|  |
| --- |
| tECHNISCHE uNIVERSITÄT bERLIN  iNSTITUT FÜR Land- und Seeverkehr  Fachgebiet VERkehrssystemplanung und Verkehrstelematik  pROF. dR. KAi Nagel  Multiagenten-simulationen von Verkehr, Sommersemester 2014 |
| Hausaufgabe 1 |
| Untersuchungsraum Cottbus und Umgebung |
| Robin Laier, 321 452  Janek Laudan, --- --- |
|  |
| 22.06.2014 |

Inhaltsverzeichnis

[1 Aufgabe 1 3](#_Toc391732732)

[2 Aufgabe 2 4](#_Toc391732733)

[3 Aufgabe 3 6](#_Toc391732734)

[4 Aufgabe 4 8](#_Toc391732735)

[5 Aufgabe 5 12](#_Toc391732736)

[6 Aufgabe 6 13](#_Toc391732737)

[7 Aufgabe 7 14](#_Toc391732738)

[8 Anhang 15](#_Toc391732739)

[8.1 TripDurationDifference 15](#_Toc391732740)

[8.2 TripDurationAnalyser 18](#_Toc391732741)

[8.3 BridgeUserAnalyser 19](#_Toc391732742)

|  |
| --- |
|  |

# Aufgabe 1

Als Grundlage für die folgenden Aufgaben wird zunächst ein zweistufiges Straßennetz des Untersuchungsgebietes erstellt. Die erzeugte XML-Netzwerkdatei kann von der Multiagentensimulation „MATSim“ verwendet werden. Zunächst wird die Osmosis Java Application heruntergeladen und zum Projekt hinzugefügt. Anschließend wird unter dem Link <http://download.geofabrik.de/europe/germany/brandenburg.html> die OSM-Datei des Untersuchungsgebietes heruntergeladen. Über <http://www.openstreetmap.org> wird nun mithilfe von Geokoordinaten der Großbereich Cottbus erfasst. In Eclipse werden dann drei Java Configurations zum Projekt hinzugefügt: Run – Run Configurations – New launch configuration. Die erste configuration dient der Verarbeitung des Gebietes um Cottbus und beinhaltet die eben ermittelten Koordinaten. Hier werden alle Straßenkategorien hinzugefügt (daher completeWays=true), indem der folgende Code als program arguments verwendet wird:

--rb file=input\_nullfall/brandenburg.osm.pbf --bounding-box top=51.8205 left=14.1631 bottom=51.6936 right=14.4872 completeWays=true --used-node --wb input\_nullfall/cottbus.osm.pbf

Danach wird eine Configuration für den Rest von Brandenburg erstellt. Hierbei sollen von der .osm.pbf-Inputdatei nur die acht wichtigsten Straßenkategorien übernommen werden (motorway, motorway\_link, trunk, trunk\_link, primary, primary\_link, secondary, secondary\_link):

--rb file= input\_nullfall/brandenburg.osm.pbf --tf accept-ways highway=motorway,motorway\_link,trunk,trunk\_link,primary,primary\_link,secondary,secondary\_link --used-node --wb input\_nullfall/brandenburg-bigroads.osm.pbf

Schließlich werden diese beiden Netzwerke mithilfe einer dritten Configuration zusammengefügt:

--rb file=input/cottbus.osm.pbf --rb input\_nullfall/brandenburg-bigroads.osm.pbf --merge --wx input\_nullfall/br-co-merged-network.osm

Damit das so erstellte Netzwerk weiterverwendet werden kann, wird es anschließend mit der Klasse CreateNetwork in eine XML-Datei umgewandelt. Dies geschieht mit dem folgenden Code, der auch die Umwandlung des Koordinatensystems in WGS84\_UTM33N umfasst:

public class CreateNetwork {  
 public static void main(String[] args) {  
 String osm = "input\_nullfall/cottbus\_merged\_network.osm";  
 Config config = ConfigUtils.createConfig();  
 Scenario sc = ScenarioUtils.createScenario(config);  
 Network net = sc.getNetwork();  
 CoordinateTransformation ct =   
 TransformationFactory  
 .getCoordinateTransformation(TransformationFactory.WGS84,  
 TransformationFactory.WGS84\_UTM33N);  
 OsmNetworkReader onr = new OsmNetworkReader(net,ct);   
 onr.parse(osm);   
 new NetworkCleaner().run(net);  
 new NetworkWriter(net)  
 .write("input\_nullfall/br-co-merged-network-xml.xml");  
 }  
}

Diese Vorgehensweise führt zu einer xml-Netzwerkdatei, die im definierten Bereich um Cottbus Straßen aller Kategorien abbildet, im restlichen Brandenburg hingegen nur die definierten Straßenkategorien.

