Analysis and simulation of game theoretical routing models

Oberdörfer Tobias, Oswald Lorena, Pobitzer Markus, Suter Antoine

May 2020

Abstract

We analyze the effects of different theoretical routing models on graphs including selfish routing, centralized routing, and a middle way, such as taxed selfish routing. Especially, we will look at Braess's Paradox in the context of computer networks and the price of anarchy caused by it. Through this analysis we, as well as the reader, can get a better understanding of how the different models work and compare them with each other.

Additionally, we created a simulation to get a better understanding how the models can be used for a graph in a network and to visualize the different models. We will describe the difficulties of implementing the various models and compare their run times to get a better understanding how feasible they are.

Contents

1 Introduction

There are many ways to handle routing in theory and in real life. If we are looking at computer network routing protocols then actual decisions over routing are made by the nodes in the graph using various methods to find a distribution of traffic close to optimal. This is called decentralized routing. However, there are different approaches which are worth discussing. One such approach comes from Game Theory, particularly non-cooperative one. In this paper, it is our goal to explore two of these models [RT02; CDR06] and compare them to the optimal routing solution such that a reader, who only knows basics about game theory and graphs, can understand it. We will do this by summarizing the theory in the first part and then creating a simulation of the chosen models. The simulation will be explained in the paper and additionally will be made available through functioning code and visualizations. To get the optimal solution on a complete graph a central entity will compute and assign each agent their path. In the end we compare the models to each other based on theory, simulation and runtime analysis. We will further discuss the challenges we had implementing the models. Finally, we will talk about possible improvements and extensions to our models and simulations.

2 Analysis

2.1 Overview over mathematical model

First, we describe the basic mathematical definitions that are used throughout the paper and also in our simulation engine. It is based on the mathematical model of [RT02].
We consider a directed graph $G(V, E)$ where the set V represents the vertices and set E represents the edges. The graph has a start node s and a destination node t. We denote the set of simple, non-negative s-t-paths P as $\mathcal{P}$. The flow over a path is a function $f_P : P \to \mathbb{R}^+$; f is the sum of all flows in P. Similarly, f_e defines the flow over one edge and both definitions are connected by the equation $\sum_{e \in P} f_P$. Overall the graph has a traffic rate r and a flow f is feasible if $\sum_{P \in \mathcal{P}} f_P = r$.
In our application we consider flow that is not continuous, i.e. we have a number of agents which are either traversing an edge or not. This simplifies the simulation.
For every edge we define a latency function l_e which describes how much latency is incurred when a flow f_e is on an edge e. It is defined as $l_e : f_e \in \mathbb{R} \to \mathbb{R}$. Correspondingly to the definition of flow, we define the latency over a path by $l_P(f_P) = \sum_{e \in P} l_e(f_e)$. If only the latency of the network graph is considered, the quality of flow can be viewed as the total latency

$$L(f) \equiv \sum_{P \in \mathcal{P}} l_P(f) \cdot f_P = \sum_{e \in E} l_e(f_e) \cdot f_e \tag{1}$$

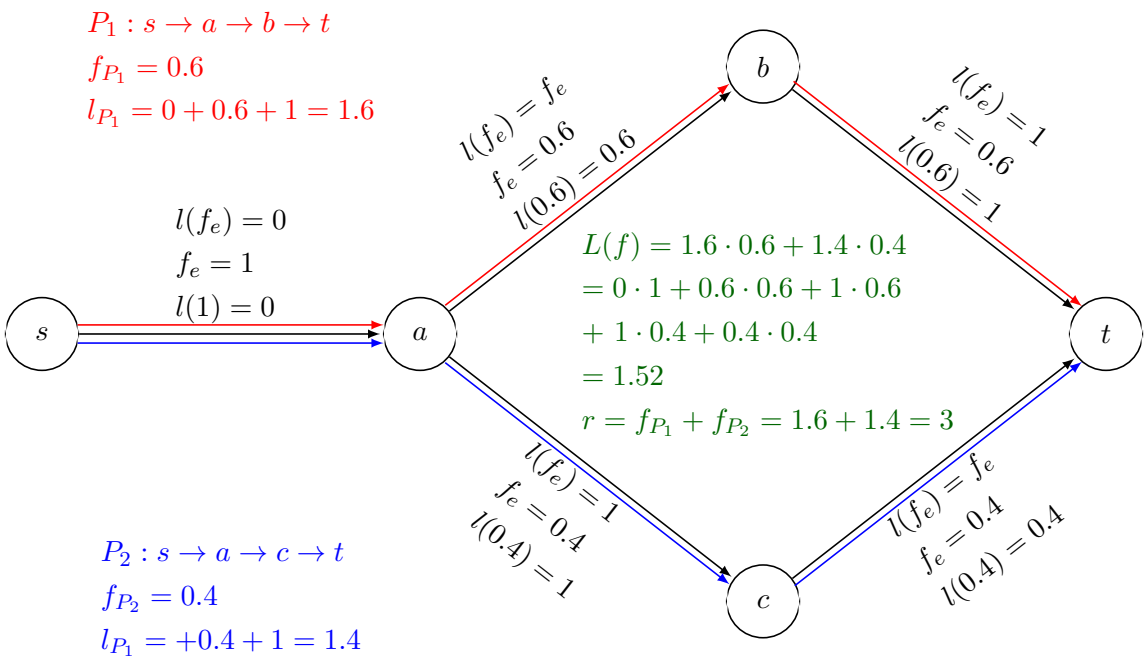


Figure 1: An example of an instance (G, r, l)

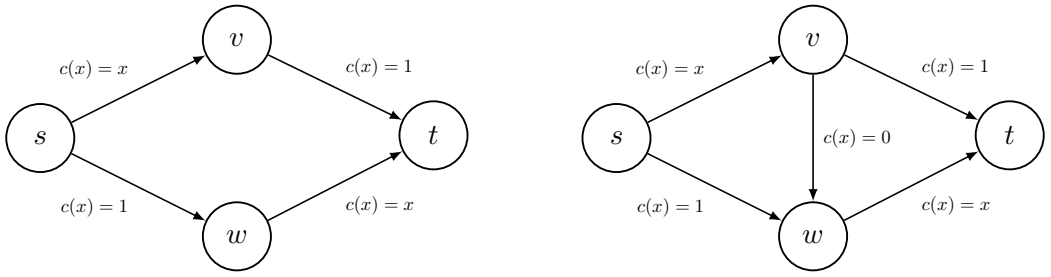
The optimal flow, that minimizes the total latency, always exists.
In some applications the input x is used for the latency function. $x \in [0, 1]$ describes the proportional load or flow of an edge. So if $x = 1$ for an edge then all flow goes through this edge.
In some of the models viewed, there exists tax on edges additionally to latency. Tax on an edge τ_e is a non-negative value. Tax on a path is defined as $\tau_p = \sum_{e \in P} \tau_e$. Cost of flow is then defined as

$$C(f, \tau) \equiv \sum_{P \in \mathcal{P}} [l_P(f) + \tau_P] \cdot f_P = \sum_{e \in E} [l_e(f_e) + \tau_e] \cdot f_e \tag{2}$$

It is obvious that if $\tau = 0$ then $L(\cdot) = C(\cdot, \tau)$.
The model is therefore defined by a triple (G, r, l) called an instance. Or respectively a triple $(G, r, l + \tau)$ called an instance with taxes. A complete example of an instance can be viewed in figure 1.

