Design Project 2 Document

This documentation refers to the code version ‘prototype’ of the route planning app designed in project 1.

Pathfinding Algorithm

I have been given the responsibility of designing the pathfinding algorithm that will be used to display the route that is being taken.

You can refer to the project 1 documentation to see sketches on how this is designed, but for now I will describe it as such:

Similar to other route planning apps such as google maps, you will be able to enter a start and end destination, then a list of available routes that you can select from and then this route and the order in which you take it will be displayed.

Now there are multiple ways this could be done, one simple way is this:

Method 1

A Racket GUI element will have two drop down menus (start & end point respectively) that contain lists of stations to other stations, the user can select a start point and end point and then this will display a route on a ‘map’ and the order in which to follow this path.

The callback on this will be a simple algorithm that for every combination of route, such as for example stations A, B, C, D, there will be a specific route plotted to the map.

If the user enters start point A, end point D, then my code will plot specific points that show the route that must be taken from A to D. If there is no direct connection from A to D, and instead A to C and C to D, I can write code so that a line is drawn from A to C and C to D. I can also make a predefined time for each of these paths to display and the order to take the route in with text.

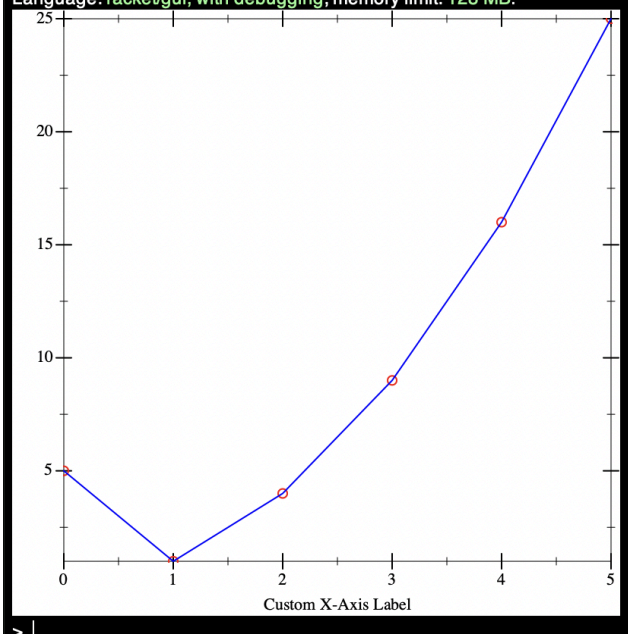
Now this works, but it isn’t exactly efficient. If there are many many stations, you would have to individually code a callback for every single combination of start and end point, the route to be displayed and the time. It also means that this cannot respond to updates to these times or routes as opposed to reading from a set of station connections, as this code would have to be manually updated every time to represent these changes.

Below are two examples of how the “Map” Could be done.

You could create a graph and plot points for “Stations” and draw lines between them to represent a route.

```
1  #lang racket/gui
2  (require plot)
3
4  (define xs '(0 1 2 3 4 5))
5  (define ys '(5 1 4 9 16 25))
6
7  (plot
8    (list
9      (points (map vector xs ys) #:color 'red)
10     (lines (map vector xs ys) #:color 'blue))
11   #:x-label "Custom X-Axis Label"
12   #:y-label "")
13
```

```
Welcome to DrRacket, version 8.10 [cs].
Language: racket/gui, with debugging; memory limit: 128 MB.
```



Or you could use the draw function to draw text and lines to represent the routes, showing and hiding these as necessary.

```
#lang racket/gui

(define (draw-A dc x y)
  (send dc draw-text "A" x y))

(define (draw-B dc x y)
  (send dc draw-text "B" x y))

(define (on-paint canvas dc)
  (draw-A dc 50 50)
  (draw-B dc 100 50)
  (send dc draw-line 60 60 100 60))

(define frame (new frame%
                   [label "A and B with Line"]
                   [width 500]
                   [height 500]))

(define canvas (new canvas%
                    [parent frame]
                    [paint-callback on-paint]))

(send frame show #t)
```


Method 2

The appropriate way to do this would be to create a pathfinding algorithm. What I mean by this is an algorithm that takes in a set or list of ‘stations’ and searches through their connections in order to find an appropriate path if possible.

For example my list may be (A B) (B C) (C D), and my pathfinding algorithm will see that A connects to B, B connects to C and C connects to D, meaning a path from A to D is possible, and the path is A B C D.

