Game Theory Report

Filip Rak

Daniel Michalak

Dawid Mączka

Warsaw, 11th January 2024

Contents

1 Definitions **4**
- 1.1 Routing game . . . 4
  - 1.1.1 Paths . . . 4
  - 1.1.2 Flows . . . 4
  - 1.1.3 Remarks . . . 4
- 1.2 Selfishness . . . 4
- 1.3 Atomic vs non-atomic . . . 5
- 1.4 Nash Equilibrium . . . 5
  - 1.4.1 Discrete . . . 5
  - 1.4.2 Continuous . . . 5
- 1.5 Price of Anarchy . . . 5
  - 1.5.1 Bounds on the Price of Anarchy . . . 5

2 Assumptions **7**
- 2.1 Parameters . . . 7
- 2.2 Graphs . . . 7

3 Data storing **7**
- 3.1 Graph representation . . . 7

4 Navigating GUI **8**

5 Miscellaneous algorithms **9**
- 5.1 Finding all paths in a graph . . . 9
  - 5.1.1 Introduction and description . . . 9
  - 5.1.2 Complexity analysis . . . 9
- 5.2 Nash Equilibrium evaluation . . . 10
  - 5.2.1 Algorithm description . . . 10
  - 5.2.2 Time complexity . . . 10

6 Nash Equilibrium finding algorithm - iterative method **10**
- 6.1 Introduction . . . 10
- 6.2 Time Complexity . . . 10
- 6.3 Nash Equilibrium convergence with Nash Equilibrium check . . . 10
- 6.4 Introduction . . . 10
- 6.5 Convergence analysis - overview . . . 11
- 6.6 Convergence - single Nash . . . 11
- 6.7 Convergence multi Nash . . . 12

7 Nash Equilibrium - greedy algorithm **13**
- 7.1 Algorithm description . . . 13
- 7.2 Time complexity . . . 13

8 Social Optimum - iterative method **13**
- 8.1 Introduction . . . 13

9 Remarks and Conclusions **13**

10 Numerical method **14**
10.1 Continuous case 14
10.1.1 Social optimum 14
10.1.2 Nash equilibrium 14
10.2 Continuous case - Results 15
10.2.1 Social optimum (by the first approach - minimize $C(f)$) 15
10.2.2 Social optimum (by the second approach - equalize marginal costs c^*) . . 16
10.2.3 Nash equilibrium (by the first approach - minimize $\phi(f)$) 17
10.2.4 Nash equilibrium (by the second approach - equalize costs c) 18
10.3 Discrete case 19
10.3.1 Social optimum 19
10.3.2 Nash equilibrium 19
10.4 Discrete case - Results 20
10.5 Remarks and Conclusions for Numerical approach 21
10.5.1 Constraints and Penalty functions 21

11 Literature **23**

The goal of this project was to explore mathematical models for atomic and non-atomic selfish routing games.

1 Definitions

1.1 Routing game

A routing game can be defined as a triplet (G, r, c). It's defined on a graph $G = (V, E)$ with a defined set of vertices called sources S, and sinks T. Each source (or rather a source-sink pair, but it's much less intuitive) has corresponding out-flowing traffic r_i (source's capacity, e.g. 1.0 in non-atomic or 10 in atomic case) that obviously must travel along the edges of G from $s_i \in S$ to $t_i \in T$. Every edge $e \in G$ has associated with it a non-negative, continuous, and non-decreasing cost function (also called latency function) [1] [2].

1.1.1 Paths

For any given triplet (G,r,c) we denote by P_i the set of paths available to source s_i, with t marking the sink vertex. [2]. Example:

$$P_1 = \{s_1vt, s_1t\}$$
$$P_2 = \{s_2vt, s_2t\}$$

The focus will be primarily on games with only one source, where we will omit the lower index for the path set P.

1.1.2 Flows

A flow is a vector representing the quantity of traffic along the paths. A flow is called feasible if:

$$\sum_{P \in P_i} f(P) = r_i$$

This translates to a very natural notion that the sum of components of f must be equal to the source's out-flowing capacity (traffic). E.g. $r = 1$, $f = \{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\}$, and $\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1$. Flow f is feasible.

1.1.3 Remarks

This definition proves very useful, as it can be easily connected with well-known concepts in Game Theory. The pairs s_i, t_i induce paths composed of edges, which together form a set of available strategies for the players. The payoff function, also known as the utilitarian function, represents a value that players aim to either maximize, such as profit, or minimize, such as the latency due to congestion. Specifically, in the context of a routing game, the objective is to minimize latency [1].

1.2 Selfishness

If every player has selected a strategy from their set of strategies, then each player receives a payoff determined by their own choice and those of their competitors. We assume player rationality, meaning the players do not select their strategies randomly. Instead, they choose strategies that match their goal, i.e., to minimize the payoff, which in this context is the latency on the possible path [1]. For now, we will set aside the optimal case for later discussion when we contrast the resultant equilibrium from selfish choices.

1.3 Atomic vs non-atomic

Also called discrete and continuous versions of the game respectively. From now on, we will use both terms interchangeably. In a non-atomic routing game, each player controls an infinitesimal part of the flow. In the atomic case, each player controls only one unit of flow[4]. In the non-atomic game the flow can be split in an arbitrary way and each infinitesimal amount of flow is controlled by a different player. [12]

1.4 Nash Equilibrium

1.4.1 Discrete

A flow f is a *Pure Nash Equilibrium* if, for every player i and any alternative path P_i' that player i might take, resulting in a new flow configuration f', the following inequality holds:

$$\text{cost}_i(f) \leq \text{cost}_i(f')$$

where f' is the flow configuration when player i uses P_i' while all other players stick to their paths in f. In other words, no player can reduce their individual cost by unilaterally changing their path[4].

