ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises:
Modelling and Simulating Social Systems with MATLAB

Project Report

Traffic simulation in the city of Zurich

Jan Dörrie, Simone Forte, Charalampos Gkonos, Athina Korfiati

Zurich
May 2014

Agreement for free-download

We hereby agree to make our source code for this project freely available for download from the web pages of the SOMS chair. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

Jan Dörrie Simone Forte Charalampos Gkonos Athina Korfiati

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of Originality

This sheet must be signed and enclosed with every piece of written work submitted at ETH.

I hereby declare that the written work I have submitted entitled

Traffic simulation in the city of Zurich

is original work which I alone have authored and which is written in my own words.*

Author(s)

Last name	First name
Dörrie	Jan Wilken
Forte	Simone
Gkonos	Charalampos
Korfiati	Athina

Supervising lecturer

Last name	First name
Woolley, Kuhn, Biasini, Helbing	Olivia, Tobias, Dario, Dirk

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (http://www.ethz.ch/students/exams/plagiarism_s_en.pdf). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Zurich, May 13th 2014
Place and date

Jan Dörrie, Simone Forte
Signature

*Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Print form

Contents

1 Abstract

In this paper, a traffic simulation for the city of Zürich is developed. The representation of traffic flows is a very useful tool for urban planning and is essential in cases that large scale construction works are planned. In the case of the city of Zürich, we were inspired from a proposed project for the construction of an underground tunnel that would connect the East and West side of the lake. The goal of this paper is to examine and derive conclusions whether such a tunnel would affect (improve or even worsen) the traffic situation of the city centre. More specifically, the travelling times from certain starting points to ending ones will be examined before and after the construction of such a technical work. At the end of the simulation, the results will be analysed.

As an initial step, the route of every car is computed, using the shortest path algorithm (A* search algorithm), in the reduced road network that is selected for the project. Then, in order to model traffic propagation on given road segments, Cellular Automata model is used. Finally, the planned tunnel will be introduced not in one, but more possible solutions, and its impact on the traffic situation for each one of them will be examined.

2 Individual contributions

In our project, all four team members were active and contributed in the development of the simulation. More specifically, MATLAB coding and implementation was done by Jan Dörrie and Simone Forte. Charalampos Gkonos and Athina Korfiati were responsible for the data acquisition and data editing. The final analysis of the results, as well as the report preparation was performed by the whole team.

3 Introduction and Motivations

Traffic is a serious problem. Almost every big city in the world, no matter if it is in a developed or a developing country has to face this problem. Even in Zürich, the biggest city in Switzerland, where a very good system of public transportation exists and serves every corner of the city, the problem of traffic is not completely solved. Sometimes there is a need to look for innovative thoughts that may lead to perfect solutions. But first of all it is obvious that you have to evaluate all these alternatives in order to take the right decision. In this particular case study, a Highway Tunnel, proposed by the ARGE

Züriring consortium (Güller Güller architecture urbanism, Synergo, Heierli Engineering AG, Roland Müller, Metron AG, Ecoplan) to the Department of Civil Engineering of the Canton of Zürich in 2002 is examined. According to this Highway Tunnel Zürich project, a tunnel that connects the two sides of the Lake of Zürich is proposed among some other alternatives. (Güller Güller architecture urbanism, 2001-2002)

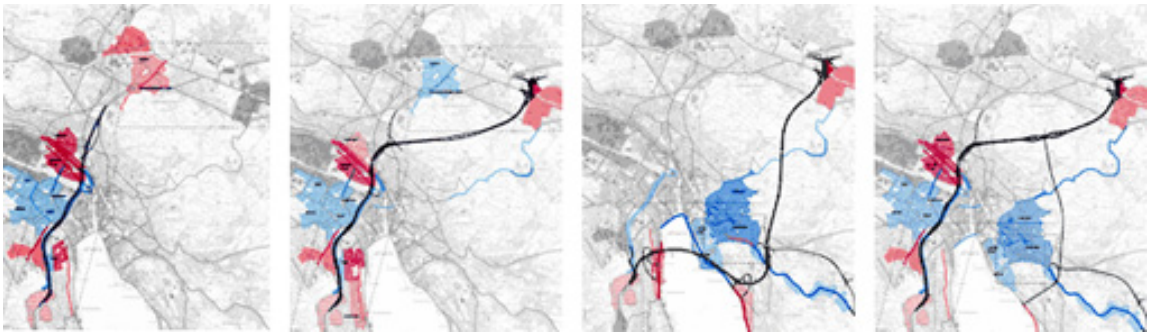

Figure 1: The four different solutions proposed in "Highway Tunnel Zürich" project [GGa14]

3.1 Fundamental questions

The basic question that this paper aims to answer is whether the construction of a tunnel will eventually improve or worsen the traffic situation in the centre of the city and in what way is this correlated with the number of the cars that use the network. The goal is to develop an appropriate and efficient traffic model based on the available public data. This model can be used to visualise the existing traffic situation in the city of Zürich, as well as to describe the expected results after the construction of the tunnel. Based on this knowledge, it is possible to draw useful conclusions, which can be analysed in order to evaluate the impact of such a tunnel.

3.2 Expected results

As far as the expected results are concerned, it is not easy to make a prediction apart from the most obvious one, that the existence of a tunnel would improve the travelling time and reduce the traffic in the centre. The problem is that the construction of huge technical works has always many more impacts in an area than the obvious (i.e. possible rise of the population living near the tunnel exits). An in-depth study is always needed which takes

into account both qualitative and quantitative factors like potential future development opportunities, construction cost, maintenance and safety and many more. Nevertheless, for the purpose of the present case study there was no need for a so detailed analysis and for that reason there is a higher probability of misjudgement.

4 Description of the Model

A classical urban transportation planning system model usually follows four steps[McN08]:

- **Trip generation** determines the frequency of origins or destinations of trips in each zone by trip purpose, as a function of land uses and household demographics, and other socio-economic factors.
- **Trip distribution** matches origins with destinations, often using a gravity model function, equivalent to an entropy maximizing model. Older models include the fratar model.
- **Mode choice** computes the proportion of trips between each origin and destination that use a particular transportation mode. (This modal model may be of the logit form, developed by Nobel Prize winner Daniel McFadden.)
- **Route assignment** allocates trips between an origin and destination by a particular mode to a route. Often (for highway route assignment) Wardrop's principle of user equilibrium is applied (equivalent to a Nash equilibrium), wherein each driver (or group) chooses the shortest (travel time) path, subject to every other driver doing the same. The difficulty is that travel times are a function of demand, while demand is a function of travel time, the so-called bi-level problem. Another approach is to use the Stackelberg competition model, where users ("followers") respond to the actions of a "leader", in this case for example a traffic manager. This leader anticipates on the response of the followers. (Wikipedia, n.d.)

The model that is developed for this case study is based on cellular automata. Every car is thus represented as an automaton which acts according to a very simple update rule; all the transitions will happen on a graph which represents the road network of a city, in this particular case this will be the city of Zürich. The model has three main parts:

- **Trip generation**: For every automaton we will generate a path that it will follow; this route will be decided at the beginning and it will stay unchanged for the rest

of the simulation. The way this routes will be generated will be based on a probability distribution defined by the use of statistical data[Wik14b] for the total of 34 neighbourhoods of the city of Zürich (file: population.csv) and the statistical data [Swi14] of the number of new companies that are founded in each area (file: companies.csv). What we aimed to model was the everyday morning journey of people from their houses to the workplace; we will thus have that the sources of the paths will be generated with a probability proportional to the number of people living in a certain area of the city and the destinations will be generated with a probability proportional to the number of companies situated in a certain area. Additionally to the city of Zürich we also took into account the population living in smaller towns southern of the river which we aggregated as incoming and 4 outgoing from the major highways located in the south of Zürich; this has been necessary in order to evaluate the impact of a tunnel in that area. A visualization of this can be seen in figure 2.

