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 Wilken Dörrie Simone Forte Charalampos Gkonos Athina Korfiati

Contents

1 Abstract

In this paper, a traffic simulation for the city of Zurich 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 Zurich, 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, and 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

3 Introduction and Motivations

4 Description of the Model

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 Zurich. 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 Zurich 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 Zurich; this has been necessary in order to evaluate the impact of a tunnel in that area.

- **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 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 extracting the OpenStreetMap file that contains Zurich. In order to do so we downloaded a recent Switzerland extract from *Geofabrik* [Geo14] and then extracted the boundaries of Zurich with the tool *Osmconvert* [Wik14]. Afterwards we used another tool called *Osmosis*[Wik12] to limit our data to the roads in Zurich. 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 Zurich we wrote an A-Star algorithm implementation in C++. We considered doing this directly in Matlab, however the performance loss was too large, so that we wrote the mex file.

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 Zurich. We enabled communication between MATLAB and Javascript using the included Java classes for Sockets. The Javascript code issues HTTP GET requests telling the MATLAB implementation which methods to perform. Afterwards the location of every car in Zurich was passed as an JSON object through the socket.

6 Simulation Results and Discussion

7 Summary and Outlook

References

[Geo14] Geofabrik. *Switzerland*. 2014. URL: http://download.geofabrik.de/europe/switzerland.html/ (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

A Star 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];
}
```