Every atomic selfish routing game (with any cost function for edges) has at least one Pure Nash Equilibrium flow.[4](Theorem 5.5), but they might not be unique.

1.4.2 Continuous

The flow f is an equilibrium flow if for every pair $P, \tilde{P} \in P_i$ of s-t paths with $f_P > 0$, the condition $c_P(f) \leq c_{\tilde{P}}(f)$ holds. In other words, a Nash flow ensures that all used paths have minimal costs. Specifically, all paths used by an equilibrium flow have equal costs.

Every non-atomic selfish routing game has a Pure Nash Equilibrium (corresponding to the potential function minimum).[4](Theorem 5.5).

Non-atomic equilibrium flows always exists, and all have equal cost, and PoA = PoS [12](page 40). That implies that any found nash equilibrium is satisfiable.

1.5 Price of Anarchy

Price of anarchy quantifies the inefficiency of equilibrium. It's given by

$$PoA = \frac{\text{cost of worst NE flow}}{\text{cost of optimal flow}}$$

1.5.1 Bounds on the Price of Anarchy

What's interesting is that the Price of Anarchy (PoA) can be arbitrarily large in some cases. Consider Figure 1(b), where we assume the traffic is $r = 1$. Under this assumption, the equilibrium travel time is 1, as the lower edge is a dominant strategy. This dominance occurs because the lower edge is never worse than the upper edge, which has a constant cost of 1. However, for the optimal solution, the travel time tends to 0 as $p \to \infty$. This result is due to the flow being split such that the flow component on the upper edge equals ϵ, and on the lower edge equals $(1 - \epsilon)$, with the majority of the flow on the lower edge. Consequently, the travel time on the upper edge becomes negligible due to the virtually nonexistent flow, and on the lower edge, it is negligible because of the virtually nonexistent cost.

$$\text{PoA} = \frac{0 \cdot 1 + 1 \cdot 1^p}{1 \cdot \epsilon + (1-\epsilon)^p \cdot (1-\epsilon)} \xrightarrow{p \to inf} \frac{0 \cdot 1 + 1 \cdot 1}{1 \cdot \epsilon + 0 \cdot (1-\epsilon)} \xrightarrow{\epsilon \to 0} \frac{0 + 1}{1 \cdot 0 + 0 \cdot 1} \to \frac{1}{0} \to \infty$$

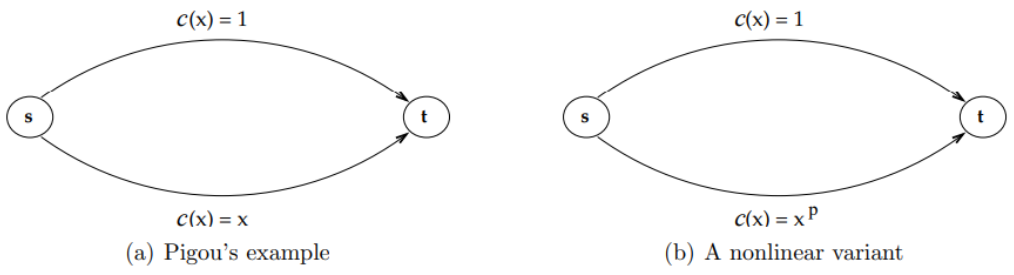


Figure 1: Pigou's example and a non-linear variant.

This was a specific case, but what's even more interesting is that networks resembling Pigou's example, with parallel edges (as shown in Figure 2), serve as models to establish an upper bound for the Price of Anarchy (PoA). Among all networks with cost functions in a set C, the largest PoA is observed in a Pigou-like network [5]. For a more detailed understanding, I refer to the formal proof in [6] for further reading. This is particularly useful because the bound is both easy to understand and calculate. In the equilibrium case, all traffic utilizes the lower edge, while in the optimal case, nearly all traffic is on the upper edge. This bound is known as the Pigou bound $\alpha(C)$, and is defined as follows [5]:

$$\alpha(C) := \sup_{c \in C} \sup_{r \geq 0} \sup_{x \geq 0} \left\{ \frac{r \cdot c(r)}{x \cdot c(x) + (r - x) \cdot c(r)} \right\}$$

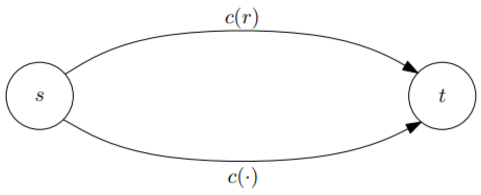


Figure 2: Pigou-like network with an arbitrary cost function

The key takeaway is that the upper bound for PoA in our study, specifically for quadratic functions, is $\frac{3\sqrt{3}}{3\sqrt{3}-2} \approx 1.6$ [5].

The lower bound for PoA is, obviously, 1, since the optimal travel time in the denominator is guaranteed to be the lowest.

This extends our testing framework, ensuring that the calculated Price of Anarchy (PoA) falls within the range of [1, 1.6].

2 Assumptions

Since we want our model to be applicable for real life problems we introduce some assumptions regarding the graphs and parameters that will be an input for the model. The assumptions will increase the performance and reduce the space consumption of the algorithm making it faster without loss of generality.

2.1 Parameters

Since we want to solve problems of optimization of the payload splitting on the graphs we assume that the edge payload functions are a functions that are increasing functions and for all possible flow values the value of this function will be not negative.

2.2 Graphs

When it comes to graphs our algorithm will accept only directed graphs. However, one should keep in mind that if we want to solve problems for undirected graphs we just have to make sure to add two edges instead of one. This means that if we want to have undirected graph G for every edge (v_i, v_k) we have to add edge (v_i, v_k) and edge (v_k, v_i).

3 Data storing

In this part of the report the problem of how to store a data will be covered. We will go through all elements that we need to store so that the algorithms will work and no information will be lost in the process.

