# Chapter 9: The Price of Anarchy

## Algorithmic Game Theory

PoA and PoS | Pigou's network | The 4/3 bound | Smoothness | PoA in the Braess network | Load balancing | Price of stability | Congestion tolls

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/agt/slides/Chapter_09_Price_of_Anarchy.html

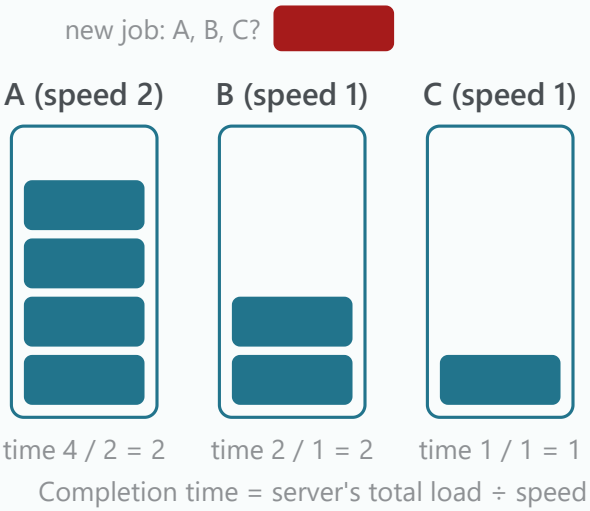
# Warm-up: The Load Balancing Game

### The rules

Jobs arrive one by one and each joins server A, B or C; a job's completion time is the server's total load divided by its speed.

### Think about it

- If every job minimizes its own completion time, is the system's total cost minimized too?
- How much worse than the best assignment can an equilibrium of selfish jobs be?



# Learning Outcomes

### Define

Define the Price of Anarchy (PoA) and the Price of Stability (PoS); show the difference with an example.

### Compute

Find equilibrium and optimal costs in the Pigou, Braess and load balancing examples and compute the ratio.

### Explain the bound

Explain the intuitive proof of PoA $\leq 4/3$ for linear latencies and why it is tight.

### Improve

Show how a congestion toll turns the optimum into an equilibrium in the two-route network and Pigou's network.

# The Price of Anarchy: The Bill for Selfishness

- The Price of Anarchy (PoA) measures the efficiency lost between the **worst equilibrium** that selfish behavior produces and the social optimum.
- In cost problems, the social optimum minimizes total cost.
- In a traffic network, total delay can rise while each user tries to cut their own travel time.
- Koutsoupias and Papadimitriou (1999) introduced the idea as the “coordination ratio”; Papadimitriou (2001) named it the “Price of Anarchy”: the cost of everyone deciding alone (anarchy) instead of a central “dictator”.

Example: in the two-route network the Wardrop equilibrium costs 75,000 and the system optimum 71,250 car·min (ratio 75,000 / 71,250 ≈ 1.053); in the Braess network the new road raised travel time from 65 to 80 min. The Price of Anarchy measures and bounds such gaps with **a single number**.

BASIC CONCEPTS

# Nash Equilibrium and Social Optimum

### Nash equilibrium (individual)

+ No player can improve their outcome by changing strategy unilaterally.
+ Reflects individual incentives.
+ There may be several, good and bad.

### Social optimum (system)

– Minimizes total cost or maximizes social welfare.
– Reflects the system's common goal.
– Usually not an equilibrium: someone wants to deviate.

Why do they differ? Individual rationality (everyone looks after their own payoff), externalities (costs imposed on others are ignored) and missing information. The Price of Anarchy measures the ratio between the two.

BASIC CONCEPTS

# Definitions: PoA and PoS

**Price of Anarchy**

$PoA = \max_{s \in NE} C(s) / C(OPT)$

worst equilibrium: a pessimistic guarantee

**Price of Stability**

$PoS = \min_{s \in NE} C(s) / C(OPT)$

best equilibrium: the optimistic scenario

NE: the game's Nash equilibria; C(s): total cost of state s; OPT: the state of minimum total cost (social optimum). PoA = cost of the worst equilibrium ÷ cost of the optimum; PoS = cost of the best equilibrium ÷ cost of the optimum.