You have to be careful though, the code and list must work together as if the list isn’t suited to the code, the pathfinding algorithm may not be able to read it correctly, and not display possible paths.

As well as this, the path must work in both directions. Unless the route has a one-way system, then if A to D is possible, D to A must also be possible.

Now in more complex versions of this there will be many more stations, this means more paths to take. This is where weight comes into effect, by giving each transition from point to point a weight, an algorithm such as dijkstra's can be used to calculate the shortest path from point to point

Using this we now have a graph of stations, a weight or “time” between them and a way to find the route to take.

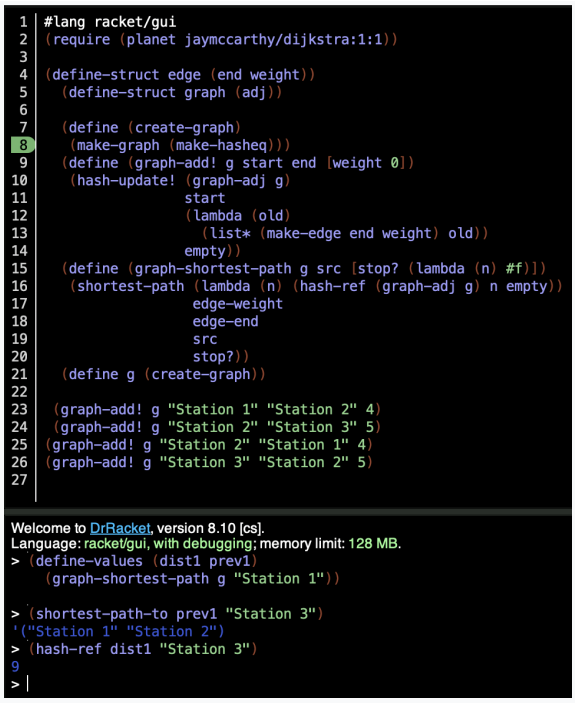
You can then use this algorithm to check whether two imputed points (our start & end) are in the graph, are reachable from each other, and the weight and order of the path, which can be displayed for our user.

This also means that as long as the graph is coded correctly, any set of stations can be plugged in with their weight and a path and time taken can be calculated. This allows the algorithm to be updated easily and even function with realtime information

What I'm doing

There is a racket package that provides an implementation of Djikstras in racket. " Dijkstra's algorithm to find the shortest path between *a* and *b*. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. "

This is what I'll be using to create my pathfinding algorithm.



As you can see in this image, the first line of code creates two hash tables "dist1" and "prev1", these contain a hashtable of the shortest distance from "Station 1" to every other node in the graph, and the total distance (weight) from "Station 1" to every other node in the graph

```
#lang racket
(require (planet jaymccarthy/dijkstra:1:1))

(define-struct edge (end weight))
  (define-struct graph (adj))

  (define (create-graph)
   (make-graph (make-hasheq)))
  (define (graph-add! g start end [weight 0])
   (hash-update! (graph-adj g)
                 start
                 (lambda (old)
                   (list* (make-edge end weight) old))
                 empty))
  (define (graph-shortest-path g src [stop? (lambda (n) #f)])
   (shortest-path (lambda (n) (hash-ref (graph-adj g) n empty))
                  edge-weight
                  edge-end
                  src
                  stop?))
  (define g (create-graph))

(graph-add! g "Station 1" "Station 2" 4)
(graph-add! g "Station 2" "Station 3" 5)
(graph-add! g "Station 3" "Station 4" 6)
(graph-add! g "Station 2" "Station 1" 4)
(graph-add! g "Station 3" "Station 2" 5)
(graph-add! g "Station 4" "Station 3" 6)

(define (print-shortest-path g src dest)
  (define-values (dist prev) (graph-shortest-path g src))
  (define path (shortest-path-to prev dest))
  (define total-time (hash-ref dist dest))
  (display "Your Route is ")
  (display path)
  (display ", then your final stop is ")
  (display dest)
  (display ". The total trip time is ")
  (display total-time)
  (display " minutes")
  (newline))

(define routeplanner (λ (x y) (print-shortest-path g x y)))

```

```
Welcome to DrRacket, version 8.10 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (routeplanner "Station 1" "Station 4")
Your Route is (Station 1 Station 2 Station 3), then your final stop is Station 4. The total trip time is 15 minutes
> (routeplanner "Station 4" "Station 1")
Your Route is (Station 4 Station 3 Station 2), then your final stop is Station 1. The total trip time is 15 minutes
>
```

What's happening here?