3.1 Graph representation

When it comes to the graph representation for each graph we will store the following information:

1. number of vertices - this helps us in the process of creation of the graph since other elements sizes are based on the number of vertices in a graph and in algorithms the size of the graph is used in many of the steps.

2. parameter matrix - the parameters matrix stores the parameters for each edge. This means that this is n by n matrix where n is the number of vertices in the given graph. Each cell in this matrix is the tuple of size 3. Each element of the tuple corresponds to the proper parameter of the latency function that is the quadratic functions. This means that first element is parameter a, second is parameter b and the last element is the parameter c of the function $ax^2 + bx + c$.

3. list of paths - the list of the paths of the graph helps us to keep track of what is the flow on each path since it is necessary to keep the track of it in the process of Nash Equilibrium checking.

4. distribution matrix - the distribution matrix is used to keep track of the flow between any two vertices. The size of this n by n where n is the number of vertices in a graph.

4 Navigating GUI

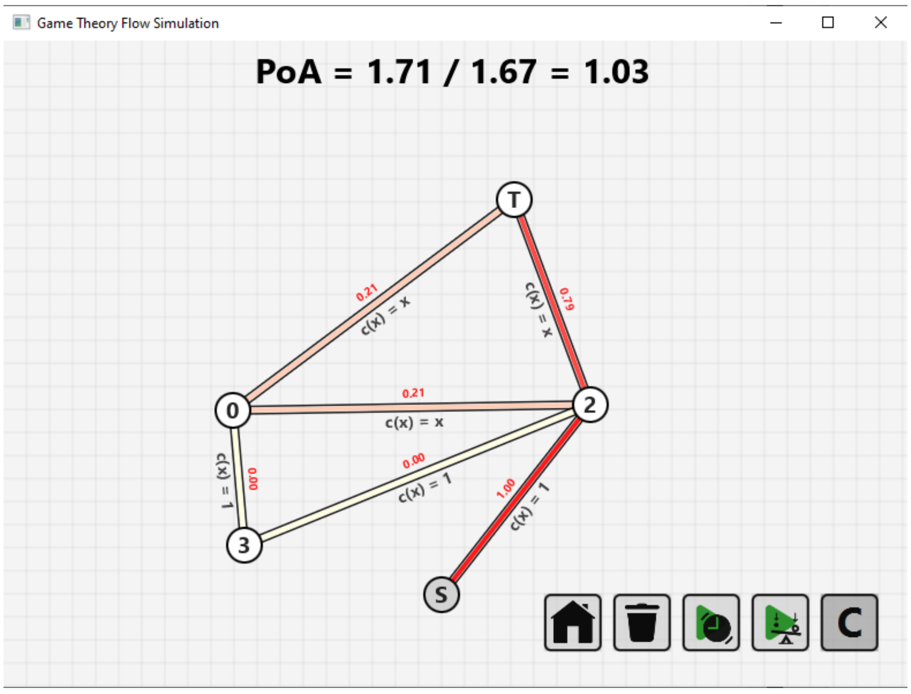


Figure 3: GUI for graph editing

Button drawer:

- Return home
- Clear graph
- Run simulation for social optimum
- Run simulation for nash equilibrium
- Continuous / Discrete case toggle

Mouse events:

- Right click - add Vertex
- Click (on Vertex) + drag - add Edge
- Shift + Click (on Vertex / Edge) - select
- Ctrl + Click + drag - drag the canvas
- Double click (on Vertex) - set source / sink
- Double click (on cost function) - set cost function

Keyboard events:

- (When selected) Esc - reset selection
- (When selected) Del - delete selected

5 Miscellaneous algorithms

In this section the algorithms used in the project will be analyzed, explained and the example of how it works will be shown. At the end of each algorithm section, the time complexity will be evaluated. Moreover, not only the algorithms for flow distribution evaluation will be explained, but also all algorithms that were created and are non-trivial.

5.1 Finding all paths in a graph

5.1.1 Introduction and description

To find all paths in the graph from v_0 to v_1 where v_0 is the starting point and v_1 is the destination, we make use of DFS [11] and backtracking [10]. The algorithm is very simple and the pseudocode is presented in Figure 4:

```
function FINDPATHS($v, P, X, V$)
    $N \leftarrow$ NEIGHBORS($v$)
    if $v = v_1$ then
        $P \leftarrow P \cup \{X\}$
    end if
    for each neighbor $n$ in $N$ do
        if $n \in V$ then
            backtrack
        else
            $X \leftarrow X \cup \{n\}$
            $V \leftarrow V \cup \{n\}$
            FINDPATHS($n, P, X, V$)
        end if
    end for
end function
```

Figure 4: Find paths using DFS and backtracking

In the algorithm above, the neighbors(v) function return the neighbors of the vertex v which are the vertices such that there exists an edge between vertex v and any vertex from the set of neighbors.To start the algorithm we run it with the initial vertex v_0, empty set P, empty set X and set V with v_0 inside. Paths are stores in the set P.

5.1.2 Complexity analysis

As we can see, we are looking for all the paths in the graph between two points. This is just a simplified algorithm for finding all paths in the graph between any two vertices. In that case the time complexity of such algorithm is bounded by the number of maximal paths which in that case is $2^|V|$ where $|V|$ is the size of the set of all vertices, thus the time complexity for such algorithm is $O(2^|V|)$. By applying the same reasoning for our case. The number of all paths is equal to $2^|V-2|$. This leaves us still with the time complexity upper bounded by $2^|V|$.