2.2 Selfish routing

Selfish routing is a strategy in network routing. Every participant wants to minimize its own cost. Costs can be in form of latency, taxes, traveling time or even a combination. This is not a centralized system, therefore every participant knows only a part of the whole network and the other participants. If we have an equilibrium in this model, we speak of a



(a) Initial network (b) Augmented Network

Figure 2: Braess's paradox

Nash equilibrium. As we will see, the Nash equilibrium is not always the optimal solution. The ratio between the worst Nash equilibrium and the optimal solution is the price of anarchy.
We introduce this model with the example of Braess's Paradox, first discovered by Braess [Bra68].

If we look at the initial graph (a) we have a graph $G(V, E)$ where V represents all nodes $\{s, v, w, t\}$, E represents all edges $\{(s, v), (s, w), (v, t), (w, t)\}$. An instance of this graph is the triple (G, r, c), with a traffic rate r and latency function $c(.)$ on every edge. The total traffic in the network is 1. The functions $c(x) = x$ represent congestion, where x represents the current traffic over this edge and the function $c(x) = 1$ are independent of congestion therefore the cost is not coupled with the amount of traffic. The Nash equilibrium of the initial network is when half the traffic takes the upper path and the other half the lower path. Then we have a cost of $\frac{3}{2}$ for every involved agent, at the same time this is the optimal cost and therefore the price of anarchy is one.
Now consider (b) the augmented network that has an additional edge (v, w) with latency cost of 0, therefore independent of congestion. If we compare the optimal flow of both networks we see that nothing has changed and the cost is still at $\frac{3}{2}$ for every participant. However, the Nash equilibrium of the selfish model differs. The new Nash equilibrium goes over the path $s \to v \to w \to t$ with a cost of 2 for every agent. At first glance one may thinks that a free edge does not increase the latency, maybe even decrease it. But this example shows that there is sometimes an opposite effect. The price of anarchy is therefore $\frac{4}{3}$.
If we consider only linear latency functions (this means that every latency function $c(x)$ has the form $c(x) = ax + b$ for some $a, b \geq 0$) on the edges, Roughgarden [RT02] showed that $\frac{4}{3}$ is the worst-case ratio (i.e. the highest value for the price of anarchy). This is a very interesting result, since many congestion protocols work with linear latency functions such

as TCP showen by Friedman [Fri02]. At the same time we see, that Braess's paradox as showen above is a worst-case scenario.
With linear latency functions an upper bound for the ratio can be found, unfortunately this does not hold for non-linear latency functions as we will see. Let us consider a variant of Pigou's example.

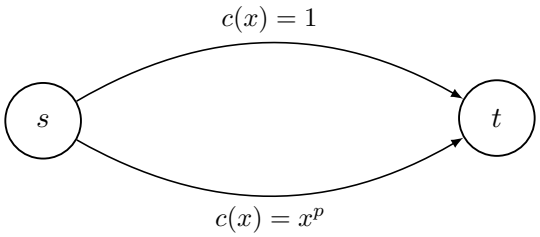


Figure 3: Pigous example

Again we have a Network $G(V, E)$ with instance (G, r, c). But this time the latency cost function of one edge is not linear. Consider the lower edge with the cost function $c(x) = x^p$ where $p > 0$ is large. The optimal flow routes a small ϵ fraction of the traffic on the upper edge and the rest on the on the lower edge. Therefore it has a cost of $\epsilon + (1-\epsilon)^{p+1}$. Where $\epsilon = 1 - (p+1)^{\frac{-1}{p}}$, see [Par11] for reasoning. As p goes to infinity the cost of the optimal flow goes to 0 and we show that the price of anarchy grows without bound.
The optimal flow:

$$\begin{aligned}\epsilon + (1-\epsilon)^{p+1} &= 1 - (p+1)^{\frac{-1}{p}} + (1 - (1 - (p+1)^{\frac{-1}{p}}))^{p+1} \\ &= 1 - p(1+p)^{\frac{-(1+p)}{p}}\end{aligned}$$

The Nash equilibrium, is the flow chosen by the selfish agents. All agents take the lower edge. Therefore the equilibrium lies at 1^p.
And the price of anarchy is the ratio between the Nash equilibrium and the optimal flow:

$$\begin{aligned}&\frac{1^p}{1 - p(1+p)^{\frac{-(1+p)}{p}}} \\ &= 1 - p(1+p)^{\frac{-(1+p)}{p}} \\ &= \frac{1}{1 - p(1+p)^{\frac{-(1+p)}{p}}}\end{aligned}$$

If we let p go to infinity we get an unbound price of anarchy. One can show that $\mathcal{O}(\frac{p}{\ln p})$ is an upper bound [Rou02].

2.3 Taxed selfish routing

Just like selfish agents, taxed selfish agents determine their path of travel to be the best possible for themselves and disregard the overall cost/latency of travel of the whole group. The difference between them and simply selfish agents, is that they have an incentive to avoid a path that increase the cost for the other agents, using this path or segments of it. In other words: they have to pay taxes. Just like in real life one pays more for what is faster or more comfortable like using a car instead of taking the more time-consuming public transportation.
Taxes can be very powerful, sometimes even more powerful than removing edges altogether [CDR06]. But lowering the latency for all the agents on a network for the price of (very) high taxes is counterproductive to the goal of reducing the cost, which can be time and taxes of each agent.
Thus, one has to place strategic taxes, as to not make the cost of each agent too high.

Another way to solve the problem is with the principle of marginal cost pricing. It assumes that taxes cause no dis-utility to network users [CDR06]. These are taxes τ, which correspond to the additional cost they create by taking an edge with a non-constant latency function.

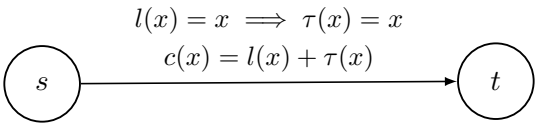


Figure 4: Example: Taxed

For the example in figure 4 there is 1 unit of agents in total. All (1/1) agents will follow the same edge and thus making it 1 unit more expensive because the latency on the edge was 0 before and now it is 1. Thus all agents would pay: 1 unit latency + 1 unit tax = 2 units of cost.

Let's look at the graph Figure 2b. We will first show how to calculate the latency and cost of each agent by applying the principle of marginal cost.
We start by transforming the graph in a mixed strategies problem, where each unique path is a strategy an agent can use.

path	probability	cost
s-u-v-t	p	$[2p + 2q] + [0] + [2p + 2(1 - p - q)]$
s-u-t	q	$[2p + 2q] + [1]$
s-v-t	$1 - p - q$	$[1] + [2(1 - p - q) + 2p]$

Table 1: Cost of paths using marginal cost

[ ] encases each edge a path follows. The reason why we use 2 variables instead of 3 for the probabilities in Table 1 will be looked at further in the code section of this agent in Section 4.2.
Assuming that agent 2 plays a mixed strategy, which makes agent 1 indifferent, let the probabilities p, q and $1-p-q$ be the chance that the agent 1 takes either of the 3 paths. In such a condition no matter what agent 2 opts for, any strategy by agent 1 would be termed as a best response. This is because payoffs are equal for all strategies for agent 1.
To get the probabilities agent 1 chooses the different paths we start by setting the cost of pairwise strategies (paths) equal.

$$\begin{aligned} cost(s-u-t) &= cost(s-u-v-t) \\ 2p + 2q + 1 &= 2p + 2q + 2p + 2(1-p-q) \\ 2q &= 1 \\ q &= \frac{1}{2} \end{aligned}$$

$$\begin{aligned} cost(s-u-t) &= cost(s-v-t) \\ 2p + 2q + 1 &= 1 + 2(1-p-q) + 2p \\ 2p + 4 \cdot \frac{1}{2} &= 2 \\ p &= 0 \end{aligned}$$

As we can see from these results, agent 1 will never take the path s-u-v-t, half of the time path s-u-t and the other half time path s-v-t. Since all the agents decide the same, they all have a latency of $\frac{3}{2}$ and tax of $\frac{1}{2}$ making it a cost of 2 overall.