# Aufgabe 2

Zunächst wird die Anzahl von Pendlern zwischen den Landkreisen für jede Paarung von Landkreisen mit Hilfe des auf ISIS bereitgestellten *ReadPendler* Scripts ermittelt. Anschließend werden Pläne für eine Samplegröße von 10% erzeugt. Dies geschieht im Wesentlichen mit folgender Methode:

private void writePlansFor(String fromKreis, String toKreis, int commuters)

{

//Vorbereitender Code

//..

for(int i = 0; i<=comm; i++)

{

String mode = "car";

if(i > carcomm)

mode = "pt";

Coord homec = drawRandomPointInTwoGeometries(this.shapeMap.get(fromKreis), buildingsAsArray);

Coord workc = drawRandomPointInTwoGeometries(this.shapeMap.get(toKreis), buildingsAsArray);

createOnePerson(i, homec, workc, mode, fromKreis + " - " + toKreis);

}

}

Für jeden erzeugten Pendler wird ein Mode-of-Transportation festgelegt. Der Modal Split beträgt 53% MIV und 47% ÖPV (Quelle: http://www.mil.brandenburg.de/media\_fast/4055/Erster%20Fahrradbericht%20f%C3%BCr%20das%20Land%20Brandenburg.pdf, S.14 auf 53% festgelegt (car) der Rest ist die Summe aus Fahrrad, ÖPNV und Fußgänger.) Es werden außerdem Koordinaten für den Wohn-, sowie Arbeitsort festgelegt. Beide Orte sollen sowohl im richtigen Landkreis als auch in einem Gebäude verortet sein. Folgende Methodik wird daher in der Methode *drawRandomPointInTwoGeometries* angewandt:

while(true)

{

int random = rnd.nextInt(size);

Geometry building = buildings[random];

if (landkreis.contains(building)) {

Point center = building.getCentroid();

result = new CoordImpl(center.getX(), center.getY());

break;

}

else

continue;

}

return result;

Au seiner Liste mit Gebäudeshapes wird zufällig eine Shape ausgewählt. Es wird anschließend überprüft, ob die Gebäudeshape innerhalb des gewünschten Landkreises liegt. Ist dies der Fall wird der Mittelpunkt des Gebäudes als Koordinate zurückgegeben. Liegt das Gebäude nicht im gewünschten Landkreis werden solange zufällig ausgewählte Gebäude ausprobiert, bis eines innerhalb des Landkreises liegt.

Schlussendlich werden den Aktivitäten *home* und *work* eine Endzeit, der Aktivität *work*  darüber hinaus eine Startzeit zugewiesen. Der Beginn der Arbeitszeit liegt dabei zwischen sechs und neun Uhr, die Arbeitszeit zwischen sieben und zehn Stunden.

//create random time to leave home

**double** start = Math.*random*() \* 3 + 6; //starting time between 6 and 9

**double** end = Math.*random*() \* 3 + 7; //end time between 7 and 10 hours later than start

end = end + start;

home.setEndTime(start \* 60 \* 60);

plan.addActivity(home);

Leg hinweg = scenario.getPopulation().getFactory().createLeg(mode);

plan.addLeg(hinweg);

Activity work = scenario.getPopulation().getFactory().createActivityFromCoord("work", coordWork);

work.setStartTime(start \* 60 \* 60);

work.setEndTime(end \* 60 \* 60);

plan.addActivity(work);

Leg rueckweg = scenario.getPopulation().getFactory().createLeg(mode);

plan.addLeg(rueckweg);

Activity home2 = scenario.getPopulation().getFactory().createActivityFromCoord("home", coord);

plan.addActivity(home2);

person.addPlan(plan);

scenario.getPopulation().addPerson(person);

# Aufgabe 3

Basierend auf den Daten aus Aufgabe 2 werden nun Iterationen durchgeführt. Von Relevanz ist dabei vor allem das Verhalten der Agenten. Dies kann in der Konfigurationsdatei config.xml eingestellt werden. Diese Datei wird folgendermaßen angepasst:

* Im Modul „network“ wird der Pfad der Network-Datei angegeben.
* Im Modul „plans“ wird der Pfad der erstellten Plans-Datei angegeben.
* Im Modul „controler“ wird die Anzahl der Iterationen angegeben. Das Ermitteln der kürzesten Reisezeit erfordert nur eine Iteration, für die weiteren Untersuchungen werden zwischen 10 und 50 Iterationen durchgeführt.
* Im Modul „qsim“ kann unter anderem die Kapazität des Netzes festgelegt werden.

Als erster Schritt wird die durchschnittliche Reisezeit im ungestauten Netz bestimmt. Die Werte „flowCapacityFactor“ und „storageCapacityFactor“ werden dabei auf 1 gesetzt. Nach eine Iteration ergibt sich eine Reisezeit von 30 Minuten und 11 Sekunden, einzusehen in der von MATSim erstellten Datei /output/ITERS/it.0/0.tripdurations.txt.

Anschließend werden die beiden Kapazitätsfaktoren analog zu der 10%-Stichprobengröße der Population ebenfalls auf 10% bzw. 0.1 herabgesetzt. Es werden 50 Iterationen durchgeführt. Im Ergebnis beträgt die durchschnittliche Reisezeit bereits nach einer Iteration nur 30 Minuten und 24 Sekunden, und das Histogramm des durchschnittlichen Scores verläuft annähernd linear. Grund hierfür könnte sein, dass in der Simulation nur Pendlerverkehre erfasst werden. In der Realität setzt sich der Verkehr jedoch noch aus weiteren Verkehren zusammen, weshalb auch die Infrastruktur im Allgemeinen für mehr Verkehr ausgelegt ist.