5.2 Nash Equilibrium evaluation

5.2.1 Algorithm description

The algorithm to check if the current flow distribution is in Nash Equilibrium is pretty easy and is based on the theory from the previous sections. When we are in Nash Equilibrium, this means that for every path in the graph, the current latency function value is lower for every other latency functions after changing the fixed amount of flow. This means that if we have a set P of all paths, function f(p, x) that return value of latency function for path p with x as the flow capacity running through path p, function g(p) which return flow capacity on path p and d to be the smallest change of flow then we are in a Nash Equilibrium if :

$$\forall_{p_1,p_2 \in P} f(p_1, g(p_1)) < f(p_2, g(p_2) + d)$$

This means that if there exist a path for which the value of the latency function is smaller after moving a fixed flow, then we are not in the Nash Equilibrium.

5.2.2 Time complexity

When comparing every two paths we are traversing all paths for every path which means we have to iterate at most n^2 where n is a number of paths. Moreover, traversing we are at each step evaluating the value of the latency function, but this is done in linear time so we are left with the complexity of n^2.

6 Nash Equilibrium finding algorithm - iterative method

6.1 Introduction

This method guarantees that we will go over all possible distributions of the flow [4]. This algorithm is excellent for graphs with small number of paths between source and destination. For each possibility, it checks if we got the Nash Equilibrium. It is a simple method, and it returned all Nash Equilibras that are in a graph.

6.2 Time Complexity

In this algorithm, as mentioned above, we are going through all possible solutions. This number is equal to the number of methods that we can put n indistinguishable players into k distinguishable paths and it is equal to $\binom{k+n-1}{n}$. The time complexity is upper bounded by this number which means that the time complexity equals O($\binom{k+n-1}{n}$) where n is the flow number and k is the number of paths.

6.3 Nash Equilibrium convergence with Nash Equilibrium check

6.4 Introduction

As we can notice from the previous sections, we have an algorithm for checking if the given distribution is in Nash Equilibrium. We can change the algorithm in such a way that it looks for two paths, such that the flow can be moved from one path to the other. It would return true if the flow was moved and false in other case. This means that if we put this algorithm in a loop, we are slowly moving the flow in such a way that at some point we should be left with the Nash Equilibrium. This means that as long as the algorithm will stop, we are able to find one proper Nash Equilibrium. When changing the flow in a given graph, what we are doing is getting closer with our flow distribution to the distribution of Nash Equilibrium. This means that the average difference for each path is reduced after each iteration and when the average difference reaches 0 then the algorithm stops.

6.5 Convergence analysis - overview

In the previous subsection, we described a method of using the Nash Equilibrium checking algorithm in order to find a Nash Equilibrium. When analyzing the convergence, we will go over two examples. First, when we have one Nash Equilibrium and the other case, when we have more than one Nash Equilibrium. In our case, the average difference is the sum of the absolute values between the flow distribution of Nash Equilibrium and our current distribution. When doing such simulations, we will start with randomly distributed flow. This happens due to the fact that as we decrease the latency we decrease the potential of the flow and us we know we can reduce the potential as long as we exhaust all of the potential and we are left with the final version which is the Nash Equilibrium.

6.6 Convergence - single Nash

In case of the single Nash when performing simulation our solution will converge linearly to the Nash Equilibrium. by saying linearly we mean that the average error will linearly reduce from maximum (initial error) to 0. Below there is an example of how it looks like for a simple graph:

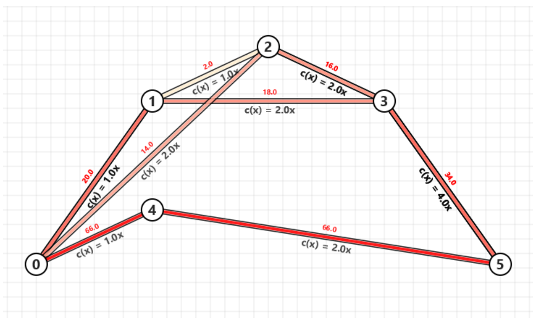


Figure 5: Final network view

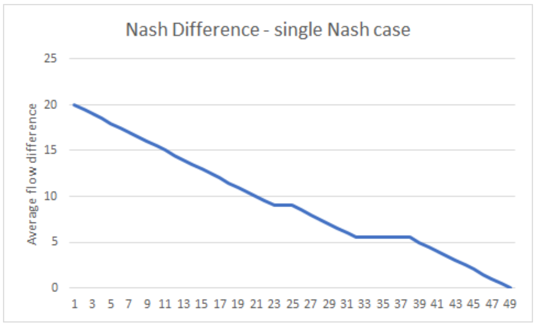


Figure 6: Nash convergence

6.7 Convergence multi Nash

In case of multi Nash graphs the convergence will be done to the Nash for which the least number of iterations is needed. Below there is a graph and convergence diagram:

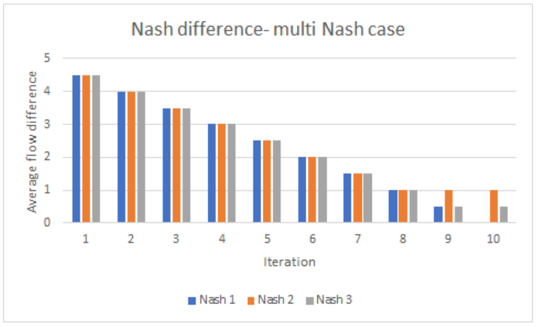


Figure 7: Final network view

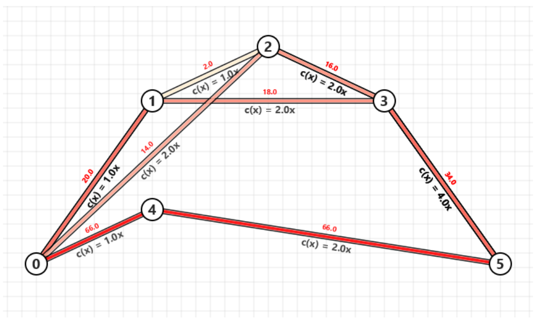


Figure 8: Nash convergence

7 Nash Equilibrium - greedy algorithm

7.1 Algorithm description

Since we know that we can use a simulation to find a Nash Equilibrium we can create a greedy algorithm that will randomly distribute the flow and then will proceed to simulation and will add the result to the set of results. Since we can have a network with multiple Nash Equilibria we have to repeat this process multiple times. The more times we will do the simulation with random distribution the lower the chance of missing any Equilibrium

7.2 Time complexity

When we do the simulation we are moving flow one by one into the place with lower potential. When moving flow we will have to move at most max flow (n). When performing algorithm we input the parameter p which tells us how many times we have to repeat the simulation. Since complexity of each simulation is O(n) then the complexity of the whole algorithm is $O(n^p)$ which means we can adjust the number depending on how precise we want the results to be.