We will use the same technique for arbitrary taxes, where we put a tax of at least $\frac{1}{2}$ on the edge u-v [CDR06].
This results in the following cost table.

path	probability	cost
s-u-v-t	p	$[p+q]+[\frac{1}{2}]+[p+(1-p-q)]$
s-u-t	q	$[p+q]+[1]$
s-v-t	$1-p-q$	$[1]+[(1-p-q)+p]$

Table 2: Cost of paths using arbitrary tax

If we solve for the unknown parameters p and q we get the same results as before.

$$\begin{aligned} cost(s-u-t) &= cost(s-u-v-t) \\ p+q+1 &= p+q+\frac{1}{2}+p+(1-p-q) \\ q &= \frac{1}{2} \end{aligned}$$

$$\begin{aligned} cost(s-u-t) &= cost(s-v-t) \\ p+q+1 &= 1+(1-p-q)+p \\ p+2\cdot\frac{1}{2} &= 1 \\ p &= 0 \end{aligned}$$

As we can see from these results, agent 1 will again never take the path s-u-v-t, half of the time path s-u-t and the other half time path s-v-t. Since all the agents decide the same, they all have a latency of $\frac{3}{2}$ and no tax making it a cost of $\frac{3}{2}$ for all.

In this example we see that both types of taxes reduce the latency from 2 as seen for the selfish agents in Section 2.2 to $\frac{3}{2}$ but the cost with the principle of marginal cost was still 2.

2.4 Centralized routing

Centralized routing means that a central entity determines the path each agent takes. It is a way to find optimal routing in a complete graph with distinct units of flow that in practice is not usually considered. It is hard to find references or papers talking about centralized routing. In discussion within our group we only found a brute force approach to solve this problem. In a practical setting it does not make sense to use a centralized approach since there are too many nodes and varying flow to consider. In reality most computer networks use decentralized dynamic routing systems [Wik20] where each node can decide where to route a packet. However, in order to get an understanding of what theoretically is the optimal way to distribute a static flow in a network to minimize latency a small example calculated this way is quite useful.

There are three parts to find an optimal routing. First, we need to find all unique simple paths from s to t. Second, we need to find all possible distributions of flow, in this case agents, over the unique paths. The number of distributions is $O(agents^{paths})$. Since the number of paths is dependant on the graph this part is non-polynomial. Third, using equation (1) calculate the total latency of each distribution and find the minimal one. Finding unique paths in a graph is NP-hard [htt20]. Since part 2 and 3 take exponential time, this will only work for smaller graphs efficiently. The algorithm and implementation is specified in section 4.3.

Before we came to this solution there were other approaches discussed. All of them had a least one flaw that made them unsuitable or gave an incorrect result. For example, at first we considered using the Bellman-Ford algorithm to determine the optimal path. Just reading this, one can see that this a completely incorrect approach since we need to find the optimal latency over the whole graph not just an optimal path. This would only work if all agents are able to use the optimal path which is not a given.

3 Simulation Engine/Visualizer

3.1 General overview

This section is supposed to give a reader the general idea of what we thought of and what our decisions were during the implementation of the simulation engine, as well as an overview for the reader to be able to expand and enhance this project.
All of the code described in this paper, as well as all of the already created simulation configurations (to be run on the SimEngine) are open-sourced under GNU GENERAL PUBLIC LICENSE V3 at routingSimEngine [T+] and hence can be cloned and followed by the interested reader.

Simulation idea Our group had at the beginning two general ideas for the simulation of routing networks. The first idea was, that each agent in a network has a specified speed and then each step of the simulation each agent is allowed to run the distance it could given it's speed. But this approach brings a lot of difficulties with it, such as not all agents arriving at the same time at crossroads, not knowing front up how long the simulation will take, the internal representation of the network has to have some sort of distance associated with it, cost of traffic on an edge is somewhat mis-represented, because agents do not travel besides each other, but rather one after another and quite a few more problems.
The other idea, with which we then went with, was that each agent knows the whole network topology, as well as the decisions of the agents before them, while deciding his final route through the network. Then each step of the simulation engine is just the decision process of the next agent in line, while always updating the cost for each edge given the decided paths of all previous agents. This also played nicely into the visualisation of this simulation, since now we were able to show through an animation, how the network gets more congested as more agents "drive" (i.e. decide) on their route through the network.

Programming language of choice The decision to use Java for the simulation engine was made, because everyone in our group knew it already well and we decided that we want to use Java's interfaces as our contract between the engine and the implementation of the different models for the agents. The decision to also use Java for the Simulation Visualizer was made later and more or less purely on the fact, that we already wrote a parser for the output of the simEngine, as well as finding the Java GraphStream project [Tea].

Project structure If you as an interested reader want to follow along during the introduction to our simulation engine and visualizer, as well as maybe even try out your own networks or even agents, here is a small overview of our project structure (Fig. 5).

First we have a folder *libs* with all our used libraries, namely the gs-* are from graphStream [Tea], then we have org.json [tea], which contains the json parser library and in the folder *z3-4.8.8-x64-ubuntu-16.04* is the constraint solver z3 [Z3], used to create taxed selfish routing agent.

In the folder *networks* we have two subfolders containing the simulation configurations (i.e. input for simEngine) and the generated ouput.

Under *src* you find multiple subfolders in total containing all of the source for this project. Namely in *agents* our different implementations of the agent models, in *simEngine* the most part of the simulation engine, in *utils* different utility functions used by both the engine and the visualizer and finally in *visualizer* the simulation visualizer.