- With C(OPT) > 0, always **1 ≤ PoS ≤ PoA**. With a unique equilibrium they are equal.
- In payoff (welfare) problems the ratio is flipped: SW(OPT) / SW(equilibrium) (SW: social welfare), so a larger value still means worse efficiency.

# Measuring and Reading Efficiency

- **Total cost**: the sum of all users' delay or resource-use costs.
- In cost problems, the PoA is the ratio of the worst equilibrium cost to the social optimum cost.
- A PoA of 1 means no efficiency loss.
- The larger the ratio, the more strategic behavior costs.

| PoA | Meaning | Example |
|---|---|---|
| 1 | no loss | Braess, before the new road |
| 4/3 | 33% extra cost | Pigou, the linear bound |
| $2 - 2/(m+1)$ | < 2 times | load balancing |
| unbounded | no guarantee | steep latency ($x^p$) |

Why the *worst* case? The PoA is a **guarantee** that holds without knowing which equilibrium players reach: “whatever happens, the loss is at most this much.” It is the game-theoretic counterpart of an algorithm's approximation ratio.

# Two Worst-Case Ratios: What Is Missing?

| Ratio | What blocks the optimum | Numerator / denominator |
|---|---|---|
| **Approximation ratio** | computational power: a problem not solvable in polynomial time (NP-hard) | solution found by a fast algorithm ÷ optimum |
| **Price of Anarchy** | coordination: no central authority, everyone pursues their own interest | worst equilibrium / optimum |

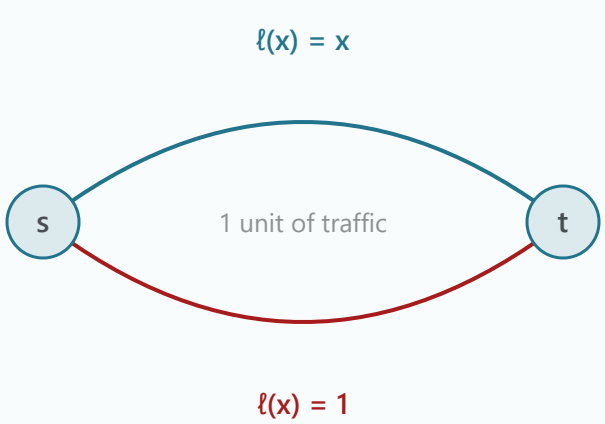
### Same problem, two losses

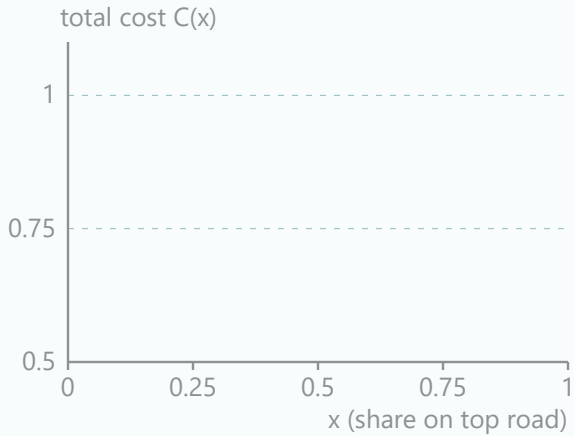
Makespan on m identical machines: Graham's greedy list algorithm is at most $2 - 1/m$ times worse; a pure equilibrium of selfish jobs at most $2 - 2/(m+1)$ times. For $m = 2$: 3/2 and 4/3.

### Limits of the analysis

Real players are not purely selfish: trust, social norms, reputation and repeated interaction can foster cooperation. So the PoA is a pessimistic guarantee, not a prediction.

