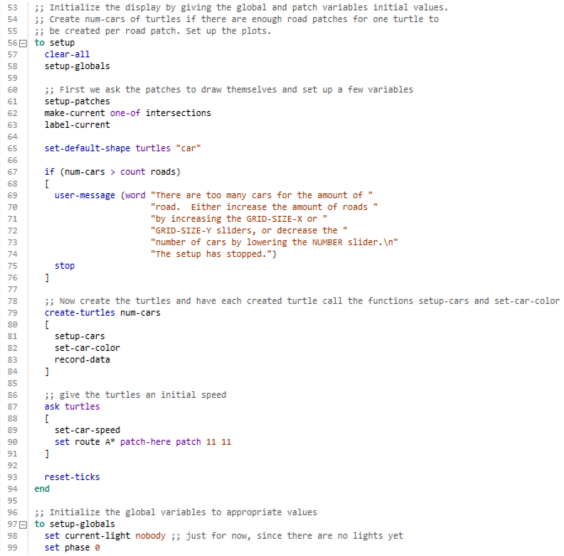
Dest. Coordinate	Our algorithm (time in ticks)	Normal pathfinding (time in ticks)
(18, -18)	212	303
(12, -7)	93	213
(8, -1)	73	161
(-8, -7)	81	122
(-3, 4)	70	154

This chart shows that in far distances and short distances, our algorithm excelled in pathfinding. In 60% of our tests, our car took half the time than normal pathfinding took to reach our destination.

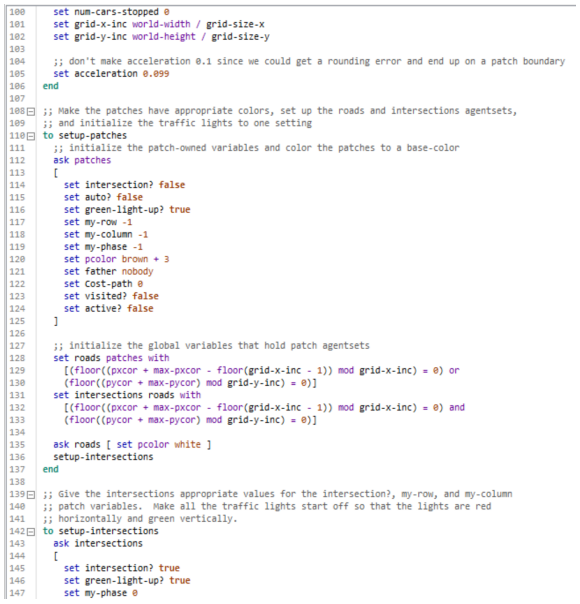
Code to Date (Used in NetLogo, 627 lines)