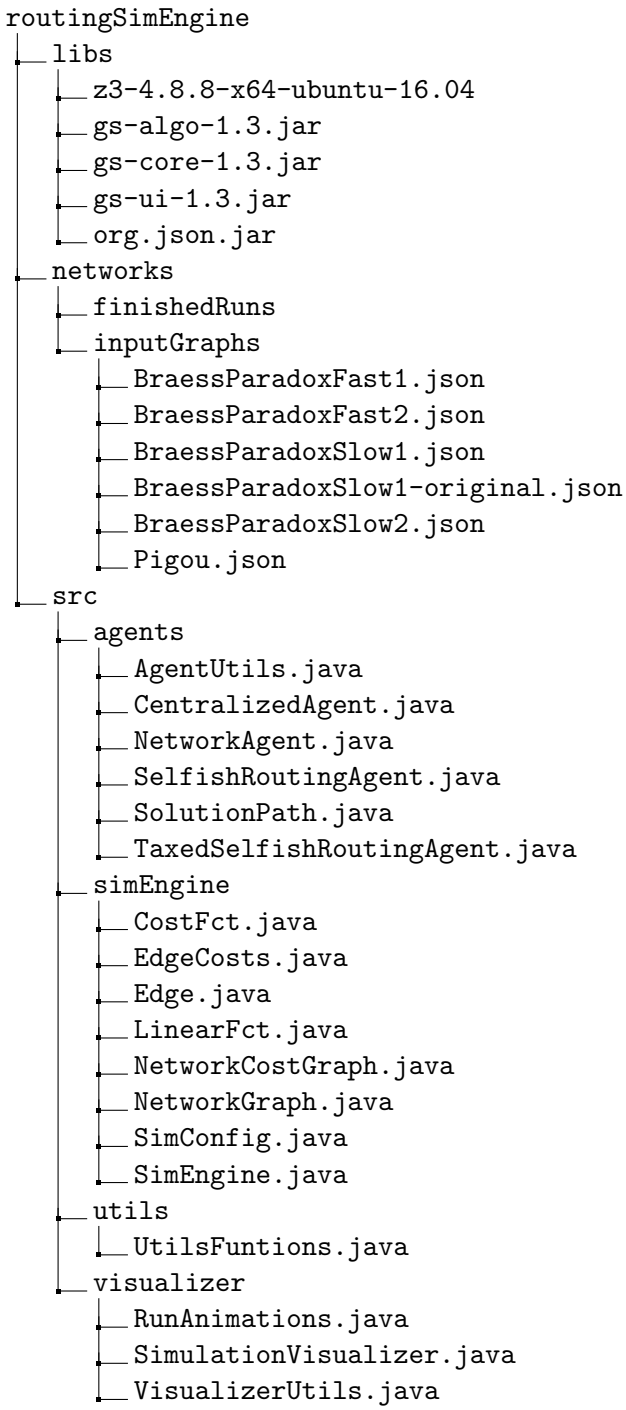


Figure 5: routingSimEngine github project structure

3.2 Details on the input/output of our framework

For the input and output of the simEngine we decided to use the json format, since we were already quite used to it and it felt to us as being an easy to use text based configuration format. We found the library org.json [tea] to be flexible enough as our json parser and hence just wrote a wrapper around it.

Simulation configuration (input) As we wanted to be able to run this simulation on multiple different networks and also make it easy for readers to create their own simulations using this simulation engine, all the configurations of a simulation, except which agents are run on it, are done in the input file.
The structure of our configuration file is really simple and just contains two main ideas over five keys, which are the network with *edges* and *nodes* and the simulation with *networkTitle*, *amountsOfAgents* and *agentsPerStep*.
The network configuration is straight forward, the *nodes* key contains an array of strings, where each string represents a node and at the same time its name. During simulation the lexicographic ordered first node is the node, from which all agents start i.e. the source, and the lexicographic ordered last is the destination of all agents i.e. the sink. The *edges* on the other hand is an array of objects each representing an edge, which again contain two keys *connection* and *cost*. The *connection* key is an array of two strings which are the two nodes, where the first string is the starting node and the second string the end node of the directed edge. The key *cost* is a string representing the cost function of this edge in the form $a * t + b$ because we only implemented linear cost function up to this point.
The simulation configuration is a bit more involved, since the *amountsOfAgents* and *agentsPerStep* are values used during the run of the simulation. Namly *amountsOfAgents* is, as the name suggests, the amount of agents the simulation engine will use, i.e. how many steps it will run. It has hence a direct influence on the runtime of the simulation. *agentsPerStep* is the amount of agents the simulation engine will accumulate into one step for the output format and therefore determines the timing of the visualizer. Lastly the *networkTitle* is self explanatory, as it is just the title used to describe this network in the console, as well as for the visualization later.

Listing 1: Simulation config for Pigou's example

```
{"networkTitle": "Pigou's example",
  "amountOfAgents": 1000,
  "agentsPerStep": 500,
  "nodes": [ "a", "b", "c", "d" ],
  "edges": [{
      "connection": [ "a", "b" ],
```

```
      "cost": "0t+1"
    }, {
      "connection": [ "c", "d" ],
      "cost": "0t+0"
    }, {
      "connection": [ "a", "c" ],
      "cost": "0.001t+0"
    }, {
      "connection": [ "b", "d" ],
      "cost": "0t+0"
    }]
}
```

The json representation (Listing 1) above is an example of such a json file, which in this case is our simulation configuration for Pigou's example.

Simulation result (output) At the beginning our output of the simulation engine was almost exactly the same as its input but during the implementation of the visualizer we changed it more and more. Our current output is structured in such a way, that multiple network simulations can be in just one json file, i.e. multiple input configurations are run and aggregated into one output file.
At the first level we have just the filenames of the input configurations (without .json) as the key for the respective simulation run. Each simulation rung has the following structure: each network has the same key and respective values for *networkTitle*, *amountsOfAgents* as well as *nodes* as the corresponding simulation configuration. The key *edges* also still corresponds to an array of objects representing edges, but these objects have an additional value. The two keys and their values *connection* and *cost* are still the same, but the key *usage* and its object is new. This object is a list of strings, which are the names of the agent models associated with an array containing the amount of agents on the current edge using this agent model's decisions for each step. For example the output `"TaxedSelfishRoutingAgent": [ 0, 0, 500 ],` of Pigou's example (Listing 2) means the edge from a to b, under the TaxedSelfishRoutingAgent model for all agents, does not have any agents on it at the point when the first 500 agents have decided their path. However, once all 1000 agents have decided on their path, 500 agents are using it.

Listing 2: Simulation output for Pigou's example

```
{ "Pigou": {
    "networkTitle": "Pigou's example",
    "amountOfAgents": 1000,
```

```
4         "nodes": [ "a", "b", "c", "d" ],
5         "edges": [ {
6             "cost": "0.0*t+1.0",
7             "usage": {
8               "SelfishRoutingAgent": [ 0, 0, 1 ],
9               "TaxedSelfishRoutingAgent": [ 0, 0, 500 ],
10              "CentralizedAgent": [ 0, 500, 500 ]
11            },
12            "connection": [ "a", "b" ]
13          }, {
14            "cost": "0.001*t+0.0",
15            "usage": { ... },
16            "connection": [ "a", "c" ]
17          }, {
18            "cost": "0.0*t+0.0",
19            "usage": { ... },
20            "connection": [ "b", "d" ]
21          }, {
22            "cost": "0.0*t+0.0",
23            "usage": { ... },
24            "connection": [ "c", "d" ]
25          }]
26      }
27    }
```

The json representation (Listing 2) above is part of one output json file, namely everything for Pigou's example except for three usage objects for brevity.

3.3 Simulation Engine

In this subsection we give you a small overview of what the most important classes of the *simEngine* do.

The *CostFct* interface, as well as the *LinearFct* class are just there to describe, as the name might suggest, the cost function of an edge in the network graph as a class, such that we can call different methods on it such as a nice *toString* method.

The *Edge* class is an abstracted implementation of an edge in the network graph, such that we can later use this edge object in java maps more easily.

SimConfig is basically a class corresponding to the simulation configuration json, just that this object already has everything parsed to Java objects and as such is easier to use.

The *EdgeCosts* class was one of the first implemented and served as part of the contract between the *simEngine* programmers and the agent programmers, hence this is used by both to figure out, how much an edge costs at this moment in the simulation.
The abstract class *NetworkGraph* and its first implementation the *NetworkCostGraph* are as the names imply, just the network graph itself and the network with the costs already calculated respectively.
Finally we have the *SimEngine* class, which now, under the usage of utility functions from the subfolder *utils*, brings all of the above mentioned classes and the agents under *agents* together to provide the full simulation engine we envisioned.