8 Social Optimum - iterative method

8.1 Introduction

Just like in the case of Nash equilibrium we traverse through all possible distributions of the flow. We are using the fact that the number of combinations is finite. For each traverse we check if the total flow value is lower than the one we stored. When we talk about the total flow value we mean the sum of values of the path where value of each path is the sum of values of latency functions for each subpath with the flow amount as an input. This means that the equation for calculating total flow equals:

$$\sum_{i=1}^{n} \sum_{s \subset S_i} f(i)\dot{l}_s(f(i))$$

where n is the number of paths S_i is a set of subpaths for i-th path, $l_s(x)$ returns a value of latency function for subpath s and $f(i)$ returns the value of flow flowing through the i-th path.

9 Remarks and Conclusions

After evaluations and going through all the possibility we discovered that we have at least two algorithms for Nash Equilibrium since we know that the simulation method leaves as with a pure Nash Equilibrium. However, due to the fact that we can split more than one player into the other paths we can't be sure that the same logic can be applied to the social optimum. In this case further calculations should be made to find a method to minimalize the game value function induced from the paths.

10 Numerical method

The flow vector must be feasible, flows must be non-negative. Costs are calculated on the edges after flow components for every path have been added to the current flow component on the edge.

10.1 Continuous case

10.1.1 Social optimum

The calculation of the **optimal** travel time is done by minimizing the:

$$C(f) = \sum_{e \in E} c_e(f_e) f_e$$

The optimal flow is marked as f^*. The flow is minimal since the $C(f)$ expresses the travel cost in a general sense [3] [12](page 33).

The second way to calculate the **optimal** travel time is to use the marginal cost functions given by:

$$c^* = \frac{d}{dx}(xc(x))$$

And calculate the Nash flow $f^\sim$ for game (G,r,c*) [3] [12](Corollary 6).

10.1.2 Nash equilibrium

The calculation of the **equilibrium** travel time is done by minimizing the potential function ϕ:

$$\phi(f) = \sum_{e \in E} \int_0^{f_e} c_e(x)\, dx$$

The equilibrium flow is marked as $f^\sim$. A feasible flow f is a Nash Flow iff it is a minima for $\phi(f)$ [3] [12](Lemma 7).

The second way to calculate the **equilibrium** flow is to minimize the dissimilarities in the cost functions. This can also be used for calculating the optimal travel time in the second approach.

Minimizing the dissimilarities

The objective function I've decided to use is the sum of squared deviations, very similar to MSE. The goal is to minimize the variance of the costs.

$$\text{Objective Function}(\mathbf{c}) = \sum_{i=1}^{n} (c_i - \bar{c})^2$$

10.2 Continuous case - Results

10.2.1 Social optimum (by the first approach - minimize $C(f)$)

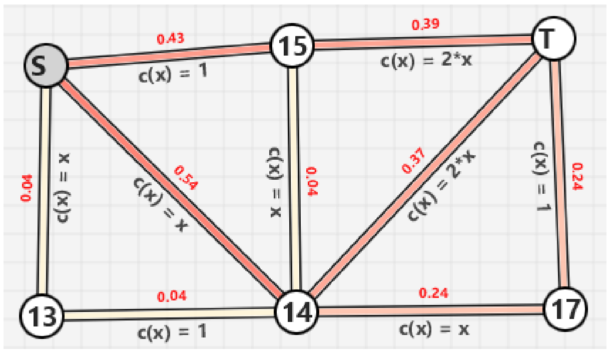
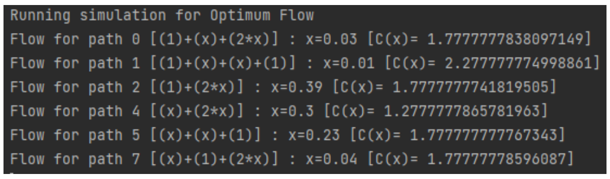


Figure 9: Optimal flow f^* as a result of minimizing the cost function $C(f)$. Avg. travel time = 1.63.



```
Running simulation for Optimum Flow
Flow for path 0 [(1)+(x)+(2*x)] : x=0.03 [C(x)= 1.7777777838097149]
Flow for path 1 [(1)+(x)+(x)+(1)] : x=0.01 [C(x)= 2.277777774998861]
Flow for path 2 [(1)+(2*x)] : x=0.39 [C(x)= 1.7777777741819505]
Flow for path 4 [(x)+(2*x)] : x=0.3 [C(x)= 1.2777777865781963]
Flow for path 5 [(x)+(x)+(1)] : x=0.23 [C(x)= 1.777777777767343]
Flow for path 7 [(x)+(1)+(2*x)] : x=0.04 [C(x)= 1.77777778596087]
```

Figure 10: The result is fairly close to the expected.