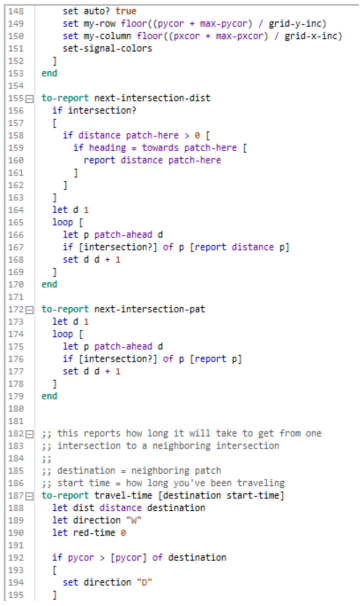
```
globals
[
  grid-x-inc               ;; the amount of patches in between two roads in the x direction
  grid-y-inc               ;; the amount of patches in between two roads in the y direction
  acceleration             ;; the constant that controls how much a car speeds up or slows down by if
                           ;; it is to accelerate or decelerate
  phase                    ;; keeps track of the phase
  num-cars-stopped         ;; the number of cars that are stopped during a single pass thru the go procedure
  current-light            ;; the currently selected light

  ;; patch agentsets
  intersections ;; agentset containing the patches that are intersections
  roads         ;; agentset containing the patches that are roads
  dest-patch
  Final-Cost
]

turtles-own
[
  speed     ;; the speed of the turtle
  up-car?   ;; true if the turtle moves downwards and false if it moves to the right
  turn-direction
  wait-time ;; the amount of time since the last time a turtle has moved
  onroad
  randomTurn
  route
  manual-route
  index
]

patches-own
[
  intersection?   ;; true if the patch is at the intersection of two roads
  green-light-up? ;; true if the green light is above the intersection.  otherwise, false.
                  ;; false for a non-intersection patches.
  my-row          ;; the row of the intersection counting from the upper left corner of the
                  ;; world.  -1 for non-intersection patches.
  my-column       ;; the column of the intersection counting from the upper left corner of the
                  ;; world.  -1 for non-intersection patches.
  my-phase        ;; the phase for the intersection.  -1 for non-intersection patches.
  auto?           ;; whether or not this intersection will switch automatically.
                  ;; false for non-intersection patches.
  active?
  visited?
  Cost-path
  father
]
```



```
;; Initialize the display by giving the global and patch variables initial values.
;; Create num-cars of turtles if there are enough road patches for one turtle to
;; be created per road patch. Set up the plots.
to setup
  clear-all
  setup-globals

  ;; First we ask the patches to draw themselves and set up a few variables
  setup-patches
  make-current one-of intersections
  label-current

  set-default-shape turtles "car"

  if (num-cars > count roads)
  [
    user-message (word "There are too many cars for the amount of "
                       "road.  Either increase the amount of roads "
                       "by increasing the GRID-SIZE-X or "
                       "GRID-SIZE-Y sliders, or decrease the "
                       "number of cars by lowering the NUMBER slider.\n"
                       "The setup has stopped.")
    stop
  ]

  ;; Now create the turtles and have each created turtle call the functions setup-cars and set-car-color
  create-turtles num-cars
  [
    setup-cars
    set-car-color
    record-data
  ]

  ;; give the turtles an initial speed
  ask turtles
  [
    set-car-speed
    set route A* patch-here patch 11 11
  ]

  reset-ticks
end

;; Initialize the global variables to appropriate values
to setup-globals
  set current-light nobody ;; just for now, since there are no lights yet
  set phase 0
```



```
    set num-cars-stopped 0
    set grid-x-inc world-width / grid-size-x
    set grid-y-inc world-height / grid-size-y

    ;; don't make acceleration 0.1 since we could get a rounding error and end up on a patch boundary
    set acceleration 0.099
  end

;; Make the patches have appropriate colors, set up the roads and intersections agentsets,
;; and initialize the traffic lights to one setting
to setup-patches
  ;; initialize the patch-owned variables and color the patches to a base-color
  ask patches
  [
    set intersection? false
    set auto? false
    set green-light-up? true
    set my-row -1
    set my-column -1
    set my-phase -1
    set pcolor brown + 3
    set father nobody
    set Cost-path 0
    set visited? false
    set active? false
  ]

  ;; initialize the global variables that hold patch agentsets
  set roads patches with
    [(floor((pxcor + max-pxcor - floor(grid-x-inc - 1)) mod grid-x-inc) = 0) or
    (floor((pycor + max-pycor) mod grid-y-inc) = 0)]
  set intersections roads with
    [(floor((pxcor + max-pxcor - floor(grid-x-inc - 1)) mod grid-x-inc) = 0) and
    (floor((pycor + max-pycor) mod grid-y-inc) = 0)]

  ask roads [ set pcolor white ]
  setup-intersections
end

;; Give the intersections appropriate values for the intersection?, my-row, and my-column
;; patch variables.  Make all the traffic lights start off so that the lights are red
;; horizontally and green vertically.
to setup-intersections
  ask intersections
  [
    set intersection? true
    set green-light-up? true
    set my-phase 0
```



```
    set auto? true
    set my-row floor((pycor + max-pycor) / grid-y-inc)