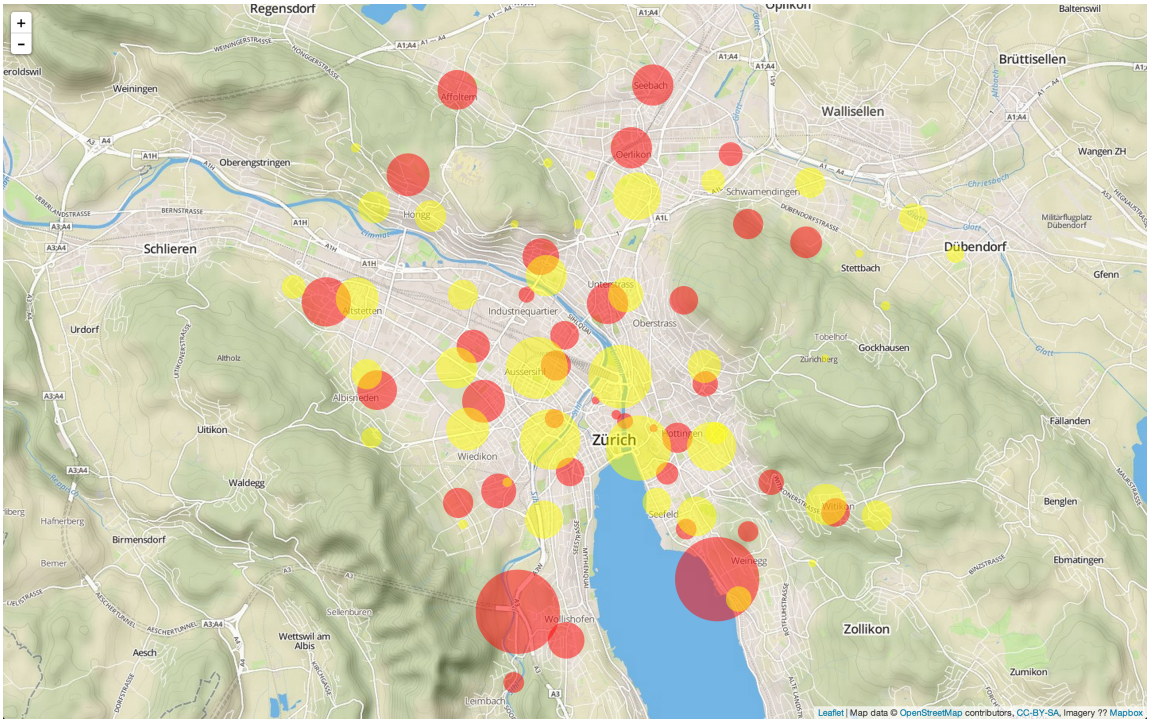

Figure 2: Visualization of the chosen population and company hubs. Red dots how population areas and yellow dots company ares.

- **Trip distribution**: After the intended journey for an automaton has been generated we will generate the actual path that the automaton intends to follow. We have in

particular considered a realistic assumption that the paths will be decided at the beginning of the journey to be the shortest path between two locations and will remain unchanged regardless of the traffic situation of a road. For the creation of these paths the A* search algorithm has been used.

- **Journey execution**: At this point the automaton knows the actual path it wants to follow. It will thus be initialized in its starting location and at every time step, having length in milliseconds equal to **m**, it will simply advance of a quantity equal to the minimum between the distance it can travel given that it travel for **m** milliseconds at the maximum speed allowed by the road it is following and the distance of the next car on the path minus a safety distance. To decide in which order update the cars in a timestep we will simply iterate on the cars in a random order and all the decisions for a car will be taken considering the position of the other cars at the time in which the car is updated.

- **Route assignment** At this point the automaton knows the actual path it wants to follow. It will thus be initialized in its starting location and at every time step, having length in milliseconds equal to m, it will simply advance of a quantity equal to the minimum between the distance it can travel given that it travel for m milliseconds at the maximum speed allowed by the road it is following and the distance of the next car on the path minus a safety distance. To decide in which order to update the cars in a timestep, we will simply iterate on the cars in a random order and all the decisions for a car will be taken considering the position of the other cars at the time in which the car is updated.

5 Implementation

We started out by obtaining the OpenStreetMap[Ope14] file that contains Zürich. In order to do so we downloaded a recent Switzerland extract from *Geofabrik* [Geo14] and then extracted the boundaries of Zürich with the tool *Osmconvert* [Wik14a]. Afterwards we used another tool called *Osmosis*[Wik12] to limit our data to the roads in Zürich. Furthermore we modified the data using *JOSM* which enabled us to simplify the road network, making our algorithms more efficient. Once we removed small isolated connected components resulting from clipping the data to Zürich with self-written Python scripts we extracted a simple graph structure, just containing a list of nodes and edges. Each node was annotated with its id, as well as latitude and longitude. For each edge we referenced existing nodes by their id and added the allowed maximum speed on this edge. This data we also extracted with the help of Python scripts and interpolated from similar results in case this data was missing. Afterwards we wrote an A-Star algorithm implementation in

C++ in order to find the shortest path between two given nodes in the graph. Initially we had used Dijkstra's algorithm for this, however once we realized that this is a prime example for A Star using the Euclidean distance we switched and obtained a much faster algorithm. We considered implementing this algorithm directly in MATLAB, however the inconvenience of working with adjacency lists and the overall much slower performance forced us to write a C++ implementation. However, by turning it into a mex file, we were still able to call this function directly from our MATLAB code, hence the integration was very smooth. We further wrote a wrapper around this function in MATLAB, that would perform sanity checks on the input and exit gracefully in case there is an error.

The MATLAB implementation mostly consists of two parts, an initialization part and an update part. In the initialization the main data structures and parameters of the model are set up; we thus generate the routes for every car, picking starting and ending location according to a probability distribution specified by the CSVs files (reference...) and the actual path will be generated by computing the shortest path between the two locations. In the update part, for every timestep, we compute the new locations of every car according to the way specified earlier in the Description of the Model section and we compute the new GEO coordinates of the cars which are then given back to the Javascript client.

Finally we made use of the Javascript Library *Leaflet*[Lea12] to visualize our results on the actual map of Zürich. We enabled communication between MATLAB and Javascript using the included Java classes for socket communication. The Javascript code issues HTTP GET requests telling the MATLAB implementation which methods to perform. The actions include initializing and advancing the simulation. After each simulation step we passed the current car locations as a JSON object through the socket. For encoding the object as a JSON string we made use of Google's Gson library[Goo14]. The Javascript implementation then made sure to display the cars at the right locations on the map.

6 Simulation Results and Discussion

To be able to interpret the results of the modelled simulation in a sensible way, the results are divided in three individual cases. The connecting point of the three cases is the average travel time, that in the end is the parameter that defines the level of success or failure of the individual simulation tests. In the first case, the correlation between the actual number of the cars that use the road network and the travel time they need in order to reach their destination is examined. In the second case, the tunnel is introduced in the existing road network and this time the simulation model calculates the travel time in correlation with some different possible positions for its construction. In the third case, the comparing

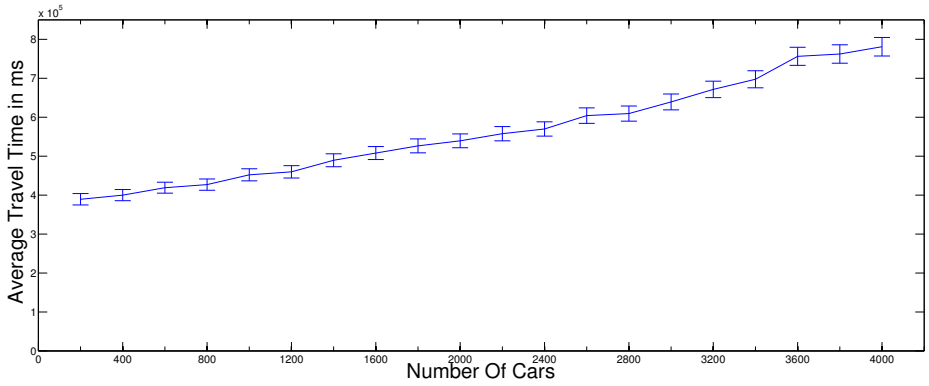

Figure 3: Plot of travel times with regard to increasing number of cars on the road

model parameter is once again the average time needed for cars to move from the traffic generation points to their destination points at first in the current situation where the tunnel does not exist and then in the case the tunnel is constructed.

6.1 In correlation with the number of cars

In this first simulation evaluation the travel time is examined using different numbers of cars that actually use the existing road network of the city of Zürich. As it was initially expected and also in the end is confirmed by the results of the simulation (as shown in Figure 3) there is a strong correlation between the travelling time and the number of cars that use the road network. Of course this is something that is expected mainly because of the capacity limitations of the existing road network (there is no prediction for a large number of cars in the city of Zürich). More specifically, there is an average increase of around 25% of the travel time when the number of cars is raising from 400 to 1600, more than 50% when the number of cars changes from 2000 to 4000, and finally a massive one of 100% when the number of cars increases from 400 to 4000.

6.2 In correlation with the chosen position for the tunnel construction

In this simulation test the travel time is evaluated with respect to the position of the tunnel. Here, sixteen different scenarios are examined, which result from all the possible combinations of eight starting/ending points on the east (four points) and west side (four

points) of the lake of Zürich.