Path	Simulated Flow	Real Flow	Difference
1+x+2x	0.03	0	0.03
1+x+x+1	0.01	0	0.01
1+2x	0.39	0.41	-0.02
x+2x	0.30	0.36	-0.06
x+x+1	0.23	0.23	0.00
x+1+2x	0.04	0	0.04

Table 1: Comparison of Simulated and Real Flow Values

10.2.2 Social optimum (by the second approach - equalize marginal costs c^*)

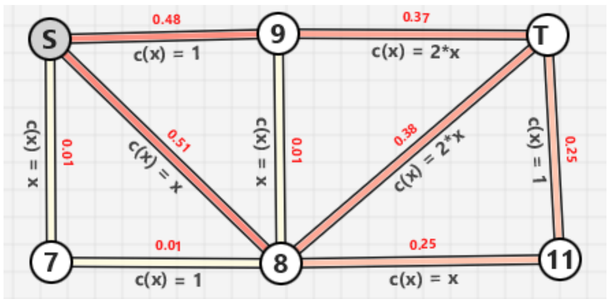
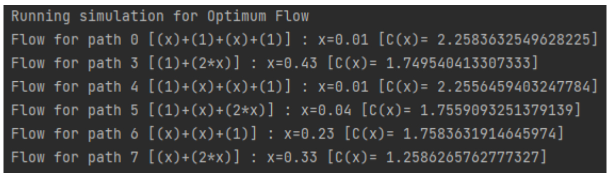


Figure 11: Optimal flow f^* as a result of minimizing the dissimilarities of the marginal cost functions. Avg. travel time = 1.63



```
Running simulation for Optimum Flow
Flow for path 0 [(x)+(1)+(x)+(1)] : x=0.01 [C(x)= 2.2583632549628225]
Flow for path 3 [(1)+(2*x)] : x=0.43 [C(x)= 1.749540413307333]
Flow for path 4 [(1)+(x)+(x)+(1)] : x=0.01 [C(x)= 2.2556459403247784]
Flow for path 5 [(1)+(x)+(2*x)] : x=0.04 [C(x)= 1.7559093251379139]
Flow for path 6 [(x)+(x)+(1)] : x=0.23 [C(x)= 1.7583631914645974]
Flow for path 7 [(x)+(2*x)] : x=0.33 [C(x)= 1.2586265762777327]
```

Figure 12: The result is fairly close to the expected. You can clearly see that the middle path with the lowest cost (x + 2*x) is prioritized a lot less. As expected, the comfort of the few is sacrificed for the benefit of the many.

Path	Simulated Flow	Real Flow	Difference
x+1+x+1	0.01	0	0.01
1+2x	0.43	0.41	0.02
1+x+x+1	0.01	0	0.01
1+x+2x	0.04	0	0.04
x+x+1	0.23	0.23	0.00
x+2x	0.33	0.36	0.03

Table 2: Comparison of Simulated and Real Flow Values

Both approaches yield satisfying results when it comes to calculating the Optimal flow f^*.

10.2.3 Nash equilibrium (by the first approach - minimize $\phi(f)$)

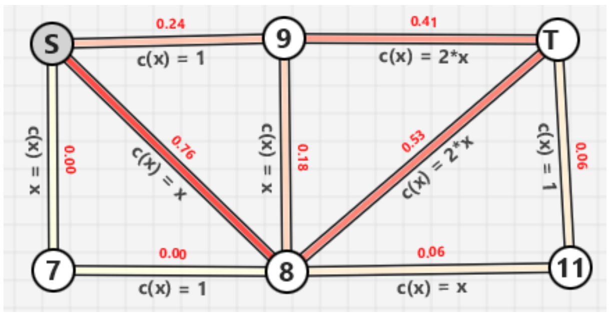
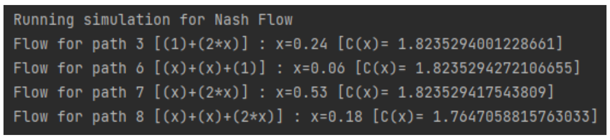


Figure 13: Nash flow $f^{\sim}$ as a result of minimizing the $\phi(f)$ function. Avg. travel time = 1.81. Close to the expected time of 1.84.



```
Running simulation for Nash Flow
Flow for path 3 [(1)+(2*x)] : x=0.24 [C(x)= 1.8235294001228661]
Flow for path 6 [(x)+(x)+(1)] : x=0.06 [C(x)= 1.8235294272106655]
Flow for path 7 [(x)+(2*x)] : x=0.53 [C(x)= 1.823529417543809]
Flow for path 8 [(x)+(x)+(2*x)] : x=0.18 [C(x)= 1.7647058815763033]
```

Figure 14: Resultant flow distribution of $f^{\sim}$ and costs alongside paths. It's fairly close to the expected solution (0.21, 0.05, 0.52, 0.21). Path 8 should be prioritized much more heavily as it's clearly a better strategy.

Path	Simulated Flow	Expected Flow	Difference
1+2x	0.24	0.21	0.03
x+x+1	0.06	0.05	0.01
x+2x	0.53	0.52	0.01
x+x+2x	0.18	0.21	-0.03

Table 3: Comparison of Simulated and Real Flow Values

10.2.4 Nash equilibrium (by the second approach - equalize costs c)

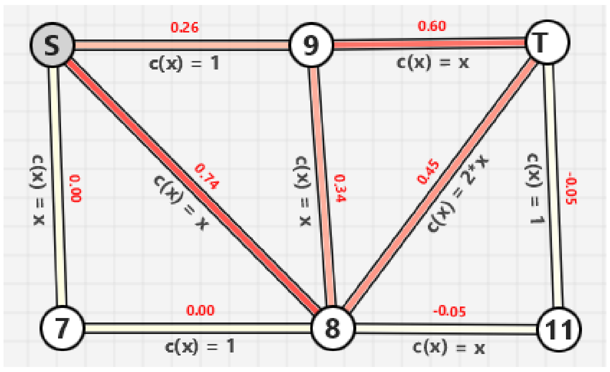
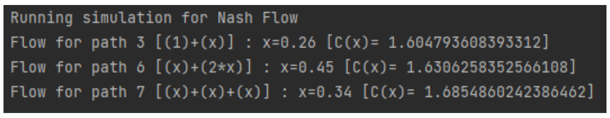