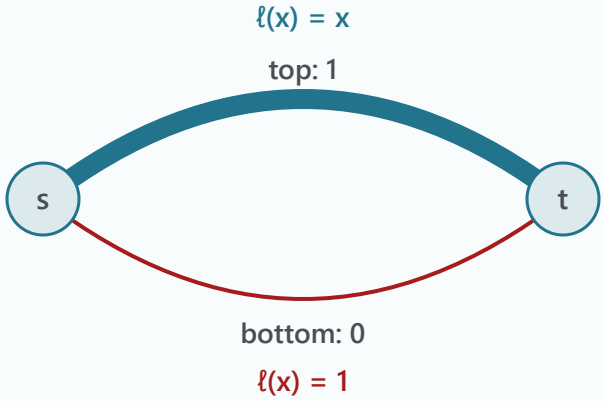
# Step by Step: The Price of Anarchy in Pigou's Network

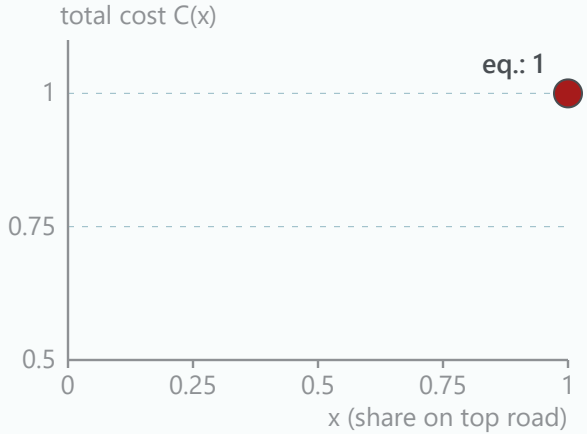



Setup (Pigou, 1920): 1 unit of traffic, many small drivers. Top road: delay x (the share on top); bottom road: always 1.

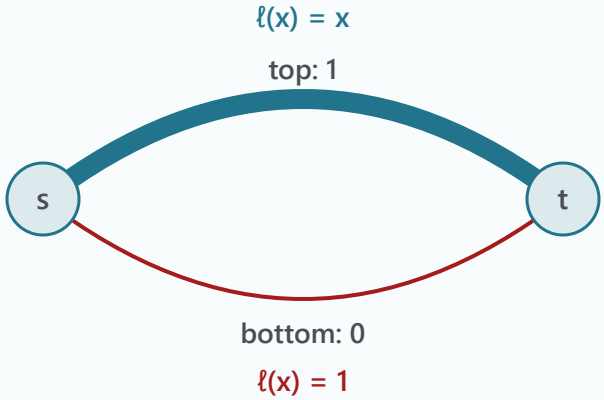
# Step by Step: The Price of Anarchy in Pigou's Network

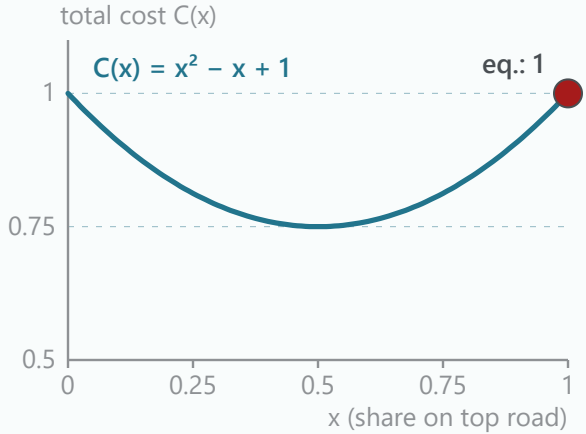



Equilibrium: the top road is never slower than the bottom ($x \le 1$) → everyone switches to the top. $x = 1$, everyone takes 1 min: **C(eq) = 1**.