Interface of an agent As an interested reader may even want to try and implement a new type of agent model, we describe here in more detail the *NetworkAgent* interface, which every agent has to implement in order to be used in the simulation engine.
The interface itself is already annotated quite extensively, but the most important function to focus on is called *agentDecide*. This function gets called once for each agent, i.e. in each simulation step once, and receives as arguments the *NetworkCostGraph*, the *EdgeCosts* for manual edge cost calculation if necessary, the amount of agents already finished with their decision and finally the total amount of agents in this network simulation. This function has to return a list of the path this agent wants to take, including start and destination, identified by the internal node representation, i.e. *Integer* values.
The *getName* method of this interface has to return, just as the method name suggests, the name of this model. This name will then be displayed during the visualization as well as being the name in the output usage object. The classes *AgentUtils* and *SolutionPath* under src/agents are utility functions.

3.4 Simulation Visualizer using GraphStream

The simulation visualizer was partly an after thought, because only looking at the simulation results in text form was not easy to grasp and we want this whole project to be easily accessible to a lot of people.
In general, all this simulation visualizer does, is parse the simulation result json, create the network graphs in *graphStream* and then visually run through the steps the *simEngine* calculated as an animation on loop for all the different types of agents. This results in a more easily understandable and more visually pleasing argument why selfish routing is so bad and why the price of anarchy does exist, even in such simple network topologies. Figure 6 shows the visualization for the Braess's paradox at the beginning, when no agent model is yet animated. In Figure 7 you can see still images of the simulation visualization running, where the amount of agents on each edge at the end of the simulation for this agent implementation is shown. The colors of the edges are an interpolation from green

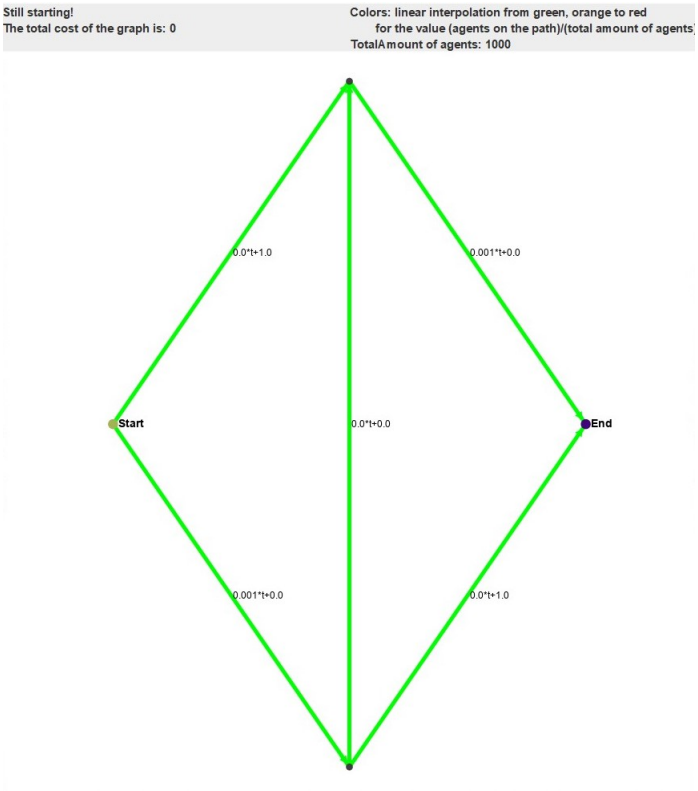
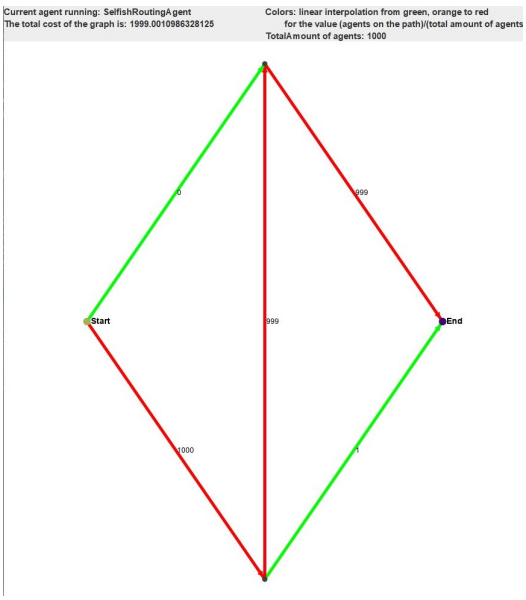


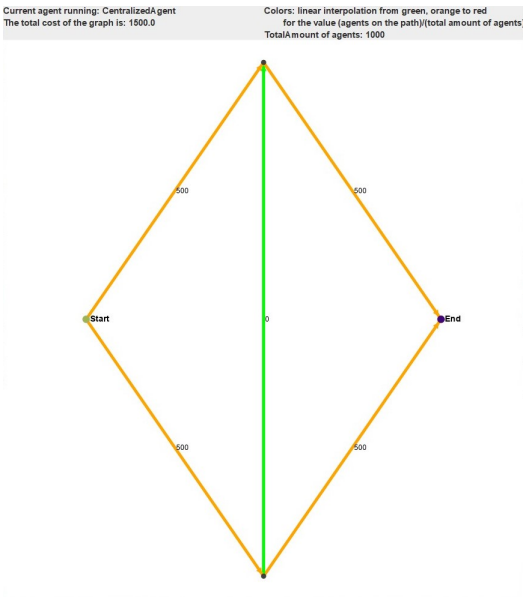
Figure 6: Network visualization with functions for Braess's Paradox

to orange to red for the amount of agents currently decided to run over the given edge in comparison to the total amount of agents in the simulation. At the top of each graph, there is a short legend, containing the name of the model currently running on the agents, the total amount of latency cost incurred if the current agents would start their travel now, as well as a short introduction to the colors of the edges. Each edge at the beginning shows its own weight function, in our case only linear functions, and during the animations the amount of agents, that decided to take this edge in their path.

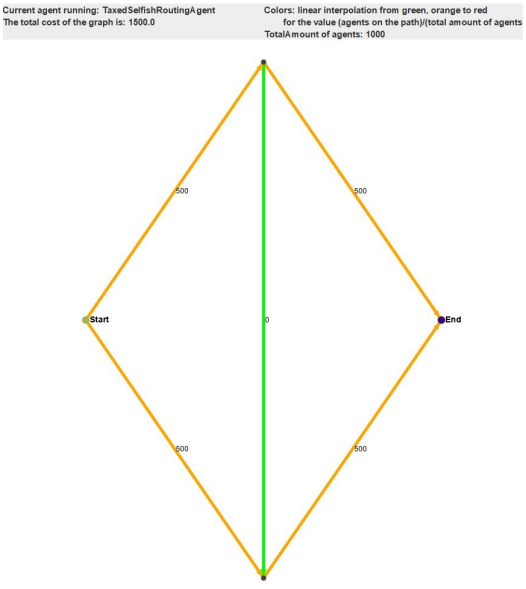
In Figure 6 you can for example see the visualisation of the Brass's paradox with the central edge before the animation begins, i.e. with their cost functions shown. In the subgraphs of Figure 7 you always the end state of the network congestion for each agent implementation. Namely the selfish and central model implementations at Subfigures 7a and 7b. The Subfigure 7c shows the taxed selfish agent with an arbitrary tax on its vertical edge that allows it to reach optimal network latency. In Figure 7d the same agent is shown but now with marginal taxing in the network, which results in a higher total cost for all of the agents, even if the distribution of them is the same.