Figure 15: Nash flow $f^{\sim}$ as a result of minimizing dissimilarities of the cost function. Avg. travel time = 1.64. Much lower than it should be.



```
Running simulation for Nash Flow
Flow for path 3 [(1)+(x)] : x=0.26 [C(x)= 1.604793608393312]
Flow for path 6 [(x)+(2*x)] : x=0.45 [C(x)= 1.6306258352566108]
Flow for path 7 [(x)+(x)+(x)] : x=0.34 [C(x)= 1.6854860242386462]
```

Figure 16: The result is far from expected. Resultant flow distribution of $f^{\sim}$ and costs alongside paths. Negative flows appear. The penalty function wasn't sufficiently strong.

Path	Simulated Flow	Expected Flow	Difference
x+2x	0.45	0.52	-0.07
1+x	0.26	0.00	0.26
x+x+x	0.34	0.00	0.34
1+2x	0.00	0.21	-0.21
x+x+1	0.00	0.06	-0.06
x+x+2x	0.00	0.18	-0.18

Table 4: Comparison of Simulated and Real Flow Values. The result is surprising as the flow components on the edges are very similar to the correct case. The constraints weren't however enforced.

Only the first approach yields satisfying results. I suspect that when it came to the paths costs c and not paths marginal costs c^*, the penalty function was too weak to enforce the constraints. When it comes to Price of Anarchy the numerical approach gives result fairly close the expected one (1.10 vs 1.11). The error is acceptable.

10.3 Discrete case

10.3.1 Social optimum

The calculation of the **optimal** travel time is done by minimizing the cost of the flow $C(f)$ which is defined in the same way as in the continuous case [12](page 37):

$$C(f) = \sum_{e \in E} c_e(f_e) f_e$$

10.3.2 Nash equilibrium

The calculation of the **equilibrium** travel time is done by minimizing the discretized potential function [12](page 51):

$$\Phi_a(f) = \sum_{e \in E} \sum_{i=1}^{f_e} c_e(i).$$

10.4 Discrete case - Results

Calculating the function minimums in discrete case numerically requires integer optimization, for which the apache's library math3 is not suited. A makeshift approach was employed, recalculating point components of the objective function to be integers that sum up to the total traffic. The function for redistributing the double flow initially rounded the values up to the nearest integer and then adjusted the flow components to achieve a balanced sum.

However, this approach did not prove to be effective. The results are presented below:

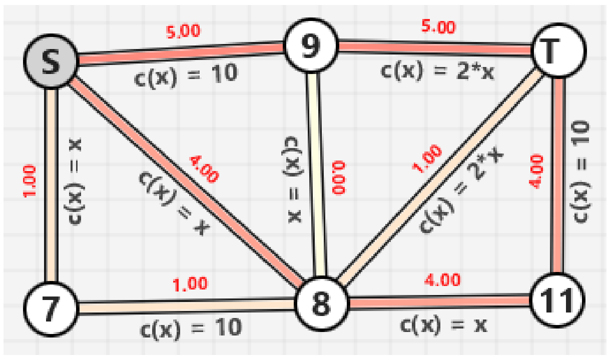


Figure 17: Optimal flow

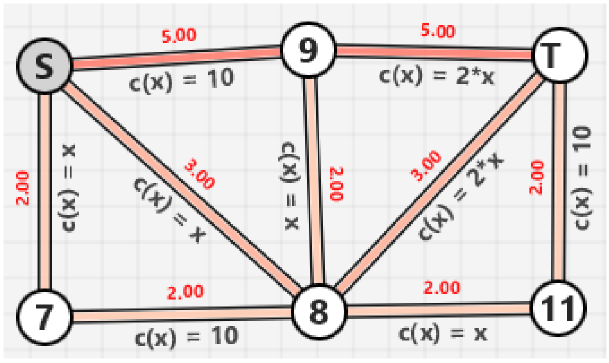
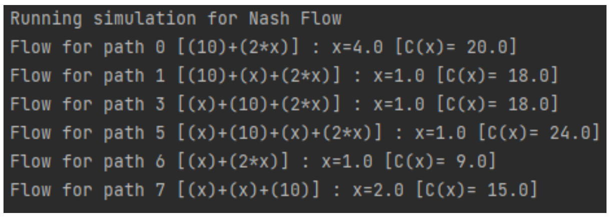


Figure 18: Supposed Nash flow for the discrete case



```
Running simulation for Nash Flow
Flow for path 0 [(10)+(2*x)] : x=4.0 [C(x)= 20.0]
Flow for path 1 [(10)+(x)+(2*x)] : x=1.0 [C(x)= 18.0]
Flow for path 3 [(x)+(10)+(2*x)] : x=1.0 [C(x)= 18.0]
Flow for path 5 [(x)+(10)+(x)+(2*x)] : x=1.0 [C(x)= 24.0]
Flow for path 6 [(x)+(2*x)] : x=1.0 [C(x)= 9.0]
Flow for path 7 [(x)+(x)+(10)] : x=2.0 [C(x)= 15.0]
```

Figure 19: The resultant flow is not an equilibrium. Individual players can adjust their strategies to reduce their travel time (for example pick path 6.).

The resultant price of Anarchy is 0.97 - clearly wrong as it cannot be below 1.

10.5 Remarks and Conclusions for Numerical approach

10.5.1 Constraints and Penalty functions

The numerical approach with optimizers is a rather elegant and generic solution, but it has some caveats.