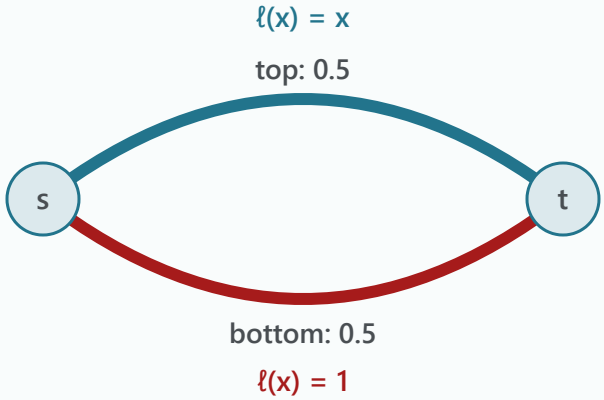
# Step by Step: The Price of Anarchy in Pigou's Network

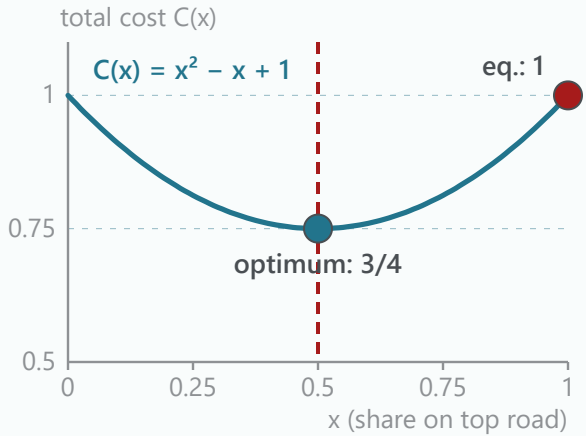



Total cost: share x on top takes x min, share 1 − x on the bottom takes 1 min → C(x) = x·x + (1 − x)·1 = $x^2$ − x + 1.

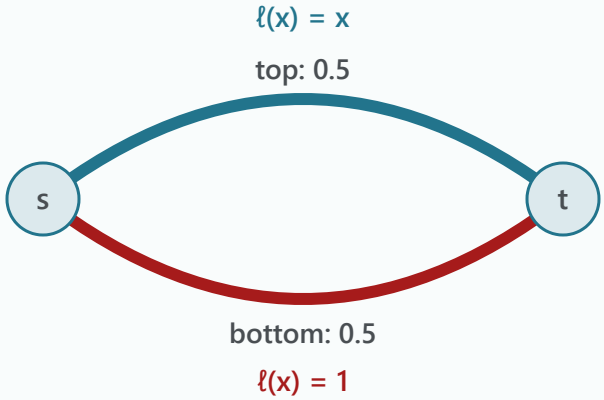
# Step by Step: The Price of Anarchy in Pigou's Network

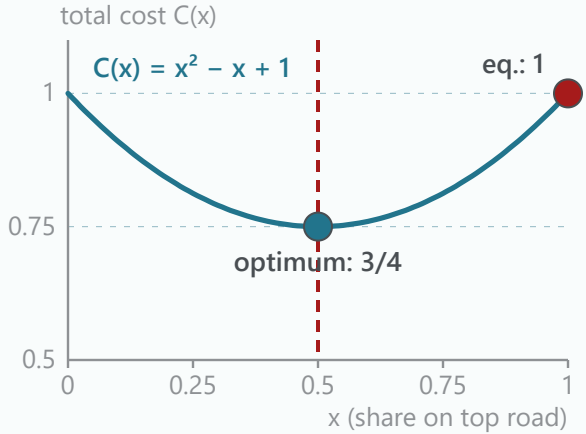



Minimum: $C'(x) = 2x - 1 = 0 \rightarrow \mathbf{x = 1/2}$. Half the traffic is "sacrificed" to the bottom road.