From the sixteen different scenarios that are evaluated and are presented in Table 1 it comes out as a conclusion that with the available data, the alternative positions for the tunnel construction do not influence significantly the average time that a driver needs to move from his starting point to his destination. This is probably due to the fact that all sixteen tunnel position alternatives are close enough to each other as it can be seen in Figure 4.

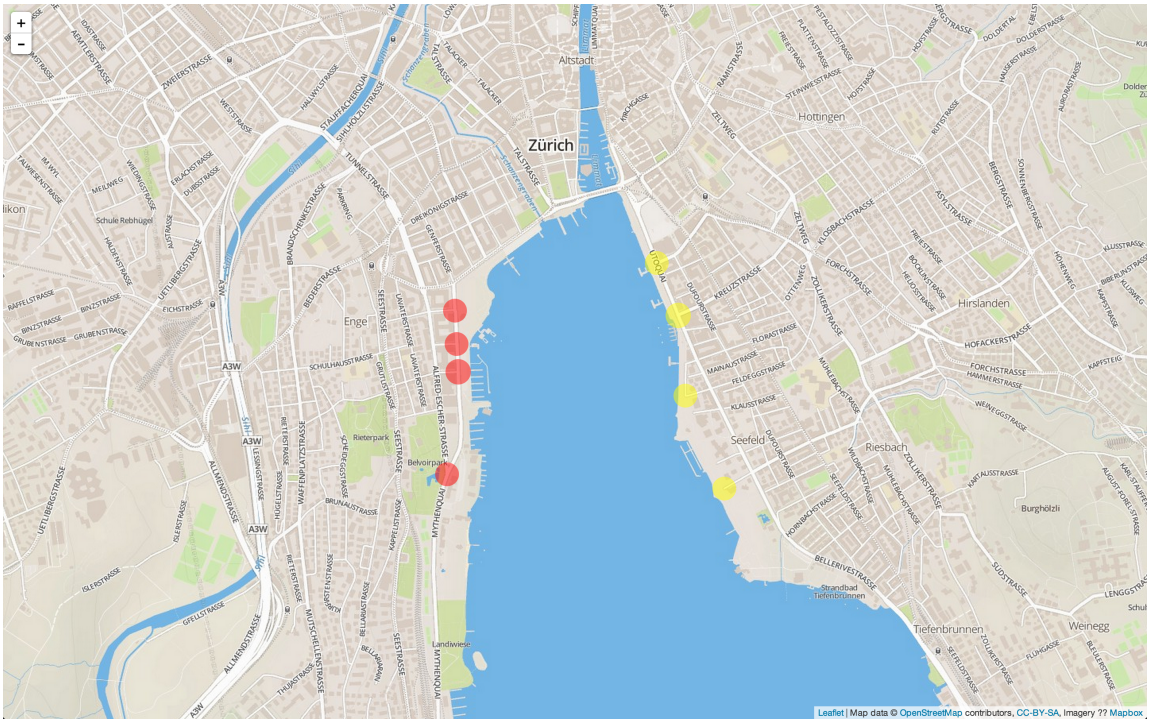

Figure 4: Visualization of the various start and end points of the tunnel. Red dots show possible starting positions and yellow dots possible end positions.

The general outcome of this evaluation is that the tunnel is expected to be used in each case by the same people, no matter its exact position. Its impact on the travel time will be almost the same in all sixteen cases, no matter what the number of cars using the road network.

Table 1: Average travel time in milliseconds with regard to tunnel location and number of cars

start node id	end node id	500 cars	1000 cars	2000 cars
170424926	1199905289	398570	447040	576070
170424926	2114250	419070	444740	542180
170424926	2114253	410970	453580	564700
170424926	80041651	404440	441430	541860
170425961	1199905289	420470	453320	544150
170425961	2114250	423860	443500	533280
170425961	2114253	421850	435940	539880
170425961	80041651	399220	450070	555110
28961405	1199905289	409720	441690	527970
28961405	2114250	414000	447560	520660
28961405	2114253	393170	447390	536280
28961405	80041651	401790	443810	553050
329026137	1199905289	417620	440940	553870
329026137	2114250	424000	440600	533280
329026137	2114253	407270	439130	531330
329026137	80041651	412490	441830	536610
—	—	400872	471644	545545

6.3 In correlation with the construction of the tunnel in general

In the last simulation the average travelling time for the drivers before and after the construction of the tunnel is evaluated. The simulation results are summarized in table 1.

From the examination and interpretation of this simulation results it is obvious that travel time is not always reduced when the tunnel is built. The whole traffic situation in the centre of the city remains almost the same as it can be seen in Table 1. The comparison of the average time travel before the construction and after reveals that travel time depends on the number of cars that use the road network. More specifically, when the number of cars is limited (500 cars) the construction of the tunnel does not improve the travel time, but in thirteen out of sixteen cases the situation is even worse. In the intermediate case where the number of cars is higher (1000 cars) an amelioration of the situation is observed for every possible node combination. In the case where the number of cars is even higher (2000 cars) the travel time is reduced in eleven out of sixteen cases but not significantly.

7 Summary and Outlook

The present case study tries to visualize the traffic situation in the city of Zürich and evaluate the impact that a tunnel construction could have on the travel time, especially between the two sides of the lake. The research questions defined in the beginning of the research were answered and the conclusion is that the tunnel could improve the travel time for the benefit of the drivers using the road network of Zürich only in specific cases, for instance when the load of the network is relatively low (approximately 1000 cars).

For the implementation of the simulation, many consumptions were made (e.g. for the population and the travelling purpose) and the data used were limited. For this reason, there are still enough points for improvement. However, the simulation model is a good base for testing real and official data for the city of Zürich and this is yet a point for future exploration. Future work on this topic should include better data acquisition. The data should refer to a wider region and not focus specifically in the city of Zürich. The main reason for that is that such a technical work is usually expected to influence a wider area and a significantly larger amount of people.

References

[Geo14] Geofabrik. *Switzerland*. 2014. URL: http://download.geofabrik.de/europe/switzerland.html/ (visited on 02/03/2014).

[GGa14] GGau. *Lake-tunnel: new highway-tunnel for Zürich*. 2014. URL: http://www.ggau.net/html/GG03.html (visited on 02/03/2014).

[Goo14] Google. *google-gson*. 2014. URL: https://code.google.com/p/google-gson/ (visited on 02/03/2014).

[Lea12] Leaflet. *An Open-Source JavaScript Library for Mobile-Friendly Interactive Maps by CloudMade*. 2012. URL: http://leaflet.cloudmade.com/ (visited on 12/03/2012).

[McN08] Michael G McNally. "The four step model". In: (2008).

[Ope14] OpenStreetMap. *OpenStreetMap*. 2014. URL: http://www.openstreetmap.org/ (visited on 02/03/2014).

[Swi14] Moneyhouse Switzerland. *Trade register data and business information*. 2014. URL: http://www.moneyhouse.ch/en (visited on 02/03/2014).

[Wik12] OpenStreetMap Wiki. *Osmosis*. 2012. URL: http://wiki.openstreetmap.org/w/index.php?title=Osmosis&oldid=834077 (visited on 12/03/2012).

[Wik14a] OpenStreetMap Wiki. *Osmconvert*. 2014. URL: http://wiki.openstreetmap.org/w/index.php?title=Osmconvert&oldid=1010333 (visited on 02/03/2014).

[Wik14b] Wikipedia. *Subdivisions of Zürich*. 2014. URL: http://en.wikipedia.org/w/index.php?title=Subdivisions_of_Z%C3%BCrich&oldid=603437314 (visited on 02/03/2014).