Beim nächsten Iterationslauf werden die beiden Faktoren auf 0.03 herabgesetzt. Die ReRoute Strategy erhält einen Wert von 0.2 und wir nach 7 Iterationen deaktiviert. Die ChangeExpBeta Strategy erhält einen Wert von 0.8 und wird nie deaktiviert (Value=null). Es werden 20 Iterationen durchgeführt. Dabei ergibt sich das in Abbildung 1 dargestellte Scorestats-Histogramm. Die stärksten Veränderungen der Scores finden in den ersten vier Iterationen statt, danach verlaufen die Kurven annähernd linear und mit nur sehr geringer Steigung. Die durchschnittliche Trip-Dauer beträgt bei Iteration 0 noch 55 Minuten und 53 Sekunden und sinkt auf 35 Minuten und 08 Sekunden bei Iteration 10 und 34 Minuten und 50 Sekunden bei Iteration 20. Das Leg Histogram für den Pkw-Verkehr ist in Abbildung 2 dargestellt und weist nach der ersten Iteration noch eine ausgeprägte Spitze im Vormittagsverkehr auf. Gegen 9:00 Uhr befinden sich fast 1400 Fahrzeuge auf den Straßen.

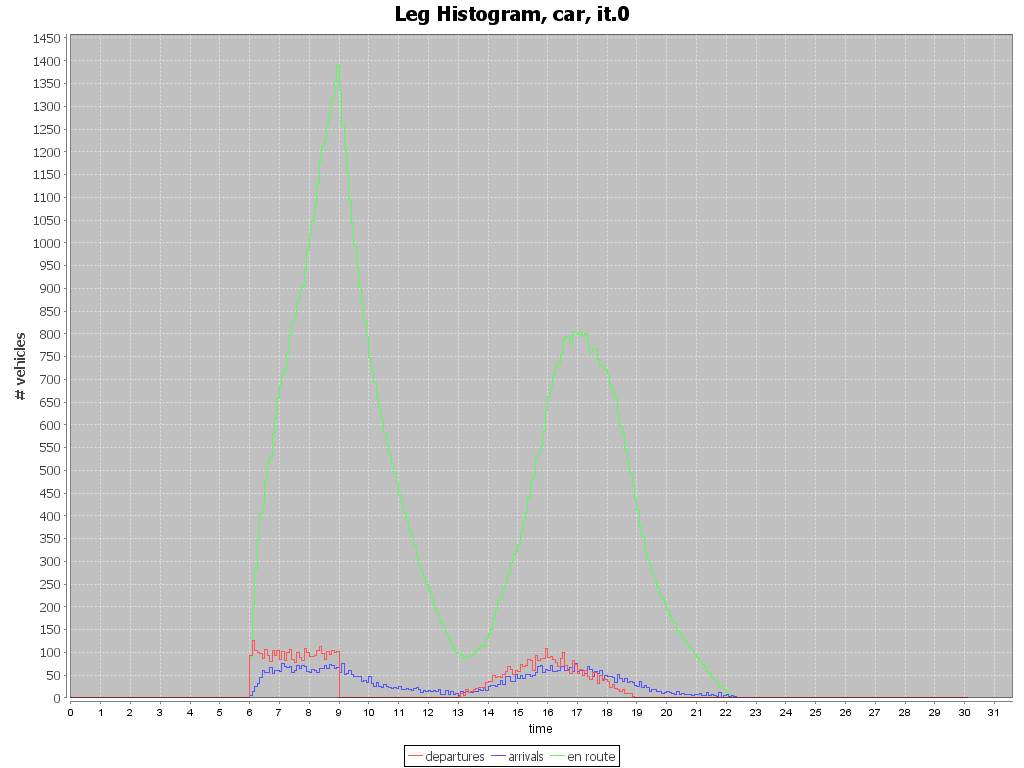


Abbildung : Score Statistics des zweiten Durchlaufs

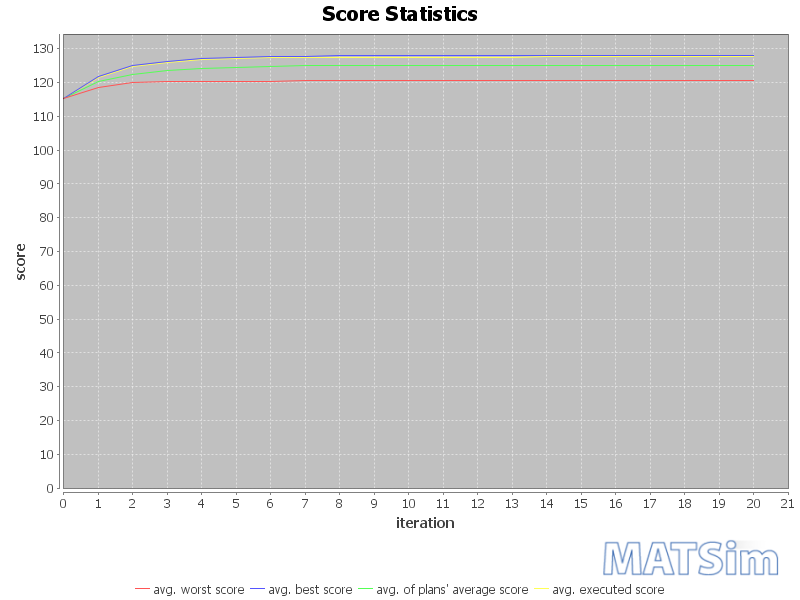


Abbildung : Vormittagsspitze im Pkw-Verkehr nach Iteration 0

Dieser „Peak“ in der Fahrzeuganzahl glättet sich im Laufe der Iterationen ab, wie das in Abbildung 3 dargestellte Histogramm nach der 20. Iteration zeigt. Es gibt weiterhin eine Verkehrsspitze, diese ist jedoch um ca. 8:25 Uhr und es sind hier weniger als halb so viele Pkw unterwegs als nach der ersten Iteration. Der Verkehr verteilt sich über einen längeren Zeitraum und versiegt um die Mittagszeit fast komplett.