    set my-column floor((pxcor + max-pxcor) / grid-x-inc)
    set-signal-colors
  ]
end

to-report next-intersection-dist
  if intersection?
  [
    if distance patch-here > 0 [
      if heading = towards patch-here [
        report distance patch-here
      ]
    ]
  ]
  let d 1
  loop [
    let p patch-ahead d
    if [intersection?] of p [report distance p]
    set d d + 1
  ]
end

to-report next-intersection-pat
  let d 1
  loop [
    let p patch-ahead d
    if [intersection?] of p [report p]
    set d d + 1
  ]
end


;; this reports how long it will take to get from one
;; intersection to a neighboring intersection
;;
;; destination = neighboring patch
;; start time = how long you've been traveling
to-report travel-time [destination start-time]
  let dist distance destination
  let direction "W"
  let red-time 0

  if pycor > [pycor] of destination
  [
    set direction "D"
  ]
```

```
    if pxcor < [pxcor] of destination
    [
      set direction "R"
    ]

    ;; when we would get to the next intersection with a green light
    let green-time (dist / speed-limit) + start-time

    ;; how long the wait time at the intersection is
    if [intersection?] of destination
    [
      set red-time [wait-time-at-intersection green-time direction] of destination
    ]

    report green-time + red-time - start-time
  end

  ;; this returns the amount of time the car has to
  ;; wait until the light ahead of it turns green
  ;;
  ;; time = time when we get to intersection
  ;; direction = direction that the car is heading (R and D)
  to-report wait-time-at-intersection [time direction]
    ;;green-light-up?
    ;;my-phase
    ;;phase

    ;; shows if the light ahead in the direction you're traveling
    ;; is green as opposed to if the light ahead is up or down
    let future-green-light? false

    ;; Shows how many times the light will change until the car gets there
    let number-of-light-changes floor((phase + time) / ticks-per-cycle)

    ;; what the phase will be at the given time
    let future-phase (phase + time) mod ticks-per-cycle

    ;; shows if the light ahead will be green going vertically or horizontally
    let future-green-light-up? green-light-up?
    if (number-of-light-changes mod 2) = 1
    [
      set future-green-light-up? not green-light-up?
    ]

    ;; sets future green light to the opposite of
    ;; future green light up when it's going right
    if (direction = "R")
    [
      set future-green-light? not future-green-light-up?
    ]

    ;; sets future green light to future green light when it's going down
    if (direction = "D")
    [
      set future-green-light? future-green-light-up?
    ]

    ;; if future green light is true then the car doesn't have to wait at a red
    ;; light. Therefore the time is 0
    ifelse (future-green-light?)
    [
      report 0
    ]
    [
      report ticks-per-cycle - future-phase
    ]

  end

  to-report Total-expected-cost [#Goal]
    report Cost-path + Heuristic #Goal
  end

  to-report Heuristic [#Goal]
    report distance #Goal
  end

  to-report one-way-neighbors
    report (patch-set patch-at 1 0 patch-at 0 -1)
  end

  ;; Fernando Sancho Caparrini
  to-report A* [#Start #Goal]
    ; clear all the information in the agents, and reset them
    ask roads with [visited?]
    [
      set father nobody
      set Cost-path 0
      set visited? false
      set active? false
    ]
    ; Active the starting point to begin the searching loop
    ask #Start
    [
      set father self
      set visited? true
      set active? true
    ]

    ; exists? indicates if in some instant of the search there are no options to continue.
    ; In this case, there is no path connecting #Start and #Goal
    let exists? true
    ; The searching loop is executed while we don't reach the #Goal and we think a path exists
    while [not [visited?] of #Goal and exists?]
    [
      ; We only work on the valid pacthes that are active
      let options roads with [active?]
      ; If any
      ifelse any? options
      [
        ; Take one of the active patches with minimal expected cost
        ask min-one-of options [Total-expected-cost #Goal]
        [
          ; Store its real cost (to reach it) to compute the real cost of its children
          let Cost-path-father Cost-path
          ; and deactivate it, because its children will be computed right now
          set active? false
          ; Compute its valid neighbors and look for an extension of the path
          let valid-neighbors one-way-neighbors with [member? self roads]
          ask valid-neighbors
          [
            ; There are 2 types of valid neighbors:
            ;   - Those that have never been visited (therefore, the path we are building is the
            ;       best for them right now)
            ;   - Those that have been visited previously (therefore we must check if the path we