Appendices

A C++ Code

A Star C++ Implementation

```
/*****************************************************************************
* Includes
*
*****************************************************************************/
#include <vector>
#include <queue>
#include <deque>
#include <algorithm>
#include <cmath>
#include <map>
#include <functional>
#include "mex.h"

/*****************************************************************************
* Using directives
*
*****************************************************************************/
using std::map;   using std::vector;  using std::pair; using std::priority_queue;
using std::deque; using std::greater; using std::max;  using std::make_pair;

/*****************************************************************************
* Typedefs
*
*****************************************************************************/
typedef vector< vector<int> > Graph;
typedef pair<double, int> State;
typedef priority_queue< State, vector<State>, greater<State> > MinHeap;


/**
 * Struct representing an OSM node. It consists of its id together with its
 * position (latitude and longitude).
 */
struct Node
{
    long long id;
    double lat, lon;

    Node(long long _id = 0, double _lat = 0.0, double _lon = 0.0) :
        id(_id), lat(_lat), lon(_lon) { }
```

```
};


/*****************************************************************************
* Global variables
*
*****************************************************************************/
const double radius = 6371.0;          // Radius of the earth in km
vector<Node> nodes;                    // Global vector keeping track of all nodes
vector<map<int, double> > dist_map;    // Global look up table for computed distances


/**
 * Method to convert degrees to radians
 * @param  deg degrees
 * @return radians
 */
inline double deg2rad(double deg) {
    return deg * (M_PI / 180.0);
}


/**
 * Computes the haversine distance between two points with given lat and lon.
 * @param  u First Node
 * @param  v Second Node
 * @return Distance
 */
double hav_dist(const Node& u, const Node& v)
{
    double dLat = deg2rad(v.lat - u.lat);
    double dLon = deg2rad(v.lon - u.lon);

    double a = sin(dLat / 2.0) * sin(dLat / 2.0) +
        cos(deg2rad(u.lat)) * cos(deg2rad(v.lat)) *
        sin(dLon / 2.0) * sin(dLon / 2.0);

    double angle = 2 * atan2(sqrt(a), sqrt(1.0 - a));
    return angle * radius;
}


/**
 * Computes the distance between two points with given lat and lon using the law
 * of cosines.
 * Makes use of dist_map to speed up future computations.
 * @param  u_idx Index of the first Node
 * @param  v_idx Index of the second Node
 * @return       Distance
 */
```

```
double cos_dist(int u_idx, int v_idx)
{
    if (dist_map[u_idx].find(v_idx) != dist_map[u_idx].end())
    {
        return dist_map[u_idx][v_idx];
    }

    const Node& u = nodes[u_idx];
    const Node& v = nodes[v_idx];

    double lat1 = deg2rad(u.lat);
    double lat2 = deg2rad(v.lat);

    double dLon = deg2rad(v.lon - u.lon);

    double dist = acos(sin(lat1) * sin(lat2) +
        cos(lat1) * cos(lat2) * cos(dLon)) * radius;
    dist_map[u_idx][v_idx] = dist_map[v_idx][u_idx] = dist;

    return dist;
}


/**
 * Approximates the distance between two points with given lat and lon using
 * their euclidean distance.
 * Makes use of dist_map to speed up future computations.
 * @param  u_idx Index of the first Node
 * @param  v_idx Index of the second Node
 * @return       Distance
 */
double eucl_dist(int u_idx, int v_idx)
{
    if (dist_map[u_idx].find(v_idx) != dist_map[u_idx].end())
    {
        return dist_map[u_idx][v_idx];
    }

    const Node& u = nodes[u_idx];
    const Node& v = nodes[v_idx];

    double lat1 = deg2rad(u.lat);
    double lat2 = deg2rad(v.lat);

    double lon1 = deg2rad(u.lon);
    double lon2 = deg2rad(v.lon);

    double x = (lon2 - lon1) * cos((lat1 + lat2) / 2);
    double y = (lat2 - lat1);
    double dist = sqrt(x * x + y * y) * radius;
```

```
    dist_map[u_idx][v_idx] = dist_map[v_idx][u_idx] = dist;

    return dist;
}


// The following functions were used for debugging, uncomment if needed.

// void print_path(const deque<int>& path)
// {
//     for (int i = 0; i < path.size(); ++i)
//     {
//         mexPrintf("%d%c", path[i], i + 1 < path.size() ? ' ' : '\n');
//     }
// }

// void print_path(const vector<long long>& path)
// {
//     for (int i = 0; i < path.size(); ++i)
//     {
//         mexPrintf("%lld%c", path[i], i + 1 < path.size() ? ' ' : '\n');
//     }
// }

// void print_array(int num_snk, double *snk_ptr)
// {
//     for (int i = 0; i < num_snk; ++i)
//     {
//         mexPrintf("%lld%c", (long long) snk_ptr[i], i + 1 < num_snk ? ' ' : '\n');
//     }
// }


/**
 * A Star implementation.
 * It constructs a graph from arrays of nodes and edges.
 * Then it runs one iteration for each src/snk pair.
 * It returns the resulting shortest paths.
 * @param  num_nodes number of nodes
 * @param  nodes_ptr pointer to node array
 * @param  num_edges number of edges
 * @param  edges_ptr pointer to edge array
 * @param  num_src   number of sources
 * @param  src_ptr   pointer to source array
 * @param  num_snk   number of sinks
 * @param  snk_ptr   pointer to sink array
 * @return           Matrix M containing ids of the nodes
 *                   on the shortest paths. M[i][j] is the
 *                   jth node on the path from src[i] to snk[i].
```

```
 */
vector<vector<long long> > run_a_star(int num_nodes, double *nodes_ptr,
    int num_edges, double *edges_ptr, int num_src, double *src_ptr,
    int num_snk, double *snk_ptr)
{
    // maps id numbers to indices
    map<long long, int> id_to_idx;
    map<int, long long> idx_to_id;

    // allocate space for the nodes
    nodes = vector<Node>(num_nodes);
    dist_map = vector<map<int, double> > (num_nodes);


    // read in all node information, store current index with id
    for (int i = 0; i < num_nodes; ++i)
    {
        Node node;
        node.id = nodes_ptr[3 * i];
        node.lat = nodes_ptr[3*i + 1];
        node.lon = nodes_ptr[3*i + 2];

        nodes[i] = node;
        id_to_idx[node.id] = i;
        idx_to_id[i] = node.id;
    }

    // allocate space for the adjacency map
    vector<vector<pair<int, double> > > G(num_nodes);

    // Keep track of the maximum speed encountered.
    // Important for the heuristic to stay admissible.
    double max_speed = 0.0;

    // read in the edges, map ids to indices and store them in the adjacency list
    for (int i = 0; i < num_edges; ++i)
    {
        long long u_id = edges_ptr[3 * i];
        long long v_id = edges_ptr[3*i+1];
        double speed   = edges_ptr[3*i+2];

        int u_idx = id_to_idx[u_id];
        int v_idx = id_to_idx[v_id];

        G[u_idx].push_back(make_pair(v_idx, speed));
        G[v_idx].push_back(make_pair(u_idx, speed));

        max_speed = max(max_speed, speed);
    }
```

```
vector<vector<long long> > shortest_paths(num_src);

// vector storing the parent in the induces spanning tree of every node
vector<int> parent(num_nodes, -1);

// run a_star for every src node
for (int id_src = 0; id_src < num_src; ++id_src)
{
    // retrieve indices of src and snk
    int src_id = src_ptr[id_src];
    int src_idx = id_to_idx[src_id];

    int snk_id = snk_ptr[id_src];
    int snk_idx = id_to_idx[snk_id];

    // allocate the heap, together with vectors keeping track of the used time
    // and if we visited the current node already
    MinHeap heap;
    vector<double> used_time(num_nodes, INFINITY);
    vector<int> visited(num_nodes);

    // initialize with the current src node
    parent[src_idx] = src_idx;
    used_time[src_idx] = 0.0;
    State start = make_pair(0.0, src_idx);

    heap.push(start);

    while (!heap.empty())
    {
        // pop the first element and retrieves the index
        State u = heap.top(); heap.pop();
        int u_idx = u.second;

        // break if we reached the sink
        if (u_idx == snk_idx)
            break;

        // otherwise continue popping the next element if we
        // visited this node already
        if (visited[u_idx])
            continue;

        // else mark it as visited
        visited[u_idx] = 1;

        // loop through all adjacent nodes
        for (int i = 0; i < G[u_idx].size(); ++i)
        {
            int adj_idx = G[u_idx][i].first;
```

```
            double speed = G[u_idx][i].second;

            // continue if this node was visited already
            if (visited[adj_idx])
                continue;

            // compute the time it takes to get from node u to the current node
            double curr_time = eucl_dist(u_idx, adj_idx) / speed;

            // compute the total time to get to this node from the source
            double new_time = curr_time + used_time[u_idx];

            // if the new time is smaller than the one stored update it
            // and push the node on the heap together with the estimated
            // time to the sink
            if (new_time < used_time[adj_idx])
            {
                parent[adj_idx] = u_idx;
                used_time[adj_idx] = new_time;
                heap.push(make_pair(new_time + eucl_dist(adj_idx, snk_idx)
                    / max_speed, adj_idx));
            }
        }
    }

    // check if we were not able to reach the sink
    int curr_idx = id_to_idx[snk_ptr[id_src]];
    if (used_time[curr_idx] == INFINITY)
    {
        mexPrintf("No Path between %lld and %lld \n", idx_to_id[src_idx],
            idx_to_id[curr_idx]);
        continue;
    }

    // reconstruct path by following the route induced by the parent vector
    // we use a deque here so that we can efficiently do push_front operations
    deque<int> path;
    while (true)
    {
        path.push_front(curr_idx);

        if (curr_idx == parent[curr_idx])
            break;

        curr_idx = parent[curr_idx];
    }

    // finally retrieve the actual id of each node on the path
    // and store it in the shortest paths matrix
    for (int i = 0; i < path.size(); ++i)
```

```
        {
            shortest_paths[id_src].push_back(idx_to_id[path[i]]);
        }
    }

    return shortest_paths;
}


/**
 * Required mexFunction, so that Matlab can communicate with this program.
 * @param nlhs number of output arguments
 * @param plhs pointer to output arguments
 * @param nrhs number of input arguments
 * @param prhs pointer to input arguments
 */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    // Do very limited sanity checks.
    // Note that in general you should never call this function directly,
    // always use the a_star.m wrapper providing much more exhaustive input checks.
    if (nrhs != 4)
    {
        mexErrMsgIdAndTxt("MATLAB:a_star:nargin",
            "A_STAR requires four input arguments.");
    }

    if (nlhs < 1 || nlhs > 2)
    {
        mexErrMsgIdAndTxt("MATLAB:a_star:nargout",
            "A_STAR requires one or two output arguments.");
    }


    for (int i = 0; i < nrhs; ++i)
    {
        if (!mxIsDouble(prhs[i]))
        {
            mexErrMsgIdAndTxt("MATLAB:a_star:inputNotDouble",
                "A_STAR requires the input to be doubles.");
        }
    }

    // get double pointers to the input arguments
    double *nodes_ptr = mxGetPr(prhs[0]);
    double *edges_ptr = mxGetPr(prhs[1]);
    double *src_ptr = mxGetPr(prhs[2]);
    double *snk_ptr = mxGetPr(prhs[3]);

    // get dimensions of the input
```

```
    int num_nodes = mxGetN(prhs[0]);
    int num_edges = mxGetN(prhs[1]);
    int num_src = mxGetN(prhs[2]);
    int num_snk = mxGetN(prhs[3]);

    // run a_star on the input
    vector<vector<long long> > shortest_paths = run_a_star(num_nodes, nodes_ptr,
        num_edges, edges_ptr, num_src, src_ptr, num_snk, snk_ptr);

    // extract max_path_length, necessary for dynamically allocating memory afterwards
    int max_path_length = 0;
    for (int i = 0; i < num_src; ++i)
    {
        max_path_length = max(max_path_length, (int) shortest_paths[i].size());
    }

    // create the output matrix
    plhs[0] = mxCreateDoubleMatrix(max_path_length, num_src, mxREAL);

    // print size of the output dimension
    // also useful to see that this routine is about to finish
    mexPrintf("num_src: %d\n", num_src);
    mexPrintf("max_path_length: %d\n", max_path_length);

    // retrieve a double pointer to the output matrix and write the computed paths to it
    double *path_ptr = mxGetPr(plhs[0]);

    for (int i = 0; i < num_src; ++i)
        for (int j = 0; j < shortest_paths[i].size(); ++j)
            path_ptr[i * max_path_length + j] = shortest_paths[i][j];
}
```