The nature of the problem poses natural constraints on the function to be optimized. The flow has to be non-negative and sum up to the initial traffic. The first part of the constraint is rather simple, and supported by nearly every optimizer (in a form of simple bounds). The second constraint is also simple and can be expressed as a linear constraint in various libraries. As a reminder, they are of form:

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$$

But in our case the linear constraint would be very simple:

$$1 \cdot x_1 + 1 \cdot x_2 + \ldots + 1 \cdot x_{n-1} \leq T$$

Where the last component of the flow vector f_n is calculated as $f_n = T - \sum_{i=1}^{n-1} f_i$. Flow **f** has n dimensions, and was calculated from point **x** of dimension n–1.

However, not every optimizer (e.g. BOBYQA) allows the use of linear constraints. The problem is turned into the class of unconstrained multivariate optimization, where we minimize the objective function of form $F(x) = C(f) + P(f)$. The penalty function $P(X)$ has to be introduced. It has to add more to the reduced objective function than would be reduced by moving outside the constrained area..

$$P_1(\mathbf{f}) = \sum_i \max(0, -f_i^2) \qquad P_2(\mathbf{f}) = \max(0, (\sum_i f_i - T)^2)$$

$$P(\mathbf{f}) = P_1(\mathbf{f}) + P_2(\mathbf{f}) \tag{1}$$

Many different penalty functions (interior/exterior) have been tested. The following observations have been made:

- Best results when penalty was in the range of (1.1, 2.0) of the total objective function value.
- Barrier penalty function (infinite penalty when the constraint is broken) never worked, even with different interior initial points. [8]
- Standard quadratic loss function [7] is a good approach. The formula is provided above (1).
- Linear loss function with smaller coefficients works better when minimizing the dissimilarities (MSE), as the objective function has smaller values.
- The lower the penalty, the faster the convergence (the speed-up is noticed at the end).

The following observations have been made regarding the optimizers:

- BOBYQA Optimizer is convenient, due to easily calculable interpolation points $(2 \cdot (paths.size() - 1) + 1)$. The value must lie within the interval [n+2, (n+1)(n+2)/2]. It also supports simple bounds. [9]
- The Simplex Optimizer (I tried multiple methods, including the Nelder-Mead method) does not support simple bounds. The initial simplex size must be calculated in consideration of the objective and penalty function values; otherwise, the method fails to converge.
- The results were not significantly different. Instead of experimenting with different optimizers from math3, one should seek a library that offers better-ported versions of Matlab's solvers, ones that support constrained optimization or integer optimization.

The following are the notes based on the recommendations from the lecturer. They are put here as a guidance for further corrections and they may have been misinterpreted them slightly:

- One proposed method to ensure that the penalty function is sufficiently large would be by running the optimizer once to obtain any cost C, and scaling the penalty function $P(x)$ by the resultant C.
- Another interesting approach would be to resign from the penalty function entirely. This approach would use simple bounds [0, T] for x, interpret the point **x** as the difference between T and f_i and rewrite the flow **f** in terms of it.
- Since the functions to be optimized ($C(f), \phi(f)$) are differentiable and convex, the focus should be placed on optimizers that make use of derivatives.

11 Literature

References

[1] Kamil Benjelloun. *ROUTING GAMES*. Mémoire d'initiation à la recherche, L3 Cycle Pluridisciplinaire d'Etudes Supérieures, Université Paris Dauphine, Paris Sciences et Lettres, June 2019. `https://www.ceremade.dauphine.fr/~vigeral/Memoire2019Benjelloun.pdf`. Accessed on [25 Jan 2024].

[2] Vince Knight. *Game Theory for Cardiff University*. Chapter 17: Routing Games. Available online at: `https://vknight.org/Year_3_game_theory_course/Content/Chapter_17_Routing_games/`. Accessed on [25 Jan 2024].

[3] Vince Knight. *Game Theory for Cardiff University*. Chapter 18: Connection between optimal and Nash flows. Available online at: `https://vknight.org/Year_3_game_theory_course/Content/Chapter_18_Connection_between_optimal_and_nash_flows/`. Accessed on [25 Jan 2024].

[4] CS 6840: Algorithmic Game Theory. Lecture 5: February 3. Lecturer: Thodoris Lykouris. Scribe: Fikri Pitsuwan. Available online at: `https://www.cs.cornell.edu/courses/cs6840/2017sp/lecnotes/lec05.pdf`. Accessed on [25 Jan 2024].

[5] Tim Roughgarden. *CS364A: Algorithmic Game Theory*. Lecture #11: Selfish Routing and the Price of Anarchy. October 28, 2013. Available online at: `https://theory.stanford.edu/~tim/f13/l/l11.pdf`.

[6] Tim Roughgarden. *The Price of Anarchy is Independent of the Network Topology*. December 23, 2002. Available online at: `https://theory.stanford.edu/~tim/papers/indep.pdf`.

[7] Stanford SISL. *Penalty Functions - Stanford SISL*. [Online Resource]. Available at: `https://web.stanford.edu/group/sisl/k12/optimization/MO-unit5-pdfs/5.6penaltyfunctions.pdf`.

[8] Nesterov, Yurii. *Lectures on Convex Optimization (2nd ed.)*. Cham, Switzerland: Springer, 2018, p. 56. ISBN 978-3-319-91577-7.

[9] BOBYQAOptimizer. [Online Resource] Available at: `https://home.apache.org/~luc/commons-math-3.6-RC2-site/apidocs/org/apache/commons/math3/optimization/direct/BOBYQAOptimizer.html#BOBYQAOptimizer(int)`.

[10] backtracking algorithm `https://www.geeksforgeeks.org/introduction-to-backtracking-data-structure-and-algorithm-tutorials/`

[11] backtracking algorithm `https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/`

[12] Piotr Sankowski. *Network Games*. Uniwersytet Warszawski, Warszawa, 30 April 2014. Available online: `https://duch.mimuw.edu.pl/~sank/wordpress/wp-content/uploads/2014/05/lecture7.pdf`.