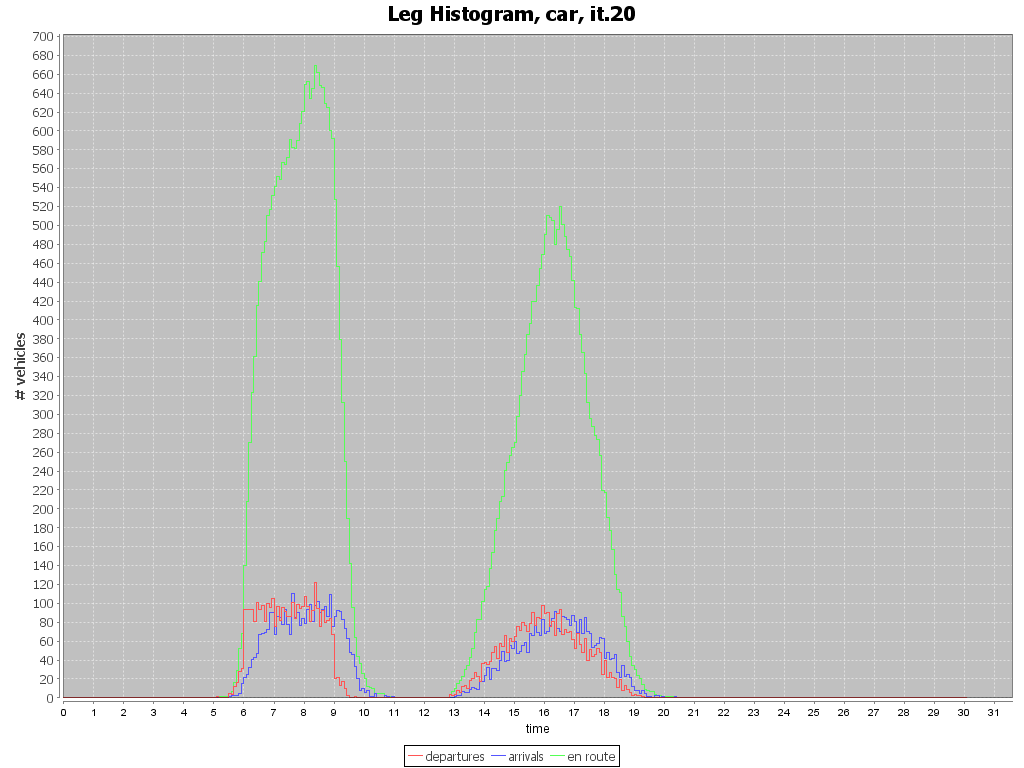


Abbildung : Leg Histogram nach der 20. Iteration

In einem weiteren Durchlauf werden der „flowCapacityFactor“ und der „storageCapacityFactor“ auf 0.05 und 0.08 hochgesetzt; die übrigen Werte werden beibehalten. Die dabei entstehenden Plots werden separat abgespeichert, um einen Vergleich mit folgenden Iterationsdurchläufen anstellen zu können.

# Aufgabe 4

Im Rahmen der vierten Aufgabe werden die „freespeed“ und die „capacity“ der Spreebrücken Franz-Mehring-Straße und Sandower Hauptstraße stark verringert. Damit soll eine „Sperrung“ dieser Brücken simuliert werden. Zu diesem Zweck wird die network.xml-Datei mit dem Visualisierungsprogramm Senozon VIA eingelesen. Die beiden Brücken werden mit dem Info-Tool (Reiter „Layers“, Fragezeichen-Symbol in der linken Spalte im Bereich „Network“) ausgewählt und deren Link ID notiert. Im konkreten Fall handelt es sich um die Links 244 und 245 (Sandower Hauptstraße) und die Links 19466 und 19467 (Franz-Mehring-Straße). Die Brücken haben jeweils zwei Links, da sie in beiden Fahrtrichtungen befahrbar sind. Die Werte „freespeed“ und „capacity“ für die vier Links werden nun in der Network-Datei auf 0.01 bzw 1.0 herabgesetzt. Dann werden die Iterationen mit den bereits erstellten Plänen erneut durchgeführt. Im Ergebnis ist zu beobachten, dass die Brücken von keinem einzigen Agenten mehr befahren werden (Abbildung 4). Nach Iteration 0 kommt es vor allem im Bereich des Nordrings, an der Kreuzung Stadtring/Gustav-Hermann-Straße und auf der Bahnhofstraße zu verlängerten Fahrzeiten während des vormittäglichen Berufsverkehrs. Ein ähnliches Bild ergibt sich zwischen 15:30 Uhr und 17:30 Uhr. Es wird auch ersichtlich, dass im Bereich innerhalb des Stadtrings kein Durchgangsverkehr mehr anfällt.