B MATLAB Code

MATLAB Wrapper around A Star Implementation

```
function paths = a_star(nodes, edges, sources, sinks)
%A_STAR Shortest paths from nodes 'sources' to nodes 'sinks' using A-Star
    algorithm.
% paths = a_star(nodes, edges, sources, sinks)
%     nodes   = 3 x n node list where the first column specifies node ids
%               and the second and third latitude and longitude.
%     edges   = (2 or 3) x m edge list referencing node ids in the first two
    columns
%               and giving an optional maximum speed in the third column.
%     sources = 1 x s vector with FROM node indices.
%     sinks   = 1 x s vector with TO node indices.
%     paths   = p x s matrix specifying paths from sources to sinks where p
    is the
%               longest path length.


    % Input Error Checking
        ************************************************
    narginchk(4, 4);
    nargoutchk(0, 1);

    [dim_nodes, ~] = size(nodes);

    if dim_nodes ~= 3
        error('nodes must contain 3 columns.');
    end

    [dim_edges, num_edges] = size(edges);

    if dim_edges ~= 2 && dim_edges ~= 3
        error('edges must contain 2 or 3 columns.');
    end

    [dim_sources, num_sources] = size(sources);

    if dim_sources ~= 1
        error('sources must be a row vector.');
    end

    [dim_sinks, num_sinks] = size(sinks);

    if dim_sinks ~= 1
        error('sinks must be a row vector.');
    end
```

```
    if num_sources ~= num_sinks
        error('sources and sinks must be of the same size.');
    end

    node_ids = nodes(1, :);

    if length(unique(node_ids)) ~= length(node_ids)
        error('The node ids must be unique.');
    end

    if ~all(all(ismember(edges(1:2, :), node_ids)))
        error('edges must reference existing nodes.');
    end

    if ~all(ismember(sources, node_ids))
        error('sources must reference existing nodes.');
    end

    if ~all(ismember(sinks, node_ids))
        error('sinks must reference existing nodes.');
    end

    if dim_edges == 3 && min(edges(3, :)) < 0.0
        error('Travel times must be non-negative.');
    end
    %% End (Input Error Checking)
        ********************************************

    if dim_edges == 2
        edges = [edges; ones(1, num_edges)];
    end

    paths = a_star_mx(nodes, edges, sources, sinks);
end
```

Advances the simulation by one step

```
function advance_simulation()
    % the meaning of this variables is explained in initialize simulation
    global graph cars nodes paths millis travel_times safetydist over
        max_speeds;

    num_cars = size(paths, 2);

    % we will use perm to iterate on the cars in a random order
    perm = randperm(num_cars);

    % this will stay set to 1 if every car has reached its final
    % destination
    over = 1;

    % for every car we update its position
    for h = 1:num_cars
        i = perm(h);

        % the position of the car on its path
        path_idx = cars(i,3);
        % the current coordinates of the car
        curr_coords = cars(i,1:2);

        % if path_idx has been set to 0 then the car has reached its final
        % destination and we don't consider it anymore
        if path_idx == 0
            cars(i,:) = [0 0 0];
            continue;
        end
        over = 0;

        % convert millis to hours
        time = millis / 3600000;

        dist = 0;
        flag = 1;

        % we find the maximum distance a car can travel in the allotted
        % time, depending on the speeds of the roads and on the next
        % closest car on the same edges
        while flag && path_idx + 1 <= size(paths, 1) && paths(path_idx + 1, i
            ) ~= 0

            % car i is on edge (u,v)
            u = paths(path_idx, i);
            v = paths(path_idx + 1, i);

            % max speed on edge (u,v)
```

```
    u_speeds = max_speeds{u};
    max_speed = u_speeds(v);

    % length of edge (u,v)
    edge_len = distance(curr_coords, nodes(:, v));

    % this is the max distance car i can travel on edge (u,v)
    % in <time> milliseconds
    dist = dist + min(time * max_speed, edge_len);
    % we update the remaining time for this car
    time = time - edge_len / max_speed;

    % we find the distance to the next closest car on the edge
    closestcar = closest_car(i,curr_coords,u,v);

    % if time elapsed or there is another car on the edge we stop
    if time <= 0 || closestcar < inf
        flag = 0;
    end

    % we go to the next edge on the path
    curr_coords = nodes(:,v);
    path_idx = path_idx + 1;
end

% we compute the actual maximum distance the car can travel, taking
% into account the next closest car and the safety distance
dist = max(0, min(dist, closestcar-safetydist));

% We now actually move the car along the path at most by the
% distance dist we just computed
path_idx = cars(i,3);
while path_idx+1 <= size(paths,1) && paths(path_idx+1,i) ~= 0 && ...
        distance(cars(i,1:2),nodes(:,paths(path_idx+1,i))) <= dist
    dist = dist - distance(cars(i,1:2),nodes(:,paths(path_idx+1,i)));
    % we move the car edge by edge
    move_car(i);
    path_idx = cars(i,3);
end

% if the path has ended we go to the next car
if path_idx >= size(paths,1) || paths(path_idx+1,i) == 0
    cars(i,:) = [0 0 0];
    continue;
end

% we update the travel times for every car
travel_times(h) = travel_times(h) + millis;

% we compute the new coordinates of the car on the edge the car has
```

```
        % finaly landed on
        curr_coords = cars(i,1:2);
        next = nodes(:,paths(path_idx+1,i));

        off = dist/distance(curr_coords,next);

        cars(i,1) = cars(i,1) + off * (next(1)-curr_coords(1));
        cars(i,2) = cars(i,2) + off * (next(2)-curr_coords(2));
    end
end
```