The function print-shortest-path is taking 3 arguments, g (the graph) src (the source node) and dest (the destination node).

It calculates the shortest path using Djikstra's algorithm (graph-shortest-path function) which returns two values: dist (the distance mapping hash table) and prev (the shortest path mapping hash table). It does this for whatever the src node is as now these hash tables can be used to find the shortest path and distance to whatever the end node is.

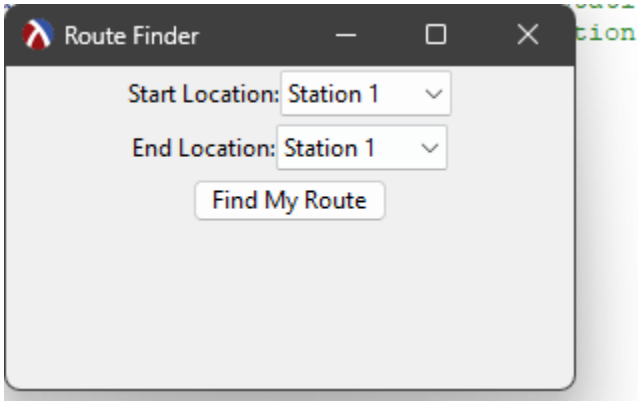
It then uses the shortest-path-to function for the path with dest as the argument, and hash-ref with dest as the argument for the time

After this, the function prints out the path and trip time. This can now be used in conjunction with a GUI, or a representation of a map, or both.

GUI

```
;;;;;;;;;;;;;;;GUI;;;;;;;;;;;;;;;;;;;;;;;;;;
(define frame (new frame%
                   [label "Route Finder"]
                   [width 300]
                   [height 200]))

(define panel (new vertical-panel%
                   [parent frame]))

(define start-locations '( "Station 1" "Station 2" "Station 3" "Station 4"))
(define end-locations '( "Station 1" "Station 2" "Station 3" "Station 4"))

(define start-dropdown (new choice%
                        [label "Start Location:"]
                        [choices start-locations]
                        [parent panel]))

(define end-dropdown (new choice%
                      [label "End Location:"]
                      [choices end-locations]
                      [parent panel]))

(define find-route-button (new button%
                           [label "Find My Route"]
                           [parent panel]
                           [callback (lambda (button event)
                                       (routeplanner (send start-dropdown get-string-selection)
                                                     (send end-dropdown get-string-selection)))]))


(send frame show #t)
```

```
Welcome to DrRacket, version 8.10 [cs].
Language: racket/gui, with debugging; memory limit: 128 MB.
Your Route is (), then your final stop is Station 1. The total trip time is 0 minutes
Your Route is (Station 1), then your final stop is Station 2. The total trip time is 4 minutes
Your Route is (Station 1 Station 2), then your final stop is Station 3. The total trip time is 9 minutes
Your Route is (Station 1 Station 2 Station 3), then your final stop is Station 4. The total trip time is 15 minutes
Your Route is (Station 3), then your final stop is Station 4. The total trip time is 6 minutes
>
```

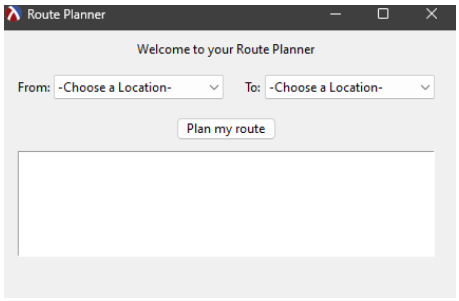


This is a basic implementation of a GUI to work with the routeplanner, it doesn't directly read from the graph, but instead, when find my route is pressed, the routeplanner function is called with the string of start location and end location used for the src and dest.

In Summary you have a working pathfinder function with a template for combining it with a GUI, In addition you could further this by adding a "map" and this will provide the underlying function for the route planning aspect of our routeplanner!