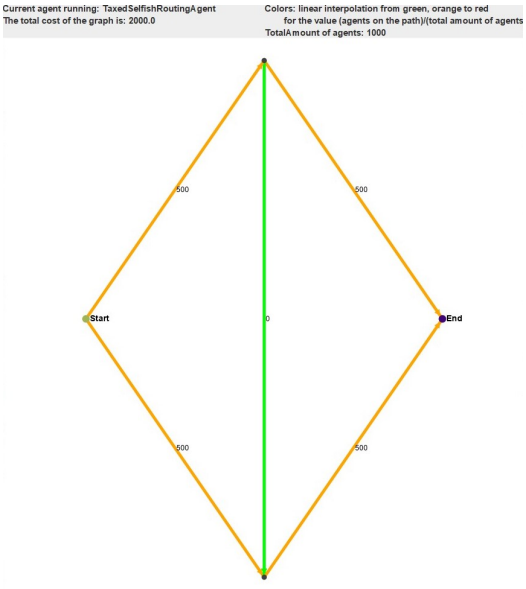
(a) End state selfish agents



(b) End state central agent



(c) Taxed selfish agent with arbitrary taxes



(d) Taxed selfish agent with marginal taxes

Figure 7: Stills from simulation visualization of Braess's paradox with central edge

4 Simulations of models

4.1 Simulation of selfish routing

A reason why selfish routing is a wide used concept in computer science comes from the facts that it is very simple, every agent can decide on its own and only little information is needed about the network. This becomes even more important in a network like the internet where a centralized solution is not feasible.
All we need for the simulation is a graph, its instance and an algorithm that finds the shortest path from the starting node s to the end node t. For the algorithm we use Dijkstra's algorithm. The only information an agent needs except for the graph instance is the current latency cost of the edges that depends on the congestion of the edge.

Implementation The actual implementation is therefore rather simple as we will see. Every agent that wants to send a package calls the following function. We need a graph denoted by G and the latency function for the edges denoted by l as parameters. The latency function gets updated whenever an agent takes a path.

Algorithm 1 SelfishPath(G, l)

1: $path \leftarrow \emptyset$
2: *path* $\leftarrow$ *Dijkstra(G, l)*
3: *UpdateLatency(l, path)*

As mentioned we take Dijkstra's algorithm to find the shortest path but any other shortest path algorithm would work as well. The function takes the graph G and the latency function l to compute the shortest path for the agent. Then we update the latency function based on the knowledge that the current agent takes the calculated path.
In theory, we would see that all agents decide their path through several iterations over the graph. But in practice we care only about the first iteration, since we wanted to mimic more a real transport protocol that decides only once and then sends the package over the chosen path. For most examples it is the same as the mathematical definitions but be wary that some situations require to recalculate the cost, i.e. adding a new edge. This means for such examples, our algorithm does not find the Nash equilibrium. All the examples we mentioned here are not affected by this.

Run time analysis In computer science it is always important to gauge if a solution is feasible. Feasible means that the program will terminate in an acceptable time, i.e. a few milliseconds to find the shortest path. Therefore we make a run time analysis to see how the run time increases dependant on the number of nodes and edges in the graph. Assume we have a graph with n nodes and m edges where $n, m \in \mathbb{R}^+$.

- line 2: The optimal runtime for Dijkstra's algorithm for our case is $\mathcal{O}(n \log n + m)$ [Cor+09]. Since we did not care about the run time in our implementation that much in our solution, we used a simpler implementation that has a run time of $\mathcal{O}(n^2 + m)$.
- line 3: To update the latency function we need to go over every edge at most once and therefore the run time is $\mathcal{O}(m)$.

In the end the runtime depends mainly on the implementation of Dijkstra's algorithm since it is the main factor and overshadows all other factors. The total runtime for our implementation is therefore $\mathcal{O}(n^2 + m)$. This is actually a really good runtime, remember in the big-O notation we get rid of all constant factors (they can still be significant!), but the highest power is very low with 2.

4.2 Simulation of taxed selfish routing

Some people might recognize that routing taxed selfish agents and selfish agents can be done the same way. In Section 2.2 Figure 2b in the case of marginal cost taxes we simply modify the graph for the edges $s - u$ and $v - t$ to have the cost function $c(x) = 2x$, where one x is the latency and the other is the tax for the taxed selfish agent. They are now combined into a higher latency cost for the selfish agent. In this case the now taxed selfish agent will take the exact same path as the centralized agent.
If we simply modify the edge $u - v$ to have a latency of $\frac{1}{2}$ the now taxed selfish agent will also avoid it like the centralized agent does. But for the sake of showing two approaches on how to solve the problem, we implemented it in the following way.
The simulation of the taxed selfish agents consist of 3 parts

Part 1 - Determine paths The first agent does all the calculations for the other agents, to save computation time.

Algorithm 2 GetDecision(G, s, d, index, totalAgents)

```
1: if index = 0 then
2:     paths ← GetUniquePaths(G, s, d)
3:     system ← GetUsage(G, paths)
4:     solution ← solve(system)
5: return choose(solution, paths, index)
```

paths and *solution* are public fields, which are accessible by all agents after the first agent calculated the solution.

Part 1 - Find all unique simple paths The first one is to find all unique paths from s to t, which is will be looked at in Part 1 in Section 4.3. In $\mathcal{O}(n^n)$ time it finds $\mathcal{O}(2^{(n-2)})$ unique paths.

Part 2 - Distribute usage from all paths Here we iterate over all paths and add the usage of each edge they go over to a pool. This is the same we did in the Tables 1 and 2 in Section 2.3. The algorithm is described in Algorithm 3.

Algorithm 3 GetUsage(G, P)

```
1:  paths ← P
2:  edges ← GetEdges(G)
3:  var ← [size(paths)]
4:  latency ← ∅
5:  system ← [size(paths)]
6:  for all i = 0, ..., size(paths) do
7:      p ← paths.get(i)
8:      for all j = 0, ..., size(p) do
9:          e ← p.get(j)
10:         edges(e) ← edges(e) + var[i]
11: for all i = 0, ..., size(paths) do
12:     p ← paths.get(i)
13:     for all j = 0, ..., size(p) do
14:         e ← p.get(j)
15:         system[i] ← system[i] + GetCostFromEdge(G, edges[e])
16:     system[i] ← system[i] = latency
17: return system
```

Part 3 - Solve the linear system of equations A equation solver called *z3* is used to solve all linear systems of equations created by setting two paths equal and solving for the unknowns. For an example see Section 2.3.

Run time - GetUsage

- line 5 - 9: Because there are at most $2^{(n-2)}$ unique paths and the inner loop has a complexity of $\mathcal{O}(e)$, the final complexity is $\mathcal{O}(e \cdot 2^{(n-2)})$
  - line 7 - 9: The for loop has a complexity of $\mathcal{O}(e)$ because any unique path is at most e long
    All the operations inside the loop are constant operations.

- line 10 - 14: Identical to for loop on line 5-9, so the complexity is $\mathcal{O}(e \cdot 2^{(n-2)})$

So the overall run time of this method is $\mathcal{O}(e \cdot 2^{(n-2)})$.