Computes the distance to the closest car

```
% For car i having current position curr_car_pos and located at edge u,v
% we find the distance to the closest next car on edge (u,v)
function [min_dist] = closest_car(i, curr_car_pos, u, v)

    global graph cars nodes;

    edge = graph{u}(v);

    min_dist = inf;

    % if the car is the only one on the edge then return
    if size(edge) <= 1
        return
    end

    dista = distance(nodes(:,u), curr_car_pos);
    % find the closest next car on edge (u,v)
    for car_id = edge.keys()
        if car_id{1} == i
            continue
        end

        distb = distance(nodes(:,u), cars(car_id{1}, 1:2));

        if distb > dista
            min_dist = min(min_dist, distb - dista);
        end
    end
end
```

Computes the distance between two given points

```
function [dist] = distance(u, v)
%DISTANCE Computes the distance between two points given as latitude and
    longitude.
% It approximates the law of cosines formula by a modified euclidean distance
     for performance
% reasons.
% [dist] = distance(u, v)
%     u    = first node
%     v    = second node
%     dist = computed distance

    radius = 6371.0;

    lat1 = (pi/180) * (u(1));
    lat2 = (pi/180) * (v(1));

    lon1 = (pi/180) * (u(2));
    lon2 = (pi/180) * (v(2));

        x = (lon2 - lon1) * cos((lat1 + lat2) / 2);
        y = (lat2 - lat1);
        dist = sqrt(x * x + y * y) * radius;
end
```

Randomly generates start and end positions from the provided population data

```
% We generates the ending and starting point for the journey of every car
% according to a probability distribution depending on population data and
% on companies distribution
function [sources,sinks] = generate_routes(numcars)
    global nodes;

    % we read the csv files with the data in percentages
    pop = csvread('../data/Population/population.csv');
    com = csvread('../data/Companies/companies.csv');

    % we generate a number between in [0,100] for the sources; this will be
    % needed to generate randomly according to the percentages in pop and com
    sources_i = rand(numcars,1)*100;
    sinks_i = rand(numcars,1)*100;

    sources_c = zeros(numcars,2);
    sinks_c = zeros(numcars,2);

    sources = zeros(numcars,1);
    sinks = zeros(numcars,1);

    % for every car we generate the ending points of the journey
    for i = 1:numcars
        % we generate a random coordinate for the source; we first pick
        % the area center and then we add a random shift
        for j = 1:size(pop,1)
            if sources_i(i) <= pop(j,3)
                sources_c(i,:) = pop(j,1:2) + (rand(1,2)*0.01);
                break;
            end
            sources_i(i) = sources_i(i) - pop(j,3);
        end
        % we generate a random coordinate for the sink
        for j = 1:size(com,1)
            if sinks_i(i) <= com(j,3)
                sinks_c(i,:) = com(j,1:2) + (rand(1,2)*0.01);
                break;
            end
            sinks_i(i) = sinks_i(i) - com(j,3);
        end

        % for source and sink we find the nearest node to the coordinates
        % we generated and we set them as the start for the journey.
        sources_mind = inf;
        sinks_mind = inf;
        for j = 1:size(nodes,2)
            a = nodes(2:3,j); b = sources_c(i,:);
```

```
            dist = ((a(1)-b(1))^2)+((a(2)-b(2))^2);
            if dist < sources_mind
                sources_mind = dist;
                sources(i) = j;
            end
            a = nodes(2:3,j); b = sinks_c(i,:);
            dist = ((a(1)-b(1))^2)+((a(2)-b(2))^2);
            if dist < sinks_mind
                sinks_mind = dist;
                sinks(i) = j;
            end
        end
    end
end
```

Returns the current time as a string

```
function date_str = get_time_str()
%GET_TIME_STR Returns the current date and time as a string in the format '
    YYYY-MM-DD_hh:mm:ss'
%       date_str = date string in the format 'YYYY-MM-DD_hh:mm:ss'
        date_str = sprintf('%04d-%02d-%02d_%02d:%02d:%02d', fix(clock));
end
```

Initializes the simulation

```
% we initialize all the data structures needed for the simulation and we
% generate the routes for every car
function initialize_simulation(numcars)
    % graph: this variable is a cell array containing a map for every
    % vertex u in the graph; every map contains an entry for every adjacent
    % v of u, and each of this maps contains the indices of the cars
    % currently on (u,v)
    % cars: it is an array of triples where cars(i,:) is (x,y,i) where
    % (x,y) are the current coordinates of the car, and i means that the
    % car is at i-th position of its path (so it will always start at 1)
    % nodes: for every i, nodes(:,i) are the coordinates of node i
    % paths: for every car i, paths(:,i) is the sequence of nodes the car
    % will follow
    % millis: this is the length of a time step
    % travel_times: in the end this will contain the total time a car has
    % travelled
    % safetydist: the safety distance in kilometers a car needs to keep
    % from the following one
    % max_speeds: this is a cell array of maps and max_speeds{u}(v) is just
    % the maximum speed allowed on edge (u,v).
    global graph cars nodes paths millis travel_times safetydist max_speeds;

    millis = 2000;
    safetydist = 0.04;

    travel_times = zeros(numcars, 1);

    % we extract the graph from a textual representation of it
    [nodes,edges] = read_graph(...
        '../data/Graph.txt');

    numnodes = size(nodes,2);

    % we generate the starting and ending position of a journey for every car
    [sources,sinks] = generate_routes(numcars);

    % we compute the shortest path between the two endpoints for every
    % journey
    paths = a_star(nodes,edges,sources',sinks');

    graph = cell(numnodes,1);
    max_speeds = cell(numnodes, 1);

    % we build graph and max_speeds
    for i = 1:numnodes
        graph{i} = containers.Map('KeyType','double','ValueType','any');
        max_speeds{i} = containers.Map('KeyType','double','ValueType','double
            ');
```

```
    end

    numedges = size(edges,2);
    for i = 1:numedges
        u = edges(1, i);
        v = edges(2, i);
        spd = edges(3, i);

        u_graph = graph{u};
        u_graph(v) = containers.Map('KeyType','double','ValueType','logical')
            ;
        v_graph = graph{v};
        v_graph(u) = containers.Map('KeyType','double','ValueType','logical')
            ;

        u_speed = max_speeds{u};
        v_speed = max_speeds{v};

        u_speed(v) = spd;
        v_speed(u) = spd;
    end

    nodes = nodes(2:3,:);

    cars = zeros(numcars,3);
    cars = [nodes(:,int32(sources(:,:)))', ones(numcars, 1)];

    % we insert every car i in the edge it start from, that is
    % (paths(1,i),paths(2,i))
    for i = 1:numcars
        t = graph{paths(1,i)};

        if paths(2,i) == 0
            continue
        end

        t = t(paths(2,i));
        t(i) = 1;
    end
end
```

Moves a given car to the next edge

```
function move_car(i)
    % the meaning of these variables is explained in initialize simulation
    global graph cars nodes paths millis tottime;

    % car i is currently on edge (u,v)
    u = paths(cars(i,3),i);
    v = paths(cars(i,3)+1,i);

    % we remove the car from the edge
    t = graph{u};
    t(v).remove(i);

    % we increase the index of the car position along its route
    cars(i,3) = cars(i,3) + 1;

    % if the path has ended we return
    if cars(i,3) + 1 > size(paths,1) || paths(cars(i,3)+1,i) == 0
        return;
    end

    % we move car i to the next edge, which will now be (u,v)
    u = paths(cars(i,3),i);
    v = paths(cars(i,3)+1,i);

    % we add the car to the edge
    t = graph{u};
    t = t(v);
    t(i) = 1;

    % cars(i,1:2) = nodes(:,paths(cars(i,3),i));
end
```

Reads in the specified graph file in MATLAB data structures

```
function [nodes, edges, node_id_to_idx, idx_to_node_id] = read_graph(filename
    )
        in_file = fopen(filename, 'r');

        num_nodes = fscanf(in_file, '%d\n', 1);
        nodes = fscanf(in_file, '%f %f %f\n', [3, num_nodes]);

        num_edges = fscanf(in_file, '%d\n', 1);
        edges = fscanf(in_file, '%f %f %f\n', [3, num_edges]);

        fclose(in_file);

        node_id_to_idx = containers.Map('KeyType', 'double', 'ValueType', '
            double');
        idx_to_node_id = containers.Map('KeyType', 'double', 'ValueType', '
            double');

        for i = 1:num_nodes
                node_id_to_idx(nodes(1, i)) = i;
                idx_to_node_id(i) = nodes(1, i);
        end

        for i = 1:num_nodes
                nodes(1, i) = node_id_to_idx(nodes(1, i));
        end

        for i = 1:num_edges
                edges(1, i) = node_id_to_idx(edges(1, i));
                edges(2, i) = node_id_to_idx(edges(2, i));
    end
```