            ;       are building is better or not, by comparing its expected length with the one
            ;       stored in the patch)
            ; One trick to work with both type uniformly is to give for the first case an upper
            ;   bound big enough to be sure that the new path will always be smaller.
            let t ifelse-value visited? [ Total-expected-cost #Goal] [2 ^ 20]
            ; If this temporal cost is worse than the new one, we substitute the information in
            ;   the patch to store the new one (with the neighbors of the first case, it will be
            ;   always the case)
            let neighbor self
            set father myself
            let neighbor-cost [travel-time neighbor Cost-path-father] of father
            if t > (Cost-path-father + Heuristic #Goal + neighbor-cost)
            [
              ; The current patch becomes the father of its neighbor in the new path
              ; set father myself
              set visited? true
              set active? true
              ; and store the real cost in the neighbor from the real cost of its father
              set Cost-path Cost-path-father + neighbor-cost
              set Final-Cost precision Cost-path 3
            ] ] ]
        ; If there are no more options, there is no path between #Start and #Goal
        [
          set exists? false
        ] ]
      ; After the searching loop, if there exists a path
      ifelse exists?
      [
        ; We extract the list of patches in the path, form #Start to #Goal by jumping back from
        ;   #Goal to #Start by using the fathers of every patch
        let current #Goal
        set Final-Cost (precision [Cost-path] of #Goal 3)
        let rep (list current)
        While [current != #Start]
        [
          set current [father] of current
          set rep fput current rep
        ]
        report rep
      ]
      [
        ; Otherwise, there is no path, and we return False
        report false
      ]
    end

    ;; Initialize the turtle variables to appropriate values and place the turtle on an empty road patch.
    to setup-cars  ;; turtle procedure
      set speed 0
      set wait-time 0
      set onroad true
      put-on-empty-road
      set up-car? false
      set turn-direction "R"

      ;ifelse intersection?
      ;[
      ;  ifelse random 2 = 0
      ;  [ set up-car? true ]
      ;  [ set up-car? false ]
      ;]
      ;[
        ; if the turtle is on a vertical road (rather than a horizontal one)
      ;  ifelse (floor((pxcor + max-pxcor - floor(grid-x-inc - 1)) mod grid-x-inc) = 0)
      ;  [ set up-car? true ]
      ;  [ set up-car? false ]
      ;]
```

```
  ;ifelse up-car?
  ;[ set heading 180 ]
  set heading 90
  if xcor > 20 or ycor < -20
  [set onroad true]

end

;; Find a road patch without any turtles on it and place the turtle there.
to put-on-empty-road  ;; turtle procedure
  setxy -18 12
end


;;;;;;;;;;;;;;;;;;;;;;;;
;; Runtime Procedures ;;
;;;;;;;;;;;;;;;;;;;;;;;;

;; Run the simulation
to go
  update-current

    ;; have the intersections change their color

    set num-cars-stopped 0

    ;; set the turtles speed for this time thru the procedure, move them forward their speed,
    ;; record data for plotting, and set the color of the turtles to an appropriate color
    ;; based on their speed
    ask turtles [



      ;;Change the direction that the car is facing
      if turn-direction = "R"
      [ set heading 90 ]
      if turn-direction = "L"
      [ set heading 270 ]
      if turn-direction = "U"
      [ set heading 0 ]
      if turn-direction = "D"
      [set heading 180]



      set-car-speed
      let nxt-int next-intersection-dist
      ifelse speed > nxt-int
      [

        fd nxt-int
        ;;set up-car? not up-car?
        ;;set index index + 1
        while [(item 0 route) != patch-here]
        [
          set route remove-item 0 route
        ]

;       set manual-route "RRRDD"
;
;       if item index manual-route = "R"
;         [set turn-direction "R"]
;       if item index manual-route = "D"
;         [set turn-direction "D"]

        ;;if item index route = 0
        ;;[set up-car? not up-car?]

        if [pxcor] of item 1 route > [pxcor] of item 0 route
          [set turn-direction "R"]

        if [pycor] of item 1 route < [pycor] of item 0 route
          [set turn-direction "D"]

        set index index + 1

      ;;Randomize turns idk
      ;;  set randomTurn random 2
      ;;  if randomTurn = 0
      ;;  [set up-car? not up-car?]

      ]
      [
        fd speed
      ]
      record-data
      set-car-color
    ]

    ;; update the phase and the global clock
    next-phase
    set-signals
    tick
  ;]
end


to choose-current
  if mouse-down?
  [
    let x-mouse mouse-xcor
    let y-mouse mouse-ycor
    if [intersection?] of patch x-mouse y-mouse
    [
      update-current
      unlabel-current
      make-current patch x-mouse y-mouse
      label-current