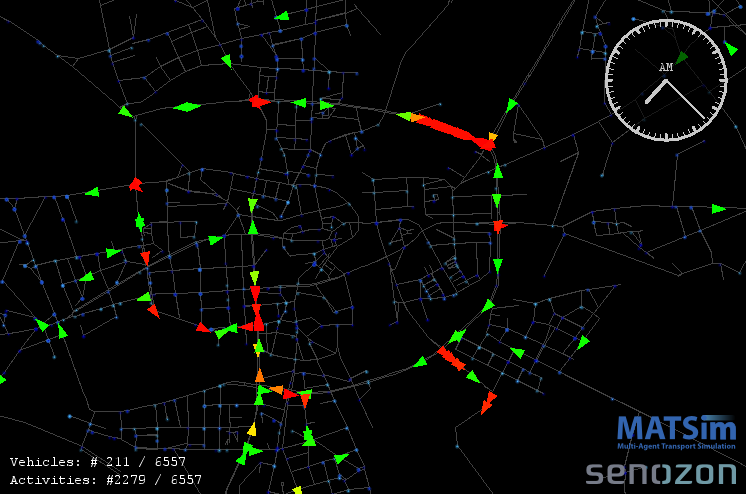


Abbildung : Morgendlicher Berufsverkehr bei Iteration 0

Nach Iteration 20 sieht die Situation zur gleichen Uhrzeit deutlich unkritischer aus (Abbildung 5). Der Verkehr verteilt sich besser auf das vorhandene Straßennetz. Überlastungen treten nur noch stark örtlich und zeitlich begrenzt auf.

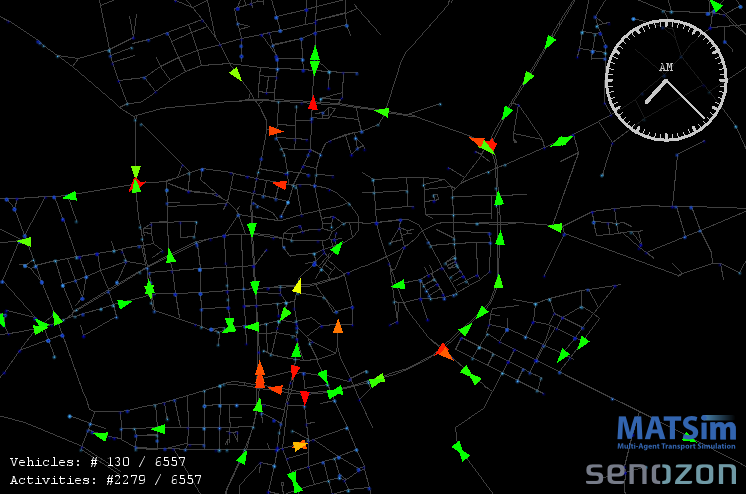


Abbildung : Morgendlicher Berufsverkehr nach Iteration 20

Ein Blick auf die Scores bestätigt die Vermutungen, die schon beim Betrachten der Visualisierung angestellt werden können (Abbildung 6). Anfangs liegt der Score noch bei ca. 110. Der Durchschnitt der schlechtesten Scores steigt dann auf etwa 118, der Durchschnitt der besten Scores steigt auf 127 an. Im Vergleich dazu sehen die Scores des Nullfalls anders aus. Hier liegt der initiale Score nach Iteration 0 schon bei 115 und steigt auch 121 (average worst score). Der Durchschnitt der besten Scores erreicht wie auch im Planfall 127 (Abbildung 7).

Beim Betrachten der Leg Histogramme in den einzelnen Iterationsläufen wird deutlich, dass die ausgeprägten Verkehrsspitzen in den beiden Hauptverkehrszeiten im Laufe der Durchläufe schwächer werden. Der Verkehr verteilt sich immer besser auf das Netz, sodass nach Iteration 20 in der Spitze maximal 670 Fahrzeuge gleichzeitig im Netz unterwegs sind (Iteration 0: 1390, Iteration 10: 700).

Die durchschnittliche Reisezeit verringert sich von anfangs 55 Minuten und 53 Sekunden (Iteration 0) über 35 Minuten und 8 Sekunden (Iteration 10) auf 34 Minuten und 50 Sekunden (Iteration 20). Damit liegt sie 15,4 % (4 Minuten und 39 Sekunden) über der eingangs ermittelten, kürzesten Reisezeit von 30 Minuten und 11 Sekunden.

