C3GP Script

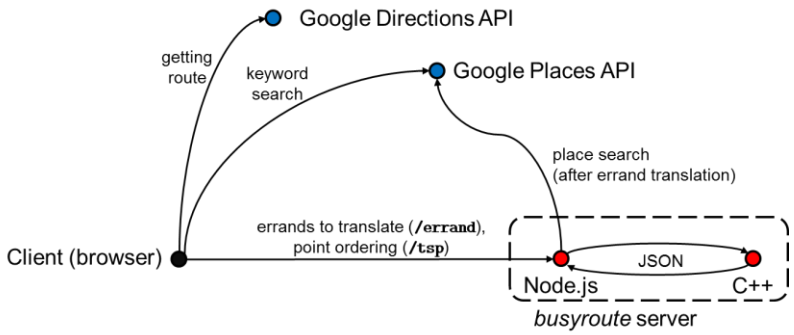
[Part 2: Busyroute Demo]

In sequence, the tasks Tom will talk about:

(1) *[I'm at Imperial…]*
    a. "Use current location"
(2) *[I'm tired, I need some caffeine…]*
    a. Type "caffeine",
    b. Select "grab a coffee",
    c. Click on the random hotel near High Street Kensington,
    d. Select Starbucks at Gloucester Road
(3) *[I need to withdraw some cash…]*
    a. Type "cash",
    b. Select ANY
(4) *[I need to collect my suit …]*
    a. Type "laundry",
    b. Select "do the laundry",
    c. Select Miracle Dry Cleaners
(5) *[I need to pick up some stationery…]*
    a. Type "stationery",
    b. Select ANY
(6) *[I need to get the route…]*
    a. Press "GET ROUTE"
(7) *[I want to edit the route…]*
    a. Press "EDIT ROUTE"
(8) *[I want to go to Portobello Market…]*
    a. Move the map to Portobello Road area (northwest, above Notting Hill Gate),
    b. Click a point along the road
(9) *[I want a sweet treat…]*
    a. Type "bakery",
    b. Click on "Teas Me" (the random bakery along Elgin Mews)
    c. Select the Hummingbird Bakery (just south of Lonsdale Road)
(10) *[I want to get the new route…]*
    a. Press "GET ROUTE"

[Part 3: Implementation, part 1]

System Architecture



How did we implement *BusyRoute*? Well, I will now walk you through, at a high level, what actually goes on when a user plans a route with *BusyRoute*.

When a user selects an errand from the web interface, such as “withdraw cash”, this request is routed to our server’s /errand endpoint, which returns a list of suitable places. Our server translates this into a suitable request to Google’s Places API, exploiting both its keyword and place type search functionality – for example, “withdraw cash” could perhaps be completed at places with type “ATM” and “Bank”.

While we can’t possibly support every possible errand, we have a means of coping with user requests in general. As demonstrated earlier with the stationery example, the user can always do a keyword search. You’ll notice we route this directly to the Google Places API from the client-side, since no preprocessing is required and thus routing through our server is unnecessary.

Once the user has decided on all the places he wants to visit, when he presses “GET ROUTE”, a call is made to our /tsp endpoint. The data on these places (and their group, for the “ANY” cases) is then sent to the C++ part of our back-end using JSON, which determines the best order in which we should visit the points. This is sent back to the client.

Now the client knows the precise locations and the order in which to visit them, we can call the Google Directions API from the client-side, to display the route to the user.

Computing Edge Weights

[Slide 1] I’ll now be focusing more on how we tried to calculate the best order to visit the points. This might look like a graph problem – and you’re right; it is one. However, before we approach the graph problem, we first need to find a way to estimate the edge weights – that is, the time taken between points of interest.

Since we know the endpoints of each leg, a possibility might be to call external services such as the same Google Directions API we use to draw the route. However, a problem we faced was usage limits; if we have *N* nodes (other than origin), we will need to make *N*^2 + *N* API calls to the Directions API, which has a free daily limit of 2500. With just 6 specific locations selected, that restricts our application to 59 requests per day, and a single “ANY” for a common errand could cause us to go over the limit. We thus needed an internal model that helps us estimate the time between points.

[Slide 2] Clearly, one way to travel between two points is walking. The straight-line distance between two points, given their latitude and longitude is given by the *haversine formula*. Of course, walking in a straight line is not always feasible or practical; we thus estimated the time required by multiplying the straight line distance by 1.25 and dividing by an individual's average walking speed of 5.0 km/h.

[Slide 3] However, walking is clearly not always the best mode of transport. We discussed with our PS and decided that for the scope of this project, considering the Tube when making routing decisions would be sufficient. We thus considered the possibility of incorporating the Tube into one's journeys – this would involve walking from the origin to a nearby station, taking the Tube to another station and then walking from said other station to the destination. The main challenge in estimating this is figuring out how long the Tube component of the journey would take.

[Slide 4] This data was not readily available from TfL; we did consider using TfL's official Journey Planner API but this required manual IP whitelisting on TfL's side and we only received approval for this very late in the project. We thus estimated the time taken between each pair of stations, based on TfL's working timetables (giving average train periods and time spent on each route segment) as well as data concerning turnstile-to-platform and platform-to-platform times (platform-to-platform used for considering changing trains). We constructed the underlying graph and used a variant of best-first search to determine the time required between all pairs of stations; an interesting challenge we faced here was that travel times could vary depending on whether one needs to change trains or not (for example, taking the tube from South Kensington to Knightsbridge takes about 2.5 minutes if entering via the eastbound Piccadilly from Gloucester Road, but 10.17 if entering via westbound District from Sloane Square due to the need to change trains). Tube stations thus did not map cleanly to individual nodes in the graph for a standard best-first search; we had to identify nodes by both their station and line.

[Slide 5] We verified the Tube estimation heuristics against TfL's official Journey Planner and found that they were largely within 5 minutes of the official estimated time. Due to differing availability of Tube lines at stations, we considered routes involving the three nearest Tube stations to the origin and destination, as well as walking, and selected the best (lowest) overall as the time taken to travel from origin to destination. I'll now hand over the presentation to my teammate Andrei, who will discuss how we use these edge weights to decide which order to visit points in.