Run time - GetDecision

- line 2: The complexity is $\mathcal{O}(n^n)$ for finding all unique paths
- line 3: Because there are $\mathcal{O}(2^{(n-2)})$ unique paths the complexity is $\mathcal{O}(e \cdot 2^{(n-2)})$ for generating all linear functions
- line 4: The complexity to solve a linear system of equations is at most $\mathcal{O}(n^3)$, where n is the amounts of unknowns. This results in a final complexity of $\mathcal{O}(2^{(n-2)\cdot 3})$ for solving the linear system of equations
- line 5: The complexity to choose a path based on the result is constant, $\mathcal{O}(1)$.

Thus the overall run time of this algorithm is $\mathcal{O}(n^n)$ because $\mathcal{O}(e \cdot 2^{(n-2)}), \mathcal{O}(2^{(n-2)\cdot 3}) \in \mathcal{O}(n^n)$. However realistically there are not as many unique paths in all graphs, especially sparse one, like the graph of a street network, as not all nodes (intersections) are connected with all others.

4.3 Simulation of centralized routing

For the simulation of centralized routing the network agent the first agent takes the of role of the central entity who decides for all other agents. This means it will calculate the optimal solution based on the input graph and total number of agents in one go. All the agents that come after just look up where they are supposed to go.
To get the optimal routing the first agent will act in 3 parts as described in section 2.4. We will explain the algorithms in detail here.

Part 1 - Find all unique simple paths To find all unique simple paths we use a modified recursive depth-first-search (DFS). A `localPath`-list is used to keep track of all nodes in the current path. This list is added to a global `uniquePath`-list that keeps track of all found unique paths. The graph `G` as an adjacency list, the source node and destination node are passed to algorithm (4).[1]

[1] In general, for functions like `add`, `size` and `remove`, the function is applied to the last object passed to it.

Algorithm 4 GetUniquePaths(G, s, d)

```
1: pathList ← ∅
2: for all i = 0, ..., size(G) do
3:     isVisited[i] ← false
4: Add(s, pathList)                                   ▷ add first node
5: RecGetUniquePaths(s, d, isVisited[ ], localPath)
```

Algorithm 5 RecGetUniquePaths(u, d, isVisited[ ], localPathList)

```
1: isVisited[u] ← true
2: if u == d then                                     ▷ unique path found
3:     Add(localPathList, uniquePaths)
4:     isVisited[u] ← false                           ▷ u can be part of next path
5: for all int i in G[u] do
6:     if isVisited[i] == false then
7:         Add(i, localPathList)
8:         RecGetUniquePaths(i, d, isVisited[ ], localPathList)
9:         Remove(i, localPathList)                   ▷ continue path 1 step back
10: isVisited[u] ← false                              ▷ u can be part of next path
```

Part 2 and 3 go hand-in-hand. We calculate the total latency of a distribution once we find the distribution and only save the currently minimal latency `minDistLatency` and distribution in a Hashmap `distributionCost` globally.

Part 2 - Find all distributions of agents [2] To find the distribution we use a recursive algorithm over the unique paths and then over number of agents. This means our recursion tree is never deeper than the number of unique paths in the graph. However it becomes wide very quickly for large number of agents. The current path `p` we are working on, the `distribution` of agents over paths and and the number of agents left to distribute as `leftToDistribute` are passed to algorithm (6).

Part 3 - Find the minimal latency To calculate the minimal latency of a distribution of agents over paths, i.e. f_P, we need to get the number of agents on an edge, i.e. f_e. We now are able to to calculate the latency of each edge l_e. Then we use formula 1 to get the total latency. The `distribution` is used as input for algorithm (7).

[2] Generally a similar approach to the taxed selfish agent could be used. After getting the unique paths the total latency would have to be minimized. Since we get $p-1$ variables the total latency could be a polynomial of up to order $p-1$. This is best solved with approximation algorithms which was not the

Algorithm 6 RecDistributionCost(p, distribution[ ], leftToDistribute)

if $p ==$ size($uniquePaths$) $- 1$ **then** ▷ last path gets all remaining agents
  $distribution[p] \leftarrow leftToDistribute$
  $currentDistLatency \leftarrow$ CalcDistributionCost($distribution[]$)
  if $currentDistLatency < minDistLatency$ **then** ▷ new minimal latency found
    Clear($distributionCost$)
    Add(($currentDistLatency$, $distribution[]$), $distributionCost$)
  if $currentDistLatency == minDistLatency$ **then** ▷ another minimal latency found
    Add(($currentDistLatency$, $distribution[]$), $distributionCost$)
  return ▷ nothing is done if currentDistLatency $>$ minDistLatency
for all i $= 0, \ldots,$ $leftToDistribute$ **do**
  $distribution[p] \leftarrow i$
  RecDistributionCost($p + 1, distribution[], leftToDistribute - 1$)

Algorithm 7 TotalLatency(distribution[ ])

for all i in $uniquePaths$ **do**
  for all j in $uniquePaths[i]$ **do** ▷ add contribution of each path to edges
    $u \leftarrow j$
    $v \leftarrow j + 1$
    $flow[u][v] = flow[u][v] + \frac{distribution[i]}{totalAgents}$
for all $i = 0, \ldots,$ Size($flow$) **do** ▷ create latency matrix
  for all $j = 0, \ldots,$ Size($flow[i]$) **do**
    $latency[i][j] =$ GetLatency($flow[u][v]$)
$totLatency \leftarrow \sum_{e \in edges} latency(e) \cdot flow(e)$ ▷ calculate total latency
return $totLatency$

Run time analysis As discussed in chapter 2.4, the run time of this algorithm is exponential. We will do a worst case anaylsis of the implemented algorithm.
n a recursive tree where each node branches b times and the depth is d the number of function calls fc is

$$fc = b^d$$

We then get the runtime R by multiplying fc with the time it takes to execute the function. This way we get for the GetUniquePaths-algorithm with function cost $\mathcal{O}(n)$[3] the runtime R_u:

$$\begin{aligned} b &\in [0, \dots, n-1], \quad n = \#\text{ nodes} \\ d &\in [0, \dots, n-1] \\ fc &= (n-d)^{(n-1)}, \\ R_u &= fc \cdot O(n) \in O(n^n) \end{aligned}$$

The number of unique paths p possible is equal to all possible leaf nodes of the recursion tree. Or in a complete graph all possible ways to pick $k \in [0, \dots, n-2]$ nodes from $V \backslash \{s, t\}$. Using the sum over the binomial coefficients we get $p = 2^{(n-2)}$.
For the TotalLatency the runtime R_L is:

$$\begin{aligned} \text{line 1-5:} \quad & R_{\text{flow}} = p \cdot n \\ \text{line 6-8:} \quad & R_{\text{latency}} = n \cdot n \\ & R_{\text{totLatency}} = m \\ & R_L = R_{\text{flow}} + R_{\text{latency}} + R_{\text{totLatency}} \in O(n \cdot 2^n) \end{aligned}$$

R_L is the cost of one function call of the RecDistributionCost-algorithm. So we get runtime R_d:

$$\begin{aligned} b &\in [0, \dots, a], \quad a = \#\text{ agents} \\ d &\in [0, \dots, p-1] \\ fc &= a^{(p-1)} \\ R_u &= fc \cdot O(n \cdot 2^n) \in O(a^{n \cdot 2^n} \cdot n \cdot 2^n) \end{aligned}$$

Therefore the complete algorithm has a worst case runtime $R \in O(n^n + a^{2^n})$.[4]

---

solution we were looking for.