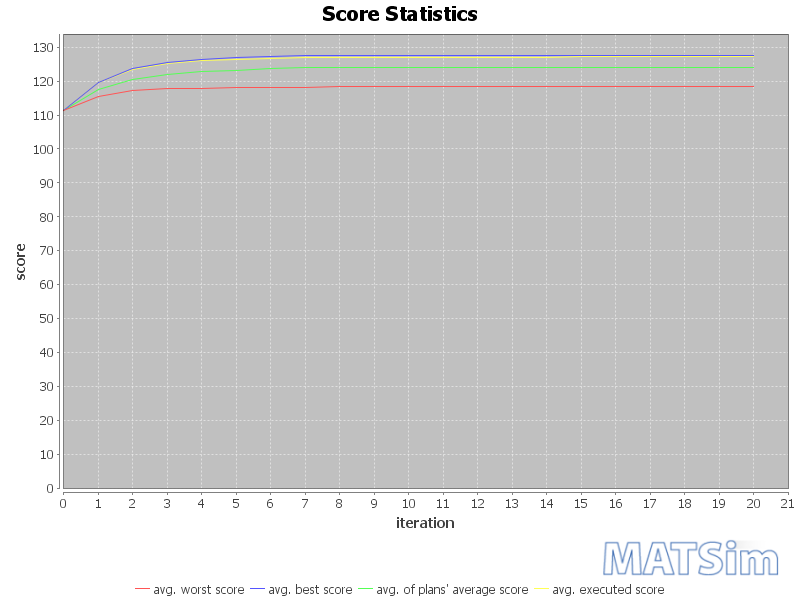


Abbildung : Score Statistics im Planfall

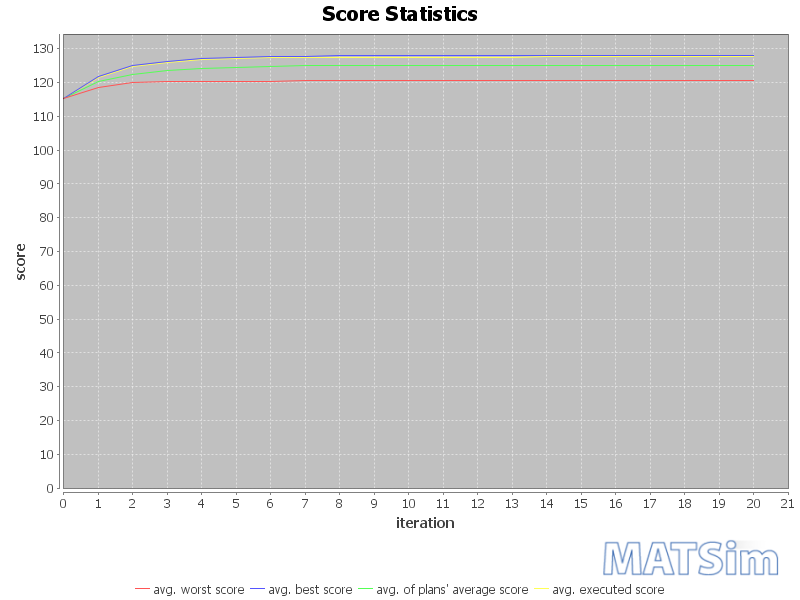


Abbildung : Score Statistics im Nullfall

# Aufgabe 5

Zur Bewertung der Auswirkungen von der in Kapitel 4 durchgeführten Maßnahme auf die Netzauslastung werden die Fahrzeuge auf allen Kanten des Netzes vor und nach der Maßnahme gezählt. Von Interesse ist dabei, wie sich die Nutzung einzelner Kanten durch die Maßnahme ändert. Die Differenz beider Werte wird in folgender Klasse gebildet:

**public** **class** NetworkUsageCounter **extends** BaseDifferenceHandler{

**private** IdCounter counter\_before = **new** IdCounter();

**private** IdCounter counter\_after = **new** IdCounter();

**private** Network netw;

**public** NetworkUsageCounter(EventsManager eventsManager, Network network, Case fall)

{

**super**(eventsManager, fall);

**this**.netw = network;

**super**.handler\_before = counter\_before;

**super**.handler\_after = counter\_after;

setHandler();

}

**public** Map<Id, Integer> calculateDifferences()

{

Map<Id, Integer> result = **new** HashMap<Id, Integer>();

**for**(Entry<Id, ? **extends** Link> entry : netw.getLinks().entrySet())

{

Integer nullfall = counter\_before.getCount(entry.getKey());

Integer planfall = counter\_after.getCount(entry.getKey());

**if**(nullfall == **null**)

nullfall = 0;

**if**(planfall == **null**)

planfall = 0;

**int** difference = planfall - nullfall;

result.put(entry.getKey(), difference);

}

**return** result;

}

}

Die Klasse *IdCounter* implementiert das Interface *LinkEnterEventHandler.* Für jede befahrene Kante wird so die Anzahl der *LinkEnterEvents*  assoziiert mit der *LinkId* gespeichert. In der Methode *calculateDifferences* der Klasse *NetworkUsageCounter* werden die Differenzen der Belastung aller Kanten, vor und nach der Maßnahme, assoziiert mit ihrer *LinkId*  ermittelt. Die Auslastungsdifferenzen werden anschließend im CVS Format gespeichert, sodass sie in QGIS visualisiert werden können. Die in Abbildung 8 dargestellten Auswirkungen der Maßnahme beschränken sich im Wesentlichen auf den Raum Cottbus. Es ist zu erkennen, dass die durch das Stadtzentrum führenden Hauptachsen entlastet werden. Allerdings wird der Stadt- bzw. Nordring wesentlich stärker befahren als vor der Maßnahme.

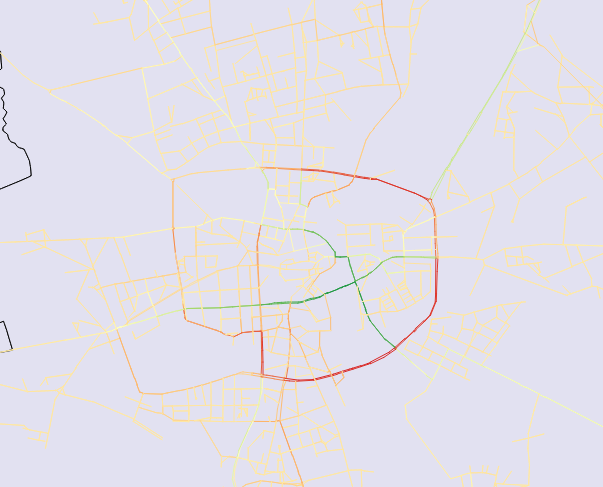


Abbildung : Differenz der Auslastung vor und nach der Maßnahme Rot= mehr Fzg. Grün = weniger Fzg.

# Aufgabe 6

Zum Bearbeiten der sechsten Aufgabe werden neue Input- und Output-Ordner sowie eine Kopie neue config.xml-Datei angelegt. In dieser wird die Verkehrsmittelwahl mithilfe des Moduls „ChengeLegMode“ freigegeben. Als Modi sollen „car“ und „pt“ (public transport) zur Verfügung stehen. Der Wert für „ignoreCarAvailability“ wird auf „false“ belassen. Die Zeitwahl wird ebenfalls freigegeben. Im Modul „planCalcScore“ wird dafür die „Öffnungszeit“ der Arbeitsstätten auf 6:00 Uhr gesetzt, die späteste Startzeit ist 10:00 Uhr. Im selben Modul wird außerdem der „Malus“ sowohl für „traveling“ als auch für „travelingPt“ auf -6 gesetzt.