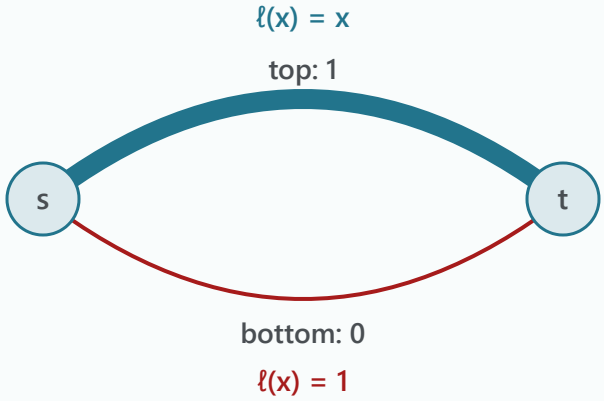
# Step by Step: The Price of Anarchy in Pigou's Network

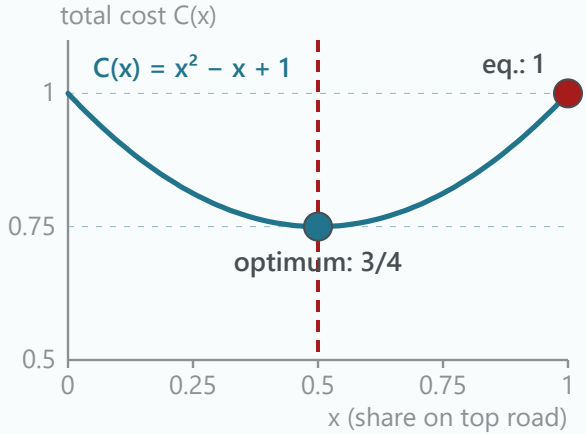



C(1/2) = 1/4 + 1/2 = **3/4**. Top drivers take 1/2 min, bottom drivers 1 min; the bottom drivers want to switch → the optimum is not an equilibrium.

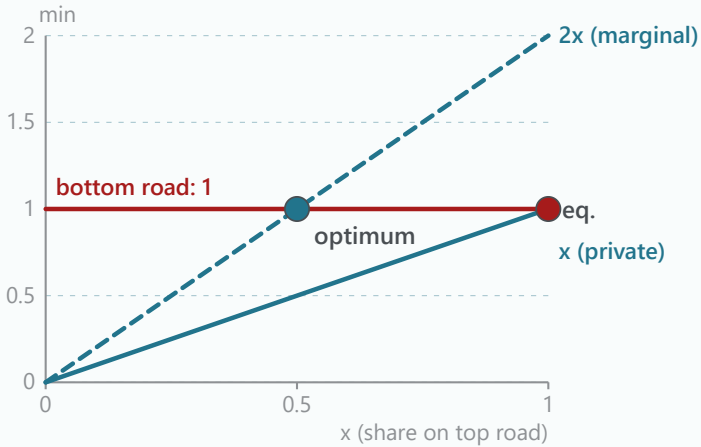
# Step by Step: The Price of Anarchy in Pigou's Network




**PoA = C(eq) / C(opt) = 1 / (3/4) = 4/3 ≈ 1.33**: selfish routing produces 33% more total delay.

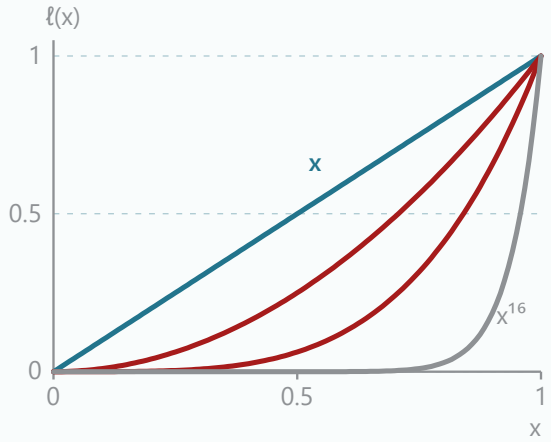
# Why the Difference? Private vs. Marginal Cost




- A driver looks at their **private** cost: compares x with 1 → x = 1.
- Society looks at **marginal** cost: the top road's total cost is x·x with derivative 2x; 2x = 1 → x = 1/2.
- The gap is x·ℓ′(x): the delay one driver imposes on the others (an externality).