Sends a response in JSON format over the socket

```
function response(out,data)
    import com.google.gson.*;

    gson = Gson;
    message = gson.toJson(data);

    out.println('HTTP/1.0 200 OK');
    out.println('Content-Type: text/json');
    out.printf('Content-Length: %d%n',int32(length(message)));
    out.println('Access-Control-Allow-Origin: *');
    out.println();
    out.println(message);
    out.flush();
end
```

Runs the simulation without visualization

```
clearvars;
global graph cars nodes paths millis travel_times over;

% Initialize mean and std vector
all_means = zeros(1, 100);
all_stds = zeros(1, 100);

time_str = get_time_str();

for num_cars = 50:50:5000
        initialize_simulation(num_cars);
        fprintf('num_cars: %d\n', num_cars);

        num_steps = 0;
        over = 0;
        while 1

            if over
               break;
            end

            advance_simulation();
        end

        fprintf('mean: %d\nstd: %d\n', mean(travel_times), int32(std(
            travel_times)));

        % update mean and std vector
        all_means(int32(num_cars / 50)) = mean(travel_times);
        all_stds(int32(num_cars / 50)) = std(travel_times);

        % write data to file after every iterations
        csvwrite(strcat(time_str, '_mean_travel_time.csv'), all_means);
        csvwrite(strcat(time_str, '_std_travel_time.csv'), all_stds);
end
```

Runs the simulation with visualization

```
javaaddpath('gson-2.2.4.jar');
import com.google.gson.*;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;
import java.io.BufferedReader;
import java.io.InputStreamReader;

clearvars;
% the meaning of these variables is explained in initialize simulation
global graph cars nodes paths millis travel_times over;

listener = ServerSocket(int32(8383));

init = 0;
json = Gson;

c = onCleanup(@() listener.close());

while 1

    socket = listener.accept();

    instream = socket.getInputStream();
    outstream = socket.getOutputStream();

    in = BufferedReader(InputStreamReader(instream));

    message = -1;
    line = in.readLine();
    if(~isempty(line) && ~isnumeric(line) && line.length() >= 6)
        line = line.substring(5,6);
        message = char(line);
    end

    out = PrintWriter(outstream);

    if message == '1'
        'initialize_simulation'
        init = 1;
        initialize_simulation(1000);
    elseif message == '0'
        'stop_simulation'
        init = 0;
        break;
    elseif message ~= '2'
```

```
            'skip'
            continue;
        end

        'advance_simulation'

        if init
            data = cars(:,1:2);
            if over
                response(out,[]);
                break;
            end

            response(out,data);
            advance_simulation();
        end

        socket.close();
    end
    listener.close();

    fprintf('mean: %d\n std: %d\n', mean(travel_times), int32(std(travel_times)))
        ;
```

Runs the simulation with text inputs

```
javaaddpath('gson-2.2.4.jar');
import com.google.gson.*;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;
import java.io.BufferedReader;
import java.io.InputStreamReader;

% the meaning of these variables is explained in initialize simulation
global graph cars nodes paths millis tottime safetydist defaultspeed;

json = Gson;
tottime = 0;
stop = 0;
while not(stop)

    message = input('command: ','s');

    if(message == '1')
        initialize_simulation();
        'initialize_simulation'
    elseif (message == '2')
        advance_simulation();
        'advance_simulation'
    elseif (message == '0')
        'stop_simulation'
        stop = 1;
        continue
    else
        'skip'
        continue
    end

    data = cars(:,1:2)
end
```

C Python Code

Fix OpenStreetMap Ways

```
#!/usr/bin/python

import re
import sys


def main():
    """
    This script expects an OpenStreetMap.
    It removes all invalid node references,
    i.e. it just keeps those where the id was defined.
    """

    # Check if file name is given as command line argument,
    # else query user for it
    my_file = raw_input() if len(sys.argv) == 1 else sys.argv[1]

    # initialize set of nodes
    nodes = {}

    # iterate through all lines in the given file
    with open(my_file) as f:
        for line in f:
            if "node_id" in line:
                # if "node id" is present in the given line split on
                # quotation mark and store extracted id in the set
                node = int(re.split("['\"]", line)[1])
                nodes[node] = 1

            elif "nd_ref" in line:
                # if "nd ref" is present in the given line check
                # if the current id was encountered before, if not skip
                node = int(re.split("['\"]", line)[1])
                if node not in nodes:
                    continue

            print line,


if __name__ == '__main__':
    main()
```

Get the maximum allowed speeds from OpenStreetMap

```
#!/usr/bin/env python
import sys
import xml.etree.ElementTree as ET
from math import floor

def print_help():
    """
    Prints usage message.
    """
    print 'Usage: {} OpenStreetMap.osm'.format(
        sys.argv[0])


def main():
    """
    Check if enough arguments were provided.
    If not print usage message and exit.
    """
    if len(sys.argv) < 2:
        print_help()
        return

    """
    Parse the input file.
    """
    tree = ET.parse(sys.argv[1])
    root = tree.getroot()

    """
    For each node check if the id is allowed.
    """

    mean_speeds = {}

    for way in root.iter('way'):
        speed = None
        highway = None
        for tag in way.iter('tag'):
            if tag.get('k') == 'maxspeed':
                speed = int(tag.get('v'))
            elif tag.get('k') == 'highway':
                highway = tag.get('v')

        if speed is not None and highway is not None:
            if highway not in mean_speeds:
                mean_speeds[highway] = [speed, 1]
            else:
                mean_speeds[highway][0] += speed
```

```
                mean_speeds[highway][1] += 1

    for street_type in mean_speeds:
        data = mean_speeds[street_type]
        avg = data[0] / float(data[1])
        avg = floor((avg + 2.5) / 5.0) * 5.0

        mean_speeds[street_type] = avg

    print mean_speeds
        # print "{}: {} ({})".format(street_type, avg, data)

if __name__ == '__main__':
    main()
```

Extract Graph from OpenStreetMap

```
#!/usr/bin/python
import sys
import xml.etree.ElementTree as ETree

# This data was extracted from the Zurich.osm file.
# The listed speeds are average maxspeeds for the indivudual road types
# rounded to the next multiple of 5.0
max_speeds = {
    'primary_link': 50.0,
    'residential': 30.0,
    'secondary_link': 50.0,
    'primary': 50.0,
    'motorway_link': 75.0,
    'motorway': 90.0,
    'trunk': 70.0,
    'trunk_link': 60.0,
    'unclassified': 40.0,
    'tertiary': 50.0,
    'secondary': 50.0,
    None: 50.0                  # this serves as the default value, we should
        never have to use this
}

def print_help():
    """
    Prints usage message.
    """
    print 'Usage: {} File.osm [Graph.txt]'.format(sys.argv[0])


def rem_ext(filename):
    """
    Takes a filename as arugment and returns it with the file extension
        removed.
    Returns the original string if no extension could be found.
    """
    last_dot_idx = filename.rfind('.')
    return filename[:last_dot_idx] if last_dot_idx != -1 else filename


def main():
    """
    Check if enough arguments were provided.
    If not print usage message and exit.
    """
    if len(sys.argv) < 2:
        print_help()
        return
```

```
in_file = sys.argv[1]
out_file = sys.argv[2] if len(sys.argv) > 2 else '{}_Graph_Speeds.txt'.
    format(rem_ext(in_file))

# Parse the input file.
tree = ETree.parse(in_file)
root = tree.getroot()

# Open the output_file for writing.
f = open(out_file, 'w')

# Count number of nodes and write it to the file.
num_nodes = len(root.findall('node'))
f.write(str(num_nodes) + '\n')

# For each node print the id together with latitude and longitude.
for node in root.iter('node'):
    f.write('{} {} {}\n'.format(node.attrib['id'], node.attrib['lat'],
        node.attrib['lon']))

# Count the number of node references.
# Subtract one to account for the fact the we do not print a line after
# processing the first reference in a way.
num_nds = 0
for way in root.iter('way'):
    num_nds = num_nds + max(len(way.findall('nd')) - 1, 0)

# Write number of edges.
f.write(str(num_nds) + '\n')

for way in root.iter('way'):
    highway = None
    maxspeed = None

    for tag in way.iter('tag'):
        if tag.get('k') == 'highway':
            highway = tag.get('v')
        elif tag.get('k') == 'maxspeed':
            maxspeed = float(tag.get('v'))

    if highway is None:
        print "Warning: way {} has no highway tag".format(tag.way('id'))

    if maxspeed is None:
        if highway not in max_speeds:
            highway = None
        maxspeed = max_speeds[highway]

    prev_nd = None
```

```
            # For each way print all edges between the referenced nodes and the
                allowed speed.
            for nd in way.iter('nd'):
                if prev_nd is not None:
                    f.write('{} {} {}\n'.format(prev_nd.attrib['ref'], nd.attrib[
                        'ref'], maxspeed))

                prev_nd = nd

        # Close the file.
        f.close()


if __name__ == '__main__':
    main()
```