Zu den Strategiemodulen wird das Modul ChangeLegMode hinzugefügt und mit einem Wahrscheinlichkeitswert von 0.5 versehen. Alle Strategiemodule werden nach Iteration 16 deaktiviert. Es werden 20 Iterationen durchgeführt.

Zum Ermitteln des Modal Split wird zunächst die Datei 0.legHistogram.txt mit OpenOffice Calc eingelesen. Die Summe der Spalte „departures-car“ wird mit der Summe der Spalte „departures-pt“ zu Gesamtverkehrsmenge addiert. Es ergibt sich dabei ein car/pt-Modal Split von 53% zu 47%. Dies war zu erwarten, da beim Erstellen der Nachfrage genau dieser Modal Split angegeben wurde. Bis zum Abschalten der Strategiemodule ändern sich diese Werte nur geringfügig, dann steigt der Anteil von „car“ auf 56% (Iteration 16) und erreicht 66% (Iteration 20). Nach zwei weiteren Iterationsdurchläufen zeigt sich, dass mit einem Wert von -4 für „travelingPt“ ein akzeptabler Modal Split von 54,9% (car) zu 45,1% (pt) nach Iteration 20 erreicht wird.

Mit diesen Einstellungen wird eine weitere Iteration für den Planfall durchgeführt. Nach Iteration 0 stellt sich dabei wieder der initiale Modal Split (53%/47%) ein. Nach Iteration 16 beträgt der Anteil von „car“ 51% und steigt bis nach Iteration 20 wieder an auf 53,6%.

# Aufgabe 7

Im Folgenden werden weitere Auswirkungen der in Kapitel 4 durchgeführten Maßnahme untersucht.

Zu diesem Zweck wurden die Klasse, *TripDurationDifference, TripDurationAnalyser* und *BridgeUserAnalyser* implementiert. Der Quellcode aller drei Klassen befindet sich im Anhang. Die Klasse *BridgeUserAnalyser* speichert die *PersonIds*  der Personen, die die Brücke vor Einführung der Maßnahmen benutzen. *TripDurationAnalyser* erfasst die Reisezeiten pro *Trip* assoziiert mit der *PersonId.* Die Klasse *TripDurationDifference* bietet Funktionalität zur Ermittlung von Reisezeitdifferenzen. Die Zeitdifferenzen sind immer Bezogen auf die durchschnittliche TripDuration einer Person. Es ist möglich die Differenz der durschnittlichen Reisezeiten für alle Personen oder einen bestimmten Prozentsatz zu ermitteln. Auch ist es möglich Reisezeitdifferenzen für einen bestimmten Personenkreis zu ermitteln.

Mit Hilfe der implementierten Hilfsmittel wurde zunächst ermittelt wie sich die Reisezeit für Nutzer der betroffenen Brücken verändert. Es wurden zu diesem Zweck zunächst mittels *BridgeUserAnalyser* alle Brückennutzer ermittelt. Anschließend wurde mit der Methode *calculateTripDurationDiff* die Reisezeitdifferenz dieser Personen ermittelt. Interessant ist hierbei, dass sich die durchschnittliche Reisezeit der betroffenen Personen um lediglich 75 Sekunden verlängert.

In einem weiteren Schritt wurde untersucht an welchen Orten diejenigen Personen wohnen und arbeiten, die eine Verlängerung ihrer durchschnittlichen Reisezeit erfahren. Betrachtet wurden 10% der am stärksten betroffenen Personen. Die Ermittlung Wohn und Arbeitsstätten dieser Personengruppe wurden mit den Methoden *getHomeOfSlowest* und *getWorkOfSlowest* ermittelt.

Die so ermittelten Daten wurden in QGIS visualisiert. In Abbildung 9 ist zu erkennen, dass ein Großteil der betroffenen innerhalb Cottbus wohnt und arbeitet. Allerdings sind auch auswärtige Pendler von der Sperrung der Brücken betroffen.

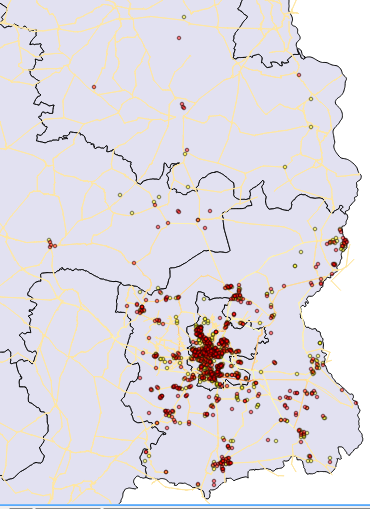


Abbildung : Wohn- und Arbeitsstätten von 10% der Pendler mit höchster Reisezeitsteigerung gelb=Arbeitsort rot = Wohnort

# Anhang

## TripDurationDifference

**public** **class** TripDurationDiference **extends** BaseDifferenceHandler{

**private** TripDurationAnalyser durationAnalyser\_before = **new** TripDurationAnalyser();

**private** TripDurationAnalyser durationAnalyser\_after = **new** TripDurationAnalyser();

**public** TripDurationDiference(EventsManager eventManager, Case fall)

{

**super**(eventManager, fall);

**super**.handler\_before = durationAnalyser\_before;

**super**.handler\_after = durationAnalyser\_after;

setHandler();