General principle: at equilibrium the delays of used paths are equal. At the optimum, "my own delay + the delay I add for others" is equalized. With f cars on a road and delay ℓ(f), the added delay is f·ℓ′(f) (ℓ′: the derivative of the delay). This gap is the basis for congestion tolls.

# Steeper Latency, Larger PoA




| p (ℓ = $x^p$) | $x^{OPT}$ | C(OPT) | PoA |
|---|---|---|---|
| 1 | 0.5 | 0.75 | **1.333** |
| 2 | 0.577 | 0.615 | **1.626** |
| 4 | 0.669 | 0.465 | **2.151** |
| 8 | 0.76 | 0.325 | **3.081** |
| 16 | 0.838 | 0.212 | **4.727** |

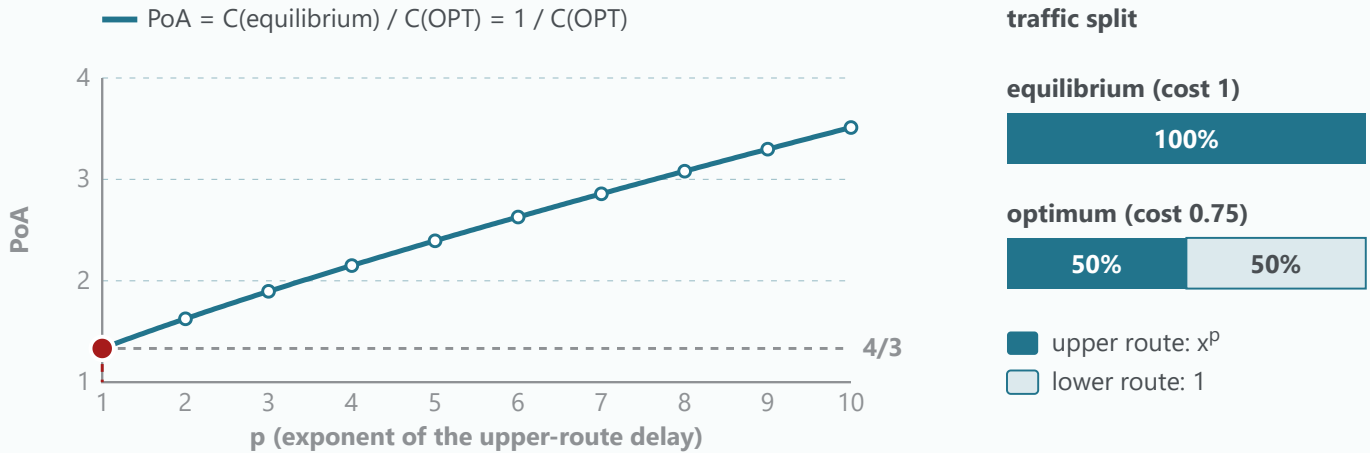
The equilibrium always sends everyone to the top road (cost 1). As latency gets steeper (larger p) the optimum improves and **the PoA grows slowly but without bound**: about 4.7 at p = 16. So 4/3 is not universal; it is specific to linear latency.

# Latency Exponent and PoA in Pigou's Network

Top road $x^p$, bottom road 1 · $C(x) = x^{p+1} + (1 - x)$ · $C'(x) = 0 \Rightarrow$ **$x^* = (p + 1)^{-1/p}$** · **PoA = 1 / C(x*)**