[3] Add and Remove are $O(1)$ for a linked list and $O(n)$ for an array list.

[4] which is absolutely ridiculous

5 Discussion and results

In hindsight on SimEngine As is the case with many projects, in hindsight you are able to find many flaws in the way of creating the whole framework, which is no different this time. As a result all of us learned new ways, such as more precisely defining beforehand what exactly is expected from each part of the project or that you probably should first look into the question if it is worth reinventing the wheel when dealing with the difficulties of working quickly and with only a vision at the end instead of a strict goal.
In general though, the simulation engine as well as the visualisation reached close to our initial idea and are in our opinion quite helpful to understand the general problems of routing in a game theoretical sense more easily. We hope this project first off may be able to help someone in the future to understand and grasp this quite intriguing subject, which is applicable nearly everywhere in our daily lives, more easily and with more fun. Further we would enjoy seeing someone else pick up where we left of and extend the whole simulation system further, such as implementing non-linear functions and agent implementations which can handle those functions, creating new and interesting network typologies to run the models on, as well as maybe even extending the model to more than single commodity flow.

Conclusion of selfish routing For networks with linear latency functions the worst-case ratio is $\frac{4}{3}$ as we have seen. This is something we must consider if we choose a viable protocol. On the other hand, the run time is better than any other method we looked at. Surely for small networks, where the run time is irrelevant it is a suboptimal solution but for big and complex networks the run time is more important. Remember we speak of an increase factor in cost of $\frac{4}{3}$ but the runtime decreases from a double exponential time (optimal solution with centralized routing) to quadratic time. However, we have seen that taxes can drastically improve the situation and the run time increase is not notable once the right tax has been found.
If we consider all latency functions, we have seen that the worst-case ratio can explode to infinity and therefore it is not a good solution. In this context a study to examine if such a phenomenon really occurs in used networks and if so, how often it occurs would be quite interesting.
In conclusion it is clear that selfish routing can be quite bad in certain situations and that there are better solutions.

Conclusion of taxed selfish routing Taxed selfish agents are a good improvement over the selfish agents but aren't always a better option as the taxes have to be chosen wisely. Otherwise agents may have to pay taxes which will make the total cost the same, as if there weren't taxes. But smartly placed taxes can improve the latency and the cost each agent has to pay. The only thing stopping it to use the full potential of the agents is the non

polynomial time it takes to find the unique paths and that the implementation is limited to one source and one destination.
Anyway, there are a many ways to improve this implementation. As for example, we can replace the method, which finds the unique paths with either Yen's algorithm which can find the k shortest paths from a source node to a destination node in $\mathcal{O}(kn(e + n \log n))$ where n and e are the nodes and edges respectively. If one has an approximation for how many unique paths there are in a sparse graph, you can find all unique paths in close to polynomial time. Hopefully much better than our current solution which finds all of the unique paths in $\mathcal{O}(n^n)$.
Another way might be to modify Bellman-Ford's algorithm or any other algorithm that calculates the shortest path between a source node and a destination node, so that it can work with the variable edge cost. I.e. it builds up a function as the result and anytime there is a comparison of which is shorter it simply adds a *min* function to the current "shortest path distance" function. And in the end one can use solvers to determine the best distribution of agents over several paths, based on the different types of agents.

Conclusions on centralized routing Centralized routing is the most basic idea how to handle routing. If there is latency proportional to traffic involved, there is only one aspect in which it truly surpasses all other routing models: it can find the optimal solution. In all other aspects, however, it is not competitive. The system has very hard restrictions which make it unusable in real life. Networks nowadays are huge and never really closed. The rate over the network is never constant and flow cannot be viewed in single entities. Further networks have more than one source and destination. The final and most important aspect is the exponential runtime.
The only real life situation where it might be useful is optimizing traffic in street network in a city. Assuming the number of cars using the network is similar each day, the city is small then the only restriction unfulfilled is that there is only one source and destination.
In our implementation it was clearly visible how soon the exponential runtime comes into play. For a Braess paradox of size 2 there are 5 possible unique paths between source and destination. We generally simulated 2000 agents on a graph. We did a test run using 200 agents which took about 50 seconds. Extrapolating from that using the simplified formula runtime = $\text{agents}^{\text{paths}}$ we get that for 2000 agents it should take $5 \cdot 10^6$ seconds or about 57 days. We decided on 500 agents which by the formula would take about 1 hour and 21 minutes. It actually took 39 minutes which is about half. Surely there are improvements possible to the code provided but they will only have a minor effect since there is no polynomial approach possible.
So overall, centralized routing is nice to play around with but never was a serious contender for an efficient and usable routing protocol where non-constant latency is involved.

In conclusion we learned and understood a lot more about these routing models through the challenge of implementing them. It is really quite interesting to see how the taxed selfish agent can be similar to the selfish and the centralized agent. It seems to connect these concepts which we never anticipated. We hope this paper reflects the research others have done well and gave an coherent introduction to game theory in routing.

References

[Bra68] Dietrich Braess. "Uber ein paradoxon der verkehrsplanung". In: *Operations Research* 12 (1968), pp. 258–268. DOI: https://doi.org/10.1007/BF01918335.

[Fri02] Eric J Friedman. *A generic analysis of selfish routing*. Tech. rep. Cornell University Operations Research and Industrial Engineering, 2002.

[Rou02] Tim Roughgarden. "The Price of Anarchy is Independent of the Network Topology". In: *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*. STOC '02. Montreal, Quebec, Canada: Association for Computing Machinery, 2002, pp. 428–437. ISBN: 1581134959. DOI: 10.1145/509907.509971. URL: https://doi.org/10.1145/509907.509971.

[RT02] Tim Roughgarden and Éva Tardos. "How Bad is Selfish Routing?" In: *J. ACM* 49.2 (Mar. 2002), pp. 236–259. ISSN: 0004-5411. DOI: 10.1145/506147.506153. URL: https://doi.org/10.1145/506147.506153.

[CDR06] Richard Cole, Yevgeniy Dodis, and Tim Roughgarden. "How much can taxes help selfish routing?" In: *Journal of Computer and System Sciences* 72.3 (2006). Network Algorithms 2005, pp. 444–467. ISSN: 0022-0000. DOI: https://doi.org/10.1016/j.jcss.2005.09.010.

[Cor+09] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009, p. 663.

[Par11] Simon Parsons. "Algorithmic Game Theory by Noam Nisan, Tim Roughgarden, Éva Tardos and Vijay V. Vazirani, Cambridge University Press, 754 pp.,£ 32.00, ISBN 0-521-87282-0". In: *The Knowledge Engineering Review* 26.1 (2011), pp. 71–72.

[htt20] araruna (https://stackoverflow.com/users/668574/araruna). *Find all paths between two graph nodes — stackoverflow*. https://stackoverflow.com/a/14621293. [Online; accessed 8-June-2020]. 2020.

[Wik20] Wikipedia contributors. *Routing — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Routing&oldid=959726764. [Online; accessed 8-June-2020]. 2020.

[T+] Oberdörfer T. et al. *RoutingSimEngine*. URL: https://github.com/Toroto006/routingSimEngine.

[Tea] GraphStream Team. *GraphStream Project*. URL: http://graphstream-project.org/. (accessed: 12.06.2020).

[tea] Json.org team. *JSON lib for Java*. URL: https://www.json.org/json-en.html.

[Z3] Microsoft Z3. *The Z3 Theorem Prover*. URL: https://github.com/Z3Prover/z3.