GUI

```
;;;;;;;;;;;;;;;;;;; GUI ;;;;;;;;;;;;;;;;;;
(define window
  (new frame%
       [label "Route Planner"]
       [width 400]
       [height 300]
       [style '(fullscreen-button)]
       [alignment '(center top)]
       ))


(define welcomemsg (new message%
     [parent window]
     [label "Welcome to your Route Planner"]
     [vert-margin 10]))

(define panel (new horizontal-pane%
     [parent window]
     [vert-margin 5]
     [horiz-margin 10]
     [alignment '(center top)]
     [stretchable-width #t]
     [stretchable-height #f]))

(define from (new choice%
    [parent panel]
    [label "From: "]
    [horiz-margin 10]
    [choices metropolitan-line]))

(define to (new choice%
    [parent panel]
    [label "To: "]
    [horiz-margin 10]
    [choices metropolitan-line]))
```



This is the GUI code created by edward, it's a straightforward implementation of a route planner that allows you to select your start and end location, it utilizes racket's ability to adjust parameters for frames and panels such as ordering elements horizontally,

```
                                ("Barbican" "Moorgate") ("Moorgate" "Barbican")
                                ("Moorgate" "Liverpool Street") ("Liverpool Street" "Moorgate")
                                ("Liverpool Street" "Aldgate") ("Aldgate" "Liverpool Street")))

(define edge (λ (x graph)
    (map (λ (y) (first (rest y))) (filter(λ (y)
              (equal? (first y) x)) graph))))

(define path (λ (x y a-graph vertex-set)
    (cond
      [(equal? x y) #t]
      [(not (set-member? vertex-set x)) #f]
      [(not (set-member? vertex-set y)) #f]
      [#t (ormap (λ (z)(path z y a-graph (set-remove vertex-set x )))
                                          (edge x a-graph ))]
      )))

(define reachable(λ (x y a-graph)
     (path x y a-graph (list->set (flatten a-graph)))))
```

Callbacks

Emmanuel worked on combining a template algorithm that finds out whether a route is “reachable” in a graph of many stations (this graph will be in the uploaded code), from a start and end point, to work with the GUI, using callbacks. This way once the pathfinding algorithm was created, it could be implemented to work with a GUI.

This is also the graph that is used for the app (within the parameters of the pathfinding algorithm) and weights will be added as such.

We all worked together to test the pathfinding algorithm with a GUI, with myself managing the algorithm itself, Ed managing the GUI and Emmanuel linking the GUI and algorithm.

```
(define planroutebutton (new button%
    [parent window]
    [vert-margin 10]
    [label "Plan my route"]
    [callback (λ (o e)
       (displayln (send from get-string-selection))
       (displayln (send to get-string-selection))
       (display(reachable (send from get-string-selection) (send to get-string-selection) metropolitan-graph)))


(define route-output (new editor-canvas%
    [parent window]
    [style '(no-vscroll no-hscroll)]
    [label "Your Route"]
    [min-width 300]
    [min-height 100]
    [horiz-margin 20]
    [stretchable-width #t]
    [stretchable-height #f]))

(send window show #t)
```

You can see this in the way the code works.

In Summary

As such this provides the main 3 elements required to build our route finder.

A GUI for selecting your start and end destination, this can be built upon to include features such as saving routes for quick use, and a map as shown in the examples prior.

A pathfinding algorithm that allows the selection of a start and end point and searches through nodes in a graph, finding the shortest path and the weight, and displays this

A way to combine these two in order to create seamless interaction between them and a user friendly UI that simplifies the use of the route finder.

We can then build upon this with the visual style as well as adding more to the graph, making sure our map representation is consistent and understandable and adding in the separate UI pages such as saving routes, previous routes, route costs, and so forth.

Just for some examples, saving routes could be done by providing an option to save the selected / previously used routes, which would store them under a unique variable and allow the user to select from them in a list

Previous routes could store all routes called through the algorithm in a list that the user could look through and select from.

Route costs could be calculated by the amount of stops or the amount of distance (weight) for a route, multiplied by an amount.

And so on.

For now, you have a working GUI that allows an algorithm to be implemented, a pathfinding algorithm that works based on a graph in both directions and displays the shortest route and time taken, and a way to implement this with the GUI.

—----------

(Samy Wanas **M00949455)**

(Emanuel Viorel Vochitoiu **M00942608)**

(Edwart Ianovici (**M00946886)**

Sob 10 (Contribute Code)
Sob 111 (Description to working prototype with limitations discussed and possible enhancements)
Sob 112 (Minor discussion of UI design)
Sob 206 (Created prototype)
Sob 207 (Demonstrated understanding of racket GUI)
Sob 229 (Written technical documentation)
Sob 101 (Uses lists and associated functions to solve problems)
Sob 145 (depends on what is considered extending a GUI)
Sob 208 (Uses graphs to solve a problem (pathfinding section))