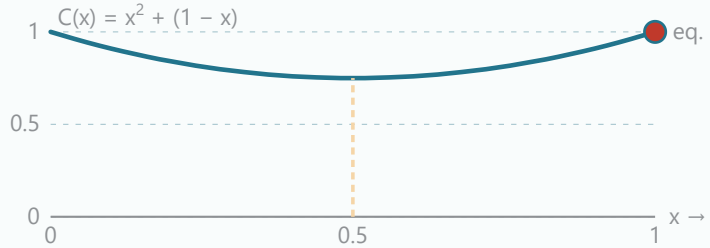
**p =** 1

$x^* = (p + 1)^{-1/p} =$ **0.5** | $C(OPT) = 1 - p(p + 1)^{-(p+1)/p} =$ **0.75** | C(equilibrium) = **1** | PoA = **1.333**




As the latency function gets steeper, the optimum improves, the equilibrium cost stays at 1 and the PoA grows without bound; with linear latency (p = 1), PoA = **4/3**.

# Code Trace: `pigou_poa(p)`




p = 1

$x^{OPT} = 0.5$

p = 1 (Pigou): $C(x) = x^2 + 1 - x$, $C'(x) = 2x - 1 = 0 \Rightarrow x^{OPT} = 2^{-1} = 0.5$.

Step 1 / 10

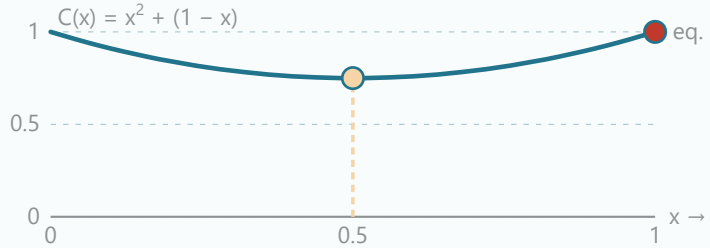
```
1  def pigou_poa(p):            # top x**p, bottom 1
▶     x = (p + 1) ** (-1 / p)   # where C'(x) = 0
3     opt = x ** (p + 1) + (1 - x)
4     return 1 / opt            # eq. cost is 1
5
▶  for p in [1, 2, 4, 8, 16]:
▶     print(p, round(pigou_poa(p), 3))
```

**VARIABLES**

p = **1** x = **0.5**

**OUTPUT**

# Code Trace: `pigou_poa(p)`




p = 1

$x^{OPT}$ = 0.5

C(OPT) = 0.75

PoA = 1 / C(OPT) = 1.333

C(OPT) = 0.25 + 0.5 = 0.75 ⇒ PoA = 4/3: Pigou's classic result.

Step 2 / 10

```
1 def pigou_poa(p):            # top x**p, bottom 1
2   x = (p + 1) ** (-1 / p)    # where C'(x) = 0
▶   opt = x ** (p + 1) + (1 - x)
▶   return 1 / opt             # eq. cost is 1
5
6 for p in [1, 2, 4, 8, 16]:
▶   print(p, round(pigou_poa(p), 3))
```

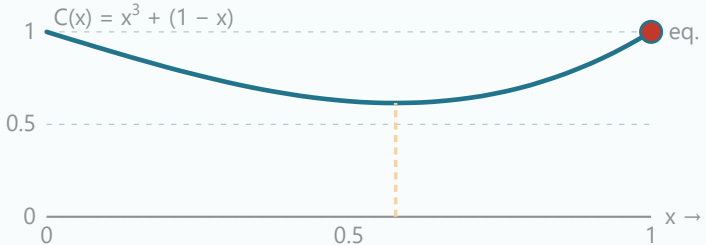
VARIABLES

`p = 1` `opt = 0.75`

OUTPUT

```
1 1.333
```

# Code Trace: `pigou_poa(p)`




p = 2

$x^{OPT} = 0.577$

p = 2: $C(x) = x \cdot x^2 + (1 - x) \cdot 1$, $C'(x) = 3x^2 - 1 = 0 \Rightarrow x^{OPT} = 3^{-1/2} \approx 0.577$.