}

**public** Map<Id, Double> calculateTripDurationDifference()

{

Map<Id, Double> result = **new** HashMap<Id, Double>();

**for**(Entry<Id, TripPerson> entry :durationAnalyser\_before.getPersons().entrySet())

{

TripPerson planfallPerson = durationAnalyser\_after.getPerson(entry.getKey());

**double** planfallAvrg = planfallPerson.averageTripDuration();

**double** nullfallAvrg = entry.getValue().averageTripDuration();

**double** difference = planfallAvrg - nullfallAvrg;

result.put(entry.getKey(), difference);

}

**return** result;

}

**public** List<Entry<Id, Double>> calculateOrderedTripDurationDifference()

{

Map<Id, Double> map = calculateTripDurationDifference();

List<Entry<Id, Double>> result = **new** ArrayList<>();

**for**(Entry<Id, Double> entry : map.entrySet())

{

result.add(entry);

}

Collections.*sort*(result, getComparator());

**return** result;

}

**public** List<Entry<Id, Double>> calculateOrderedTripDurationDifferenceSlowest(**double** offset)

{

List<Entry<Id, Double>> durations = calculateOrderedTripDurationDifference();

List<Entry<Id, Double>> result = **new** ArrayList<Entry<Id, Double>>();

**int** tenPerCent = (**int**) (durations.size() \* offset);

**for**(**int** i = durations.size() - 1; i > durations.size() - tenPerCent; i--)

{

result.add(durations.get(i));

}

**return** result;

}

**public** Map<Id, Coord> getHomeOfSlowest(**double** offset, Scenario sc)

{

List<Entry<Id, Double>> slowest = calculateOrderedTripDurationDifferenceSlowest(offset);

Map<Id, Coord> result = **new** HashMap<Id, Coord>();

**for**(Entry<Id, Double> entry : slowest)

{

Person person = sc.getPopulation().getPersons().get(entry.getKey());

Activity actHome = (Activity) person.getSelectedPlan().getPlanElements().get(0); //first Activity home

result.put(entry.getKey(), actHome.getCoord());

}

**return** result;

}

**public** Map<Id, Coord> getWorkOfSlowest(**double** offset, Scenario sc)

{

List<Entry<Id, Double>> slowest = calculateOrderedTripDurationDifferenceSlowest(offset);

Map<Id, Coord> result = **new** HashMap<Id, Coord>();

**for**(Entry<Id, Double> entry : slowest)

{

Person person = sc.getPopulation().getPersons().get(entry.getKey());

Activity actWork = (Activity) person.getSelectedPlan().getPlanElements().get(2); //1 should be leg home - work, thus 2 is work

result.put(entry.getKey(), actWork.getCoord());

}

**return** result;

}

**public** **double** calculateTripDurationDiff(Set<Id> personIds)

{

**double** totalDiff = 0;

**for**(Id id : personIds)

{

**double** diffBefore = durationAnalyser\_before.getPerson(id).averageTripDuration();

**double** diffAfter = durationAnalyser\_after.getPerson(id).averageTripDuration();

totalDiff += diffAfter - diffBefore;

}

**return** totalDiff / personIds.size();

}

**private** Comparator<Entry<Id, Double>> getComparator()

{

**return** **new** Comparator<Entry<Id, Double>>(){

**public** **int** compare(Entry<Id, Double> o1, Entry<Id, Double> o2) {

**if**(o1.getValue() > o2.getValue())

**return** 1;

**else** **if**(o1.getValue() < o2.getValue())

**return** -1;

**return** 0;

}

};

}

}

## TripDurationAnalyser

**public** **class** TripDurationAnalyser **implements** AgentDepartureEventHandler, AgentArrivalEventHandler{

**private** Map<Id, TripPerson> persons = **new** HashMap<Id, TripPerson>();

**public** Map<Id, TripPerson> getPersons()

{

**return** persons;

}

**public** TripPerson getPerson(Id personId)

{

**return** persons.get(personId);

}

**public** **void** reset(**int** arg0) {

// **TODO** Auto-generated method stub

}

**public** **void** handleEvent(AgentArrivalEvent arg0) {

Id personId = arg0.getPersonId();

TripPerson person = persons.get(personId);

person.finishLastTripo(arg0.getTime());

persons.put(personId, person);

}

**public** **void** handleEvent(AgentDepartureEvent arg0) {

Id personId = arg0.getPersonId();

TripPerson person = persons.get(personId);

**if**(person == **null**)

person = **new** TripPerson(personId);

person.startTrip(arg0.getTime());

persons.put(personId, person);

}

}

## BridgeUserAnalyser

**public** **class** BridgeUserAnalyser **implements** LinkEnterEventHandler{

**private** Set<Id> bridgeUser = **new** HashSet<Id>();

**private** Set<Id> bridgeLinks = **new** HashSet<Id>();

**public** BridgeUserAnalyser(Set<Id> bridgeLinks)

{

**this**.bridgeLinks = bridgeLinks;

}

**public** Set<Id> getBridgeUsers()

{

**return** bridgeUser;

}

@Override

**public** **void** reset(**int** arg0) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** handleEvent(LinkEnterEvent arg0) {

**if**(bridgeLinks.contains(arg0.getLinkId()))

bridgeUser.add(arg0.getPersonId());

}

}