Remove unwanted Nodes from OpenStreetMap

```
#!/usr/bin/env python
import sys
import xml.etree.ElementTree as ET


def print_help():
    """
    Prints usage message.
    """
    print 'Usage: {} File.osm Node_Blacklist.txt [New_Graph.osm]'.format(
        sys.argv[0])


def main():
    """
    Check if enough arguments were provided.
    If not print usage message and exit.
    """
    if len(sys.argv) < 3:
        print_help()
        return

    """
    Parse the input file.
    """
    tree = ET.parse(sys.argv[1])
    root = tree.getroot()

    """
    Allocate space for the allowed nodes
    """
    forbidden_nodes = set()
    graph_file = sys.argv[2]

    """
    Parse the graph file and extract the node ids
    """
    with open(graph_file, 'r') as g:
        for line in g:
            forbidden_nodes.add(int(line))

    """
    Open the output_file for writing.
    """
    out_file = sys.argv[3] if len(sys.argv) > 3 else '{}_Single_Component.osm
        '.format(
        sys.argv[1][:-4])
    f = open(out_file, 'w')
```

```
    """
    For each node check if the id is allowed.
    """
    for node in list(root.iter('node')):
        node_id = int(node.attrib['id'])
        if node_id in forbidden_nodes:
            root.remove(node)

    """
    Remove invalid node references
    """
    ways = list(root.iter('way'))
    for way in ways:
        # root.remove(way)
        """
        For each way check if the referenced node is allowed.
        """
        nds = list(way.iter('nd'))
        for nd in nds:
            node_id = int(nd.attrib['ref'])
            if node_id in forbidden_nodes:
                way.remove(nd)

        if len(list(way.iter('nd'))) == 0:
            root.remove(way)

    f.write(ET.tostring(root))

    """
    Close the file.
    """
    f.close()


if __name__ == '__main__':
    main()
```

Simplify an OpenStreetMap Graph

```
#!/usr/bin/python
import sys
import xml.etree.ElementTree as ET


def print_help():
    """
    Prints usage message.
    """
    print 'Usage: {} File.osm Simple_Graph.txt [New_Graph.osm]'.format(
        sys.argv[0])


def main():
    """
    Check if enough arguments were provided.
    If not print usage message and exit.
    """
    if len(sys.argv) < 3:
        print_help()
        return

    """
    Parse the input file.
    """
    tree = ET.parse(sys.argv[1])
    root = tree.getroot()

    """
    Allocate space for the allowed nodes
    """
    allowed_nodes = set()
    graph_file = sys.argv[2]

    """
    Parse the graph file and extract the node ids
    """
    with open(graph_file, 'r') as g:
        num_nodes = int(g.readline())

        for x in xrange(num_nodes):
            line = g.readline()
            allowed_nodes.add(int(line.split()[0]))

    """
    Open the output_file for writing.
    """
    out_file = sys.argv[3] if len(sys.argv) > 3 else '{}_Reduced.osm'.format(
```

```
        sys.argv[1][:-4])
    f = open(out_file, 'w')

    """
    For each node check if the id is allowed.
    """
    for node in list(root.iter('node')):
        node_id = int(node.attrib['id'])
        if node_id not in allowed_nodes:
            root.remove(node)

    """
    Remove invalid node references
    """
    ways = list(root.iter('way'))
    for way in ways:
        # root.remove(way)
        """
        For each way check if the referenced node is allowed.
        """
        nds = list(way.iter('nd'))
        for nd in nds:
            node_id = int(nd.attrib['ref'])
            if node_id not in allowed_nodes:
                way.remove(nd)

        if len(list(way.iter('nd'))) == 0:
            root.remove(way)

    f.write(ET.tostring(root))

    """
    Close the file.
    """
    f.close()


if __name__ == '__main__':
    main()
```

D JavaScript Code

Visualize the Simulation

```
<!DOCTYPE html>
<html>
<head>
    <title>Leaflet Quick Start Guide Example</title>
    <meta charset="utf-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <link rel="stylesheet" href="../dist/leaflet.css" />
    <link rel="stylesheet" href="VisualizeSimulation.css" />
</head>
<body>
    <script src="../dist/leaflet.js"></script>
    <script src="../jQuery/jquery-1.11.1.js"></script>

    <div id="map"></div>

    <script>
        /**  Global variables */
        var port = 8383;                                    // port to
            communicate
        var host = "http://localhost:" + port + "/";        // host name
        var stop = false;                                   // indicator variable
        var millis = 50;                                    // update interval


        /**
         * Issues an HTML GET request to the specified URL
         * @param  {string} theUrl Requested URL
         * @return {string}        response text
         */
        function httpGet(theUrl)
        {
            var xmlHttp = null;

            xmlHttp = new XMLHttpRequest();
            xmlHttp.open( "GET", theUrl, false );
            xmlHttp.send( null );
            return xmlHttp.responseText;
        };


        /**
         * Initializes the cars on the map.
         * Requests the page "1" from the host to do so.
```

```
 * Then parses the response as a JSON object and returns it.
 * @return {Double Array} Initial car coordinates
 */
function init_cars()
{
    var request = httpGet(host + "1");
    var obj = $.parseJSON(request);
    console.log("init");
    return obj;
};


/**
 * Updates the cars on the map.
 * Requests the page "2" from the host to do so.
 * Then parses the response as a JSON object and returns it.
 * @return {Double Array} Updated car coordinates
 */
function update_cars()
{
    var request = httpGet(host + "2");
    var obj = $.parseJSON(request);
    console.log("update");
    return obj;
};


/**
 * Sleep function.  Pauses execution of the callback function for
     millis ms.
 * @param  {int}   millis   Number of milliseconds
 * @param  {Function} callback Function which execution is paused.
 * @return {void}
 */
function sleep(millis, callback) {
    setTimeout(function() {
        callback();
    }, millis);
};

/**
 * This function adds circles at the specified coordinates.
 * @param {Double Array} coords Coordinate Vector
 */
function add_circles(coords) {
    for (var i = 0; i < coords.length; i++)
    {
        L.circle(coords[i], 30, {
              stroke: false,
            fillColor: 'red',
```

```
            fillOpacity: 0.5
        }).addTo(markers);
    };
};


/**
 * Updates the circles on the map by changing their coordinates
 * and redrawing them.
 * @param  {[Double Array]} coords [New positions]
 */
function redraw_circles(coords) {
    var circles = markers.getLayers();

    for (var i = 0; i < Math.min(circles.length, coords.length); i++)
    {
        circles[i].setLatLng(coords[i]);
        circles[i].redraw();
    };
};


/**
 * Wrapper around redraw circles that detects the end of the
    simulation.
 * This is needed to be able to sleep recursively.
 */
function update_circles()
{
    var coords = update_cars();
    if(coords == null) {
        stop = true;
        console.log("Simulation stopped!");
        return;
    }
    sleep(millis,update_circles);
    redraw_circles(coords);
};


/** Zurichs position in latitude and longitude */
var zurich_lat_lon = [47.3685586, 8.5404434];

/** Leaflet map object */
var map = L.map('map').setView(zurich_lat_lon, 13);


/** initializes the map to the correct view and sets copyright */
L.tileLayer('https://{s}.tiles.mapbox.com/v3/{id}/{z}/{x}/{y}.png', {
    maxZoom: 18,
```

```
            attribution: 'Map data &copy; <a href="http://openstreetmap.org">
                OpenStreetMap</a> contributors, ' +
                '<a href="http://creativecommons.org/licenses/by-sa/2.0/">CC-
                    BY-SA</a>, ' +
                'Imagery ?? <a href="http://mapbox.com">Mapbox</a>',
            id: 'examples.map-9ijuk24y'
        }).addTo(map);

        /** marker layer holding the circles in the end */
        var markers = new L.LayerGroup().addTo(map);

        /** initialization of the cars */
        var coords = init_cars();
        add_circles(coords);

        /** starting the update process */
        sleep(millis, update_circles);
    </script>
</body>
</html>
```