      stop
    ]
  ]
end

;; Set up the current light and the interface to change it.
to make-current [light]
  set current-light light
  set current-phase [my-phase] of current-light
  set current-auto? [auto?] of current-light
end

;; update the variables for the current light
to update-current
  ask current-light [
    set my-phase current-phase
    set auto? current-auto?
  ]
end

;; label the current light
to label-current
  ask current-light
  [
    ask patch-at -1 1
    [
      set plabel-color black
      set plabel "current"
    ]
  ]
end

;; unlabel the current light (because we've chosen a new one)
to unlabel-current
  ask current-light
  [
    ask patch-at -1 1
    [
      set plabel ""
    ]
  ]
end

;; have the traffic lights change color if phase equals each intersections' my-phase
to set-signals
  ask intersections with [auto? and phase = floor ((my-phase * ticks-per-cycle) / 100)]
  [
    set green-light-up? (not green-light-up?)
    set-signal-colors
  ]
end

;; This procedure checks the variable green-light-up? at each intersection and sets the
;; traffic lights to have the green light up or the green light to the left.
to set-signal-colors  ;; intersection (patch) procedure
  ifelse power?
  [
    ifelse green-light-up?
    [
      ask patch-at -1 0 [ set pcolor red ]
      ask patch-at 0 1 [ set pcolor green ]
    ]
    [
      ask patch-at -1 0 [ set pcolor green ]
      ask patch-at 0 1 [ set pcolor red ]
    ]
  ]
  [
    ask patch-at -1 0 [ set pcolor white ]
    ask patch-at 0 1 [ set pcolor white ]
  ]
end

;to up-until-intersection [speed]


;; set the turtles' speed based on whether they are at a red traffic light or the speed of the
;; turtle (if any) on the patch in front of them
to set-car-speed  ;; turtle procedure

  ifelse pcolor = red
  [set speed 0]
  [
    ;;Right
    if turn-direction = "R"
```

```
    [set-speed 1 0]
    ;;Left
    if turn-direction = "L"
    [set-speed -1 0]
    ;;Up
    if turn-direction = "U"
    [set-speed 0 1]
    ;;Down
    if turn-direction = "D"
    [set-speed 0 -1]
  ]

  ;;ifelse pcolor = red
  ;;[set speed 0]
  ;;[
  ;;  ifelse up-car?
  ;;  [ set-speed 0 -1 ]
  ;;  [ set-speed 1 0 ]
  ;;]

end



;; set the speed variable of the car to an appropriate value (not exceeding the
;; speed limit) based on whether there are cars on the patch in front of the car
to set-speed [ delta-x delta-y ]  ;; turtle procedure
  ;; get the turtles on the patch in front of the turtle
  let turtles-ahead turtles-at delta-x delta-y

  ;; if there are turtles in front of the turtle, slow down
  ;; otherwise, speed up
  ifelse any? turtles-ahead
  [
    ifelse any? (turtles-ahead with [ up-car? != [up-car?] of myself ])
    [
      set speed 0
    ]
    [
      set speed [speed] of one-of turtles-ahead
      slow-down
    ]
  ]
  [ speed-up ]
end

;; decrease the speed of the turtle
to slow-down  ;; turtle procedure
```

```
  ifelse speed <= 0  ;;if speed < 0
  [ set speed 0 ]
  [ set speed speed - acceleration ]
end

;; increase the speed of the turtle
to speed-up  ;; turtle procedure
  ifelse speed > speed-limit
  [ set speed speed-limit ]
  [ set speed speed + acceleration ]
end

;; set the color of the turtle to a different color based on how fast the turtle is moving
to set-car-color  ;; turtle procedure
  ifelse speed < (speed-limit / 2)
  [ set color blue ]
  [ set color cyan - 2 ]
end

;; keep track of the number of stopped turtles and the amount of time a turtle has been stopped
;; if its speed is 0
to record-data  ;; turtle procedure
  ifelse speed = 0
  [
    set num-cars-stopped num-cars-stopped + 1
    set wait-time wait-time + 1
  ]
  [ set wait-time 0 ]
end

to change-current
  ask current-light
  [
    set green-light-up? (not green-light-up?)
    set-signal-colors
  ]
end

;; cycles phase to the next appropriate value
to next-phase
  ;; The phase cycles from 0 to ticks-per-cycle, tFhen starts over.
  set phase phase + 1
  if phase mod ticks-per-cycle = 0
    [ set phase 0 ]
end
```

The Conclusions We Reached by Analyzing Our Results

If Jimmy had used our algorithm, he would have avoided the red lights and would not have been suspended. For others who are also late often, or have gas problems, this would help them. Through our tests, we found that by avoiding red lights, our traveling time was greatly reduced. While pathfinding is good for getting to your destination, pathfinding and adjusting to traffic lights is even better.

Most Significant Achievement on the Project

“I think getting the cars to go a different direction than up and down was a big achievement. It doesn’t seem like something bug but we spent a lot of time trying to get that to work” - Isaac

“When we made the car go down and right instead of just going down, or just going right” - McLight

Acknowledgment of the People and Organizations that Helped Us

Caia Brown
Wayne Witzel
Fernando Sancho Caparrini
NetLogo