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 worsen) the traffic problem of the city center. More specifically, the traveling 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 analyzed.

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 Automaton 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)

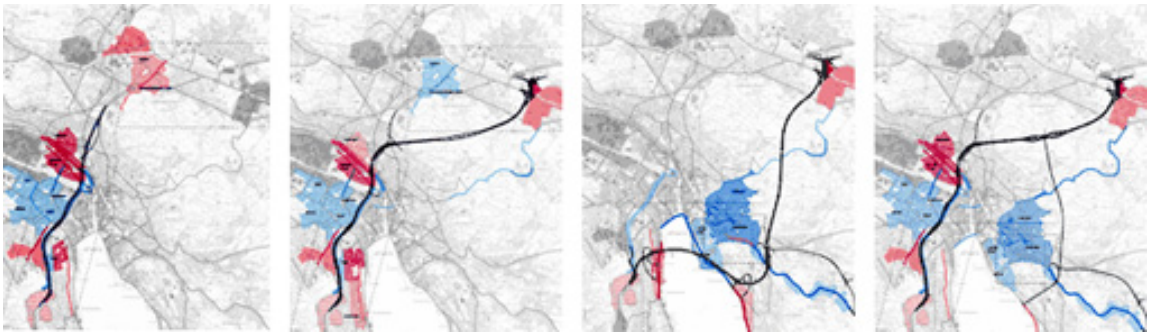

Figure 1: The four different solutions proposed in "Highway Tunnel Zürich" project [GGa14]

With the present case study the goal is to develop an appropriate and efficient traffic model based on the available public data. This model can be used to visualize 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 analyzed in order to evaluate the impact of such a tunnel. As far as it concerns the expected results, it is not easy to make a prediction. 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 misjudgment.

4 Description of the Model

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

- **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 we developed is loosely 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 two main parts:

- **Journey generation**: for every automaton we will generate a path that it will follow; this route will be fixed 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 CSVs files population.csv and companies.csv; what we aimed at modeling 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 south of the river which we aggregated as incoming and 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.
- **Route planning**: 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

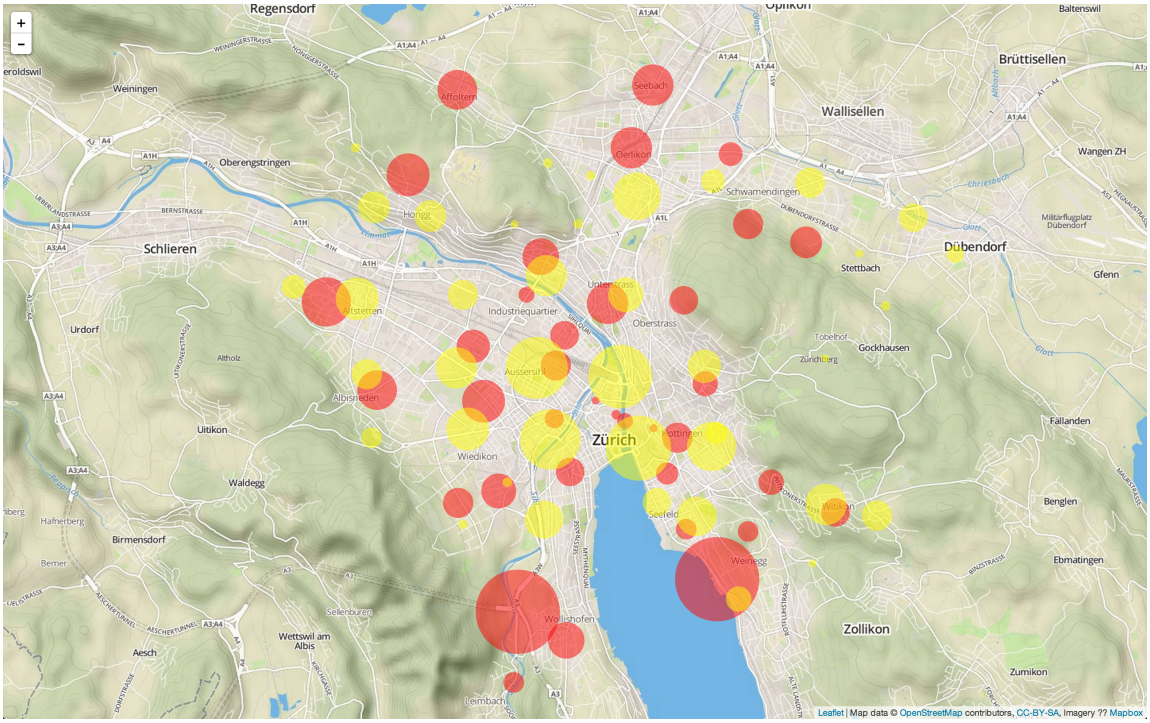

Figure 2: Visualization of the chosen population and company hubs. Red dots how population areas and yellow dots company ares.

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.

- **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.

5 Implementation

We started out by obtaining the OpenStreetMap 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* [Wik14]. 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

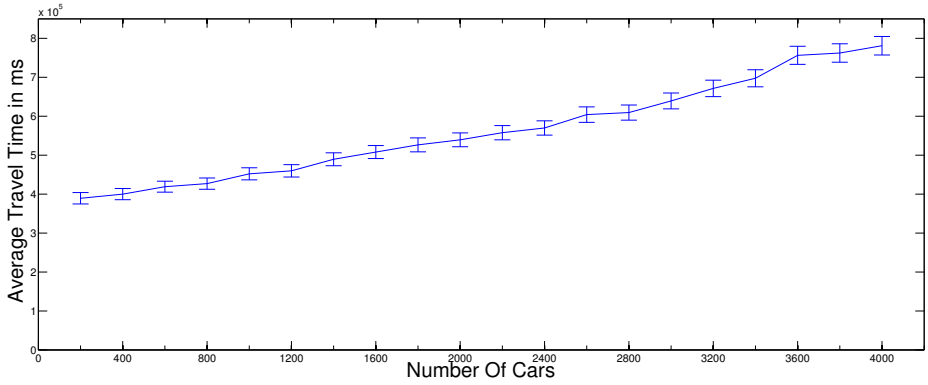

Figure 3: Plot of travel times with regard to increasing number of cars on the road

implementation then made sure to display the cars at the right locations on the map.

6 Simulation Results and Discussion

6.1 In correlation with the number of cars

The figure that is presented below, shows that 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 limitations of the existing road network (there is no projection for a huge number of cars in the city of Zürich). More specifically there is an average increase of around 25% to the travelling time when the number of cars is going from 400 to 1600, more than 50% when the number of cars changes from 2000 to 4000, and finally a massive 100% when the number of cars increases from 400 to 4000.

6.2 In correlation with the chosen position for the tunnel construction

From the sixteen different scenarios that are evaluated it comes out as a conclusion that with the available data, the alternative positions for the tunnel construction do not influence that much the average time that a driver needs to move from his starting point to his destination. This probably happens because all these sixteen alternatives are close enough 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.

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 reduced in the case the tunnel is built. The whole traffic situation in the centre of the city is improved and there are less conjunction points compared to the initial situation. More specifically, it is underlined by the results that...

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

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 question defined in the beginning of the research was answered and the conclusion was that the tunnel could improve the travel time for the benefit of the drivers using the road network of Zürich. 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).

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

[Wik14] OpenStreetMap Wiki. *Osmconvert*. 2014. URL: http://wiki.openstreetmap.org/w/index.php?title=Osmconvert&oldid=1010333 (visited on 02/03/2014).

Appendix

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];
}
```

MATLAB Code

MATLAB Wrapper

```
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
```