Step 3 / 10

```
1 def pigou_poa(p):           # top x**p, bottom 1
▶   x = (p + 1) ** (-1 / p)   # where C'(x) = 0
3   opt = x ** (p + 1) + (1 - x)
4   return 1 / opt            # eq. cost is 1
5
▶ for p in [1, 2, 4, 8, 16]:
▶   print(p, round(pigou_poa(p), 3))
```

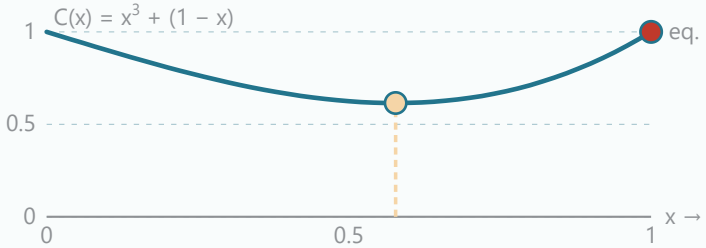
VARIABLES

p = 2 x = 0.5774

OUTPUT

```
1 1.333
```

# Code Trace: `pigou_poa(p)`




p = 2

$x^{OPT}$ = 0.577

C(OPT) = 0.615

PoA = 1 / C(OPT) = 1.626

The curve got steeper: the optimum fell to 0.615 while the equilibrium is still 1 ⇒ PoA 1.626.

Step 4 / 10

```
1 def pigou_poa(p):           # top x**p, bottom 1
2   x = (p + 1) ** (-1 / p)   # where C'(x) = 0
▶   opt = x ** (p + 1) + (1 - x)
▶   return 1 / opt            # eq. cost is 1
5
6 for p in [1, 2, 4, 8, 16]:
▶   print(p, round(pigou_poa(p), 3))
```

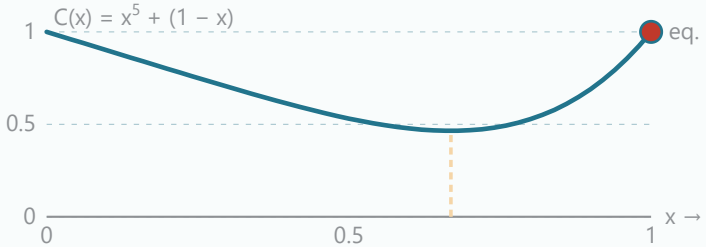
VARIABLES

p = 2 opt = 0.6151

OUTPUT

```
1 1.333
2 1.626
```

# Code Trace: `pigou_poa(p)`




p = 4

$x^{OPT} = 0.669$

p = 4: $C(x) = x \cdot x^4 + (1 - x) \cdot 1$, $C'(x) = 5x^4 - 1 = 0 \Rightarrow x^{OPT} = 5^{-1/4} \approx 0.669$.

Step 5 / 10

```
1 def pigou_poa(p):           # top x**p, bottom 1
▶   x = (p + 1) ** (-1 / p)   # where C'(x) = 0
3   opt = x ** (p + 1) + (1 - x)
4   return 1 / opt            # eq. cost is 1
5
▶ for p in [1, 2, 4, 8, 16]:
▶   print(p, round(pigou_poa(p), 3))
```

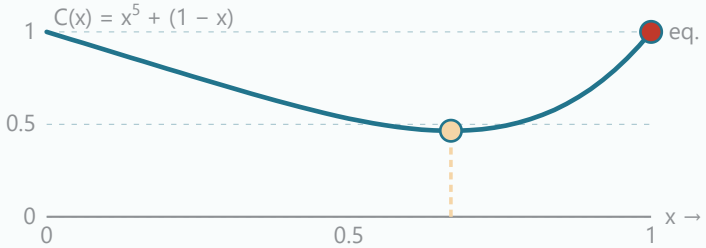
VARIABLES

p = 4 | x = 0.6687

OUTPUT

```
1 1.333
2 1.626
```

# Code Trace: `pigou_poa(p)`




p = 4: at the optimum 67% of the traffic is on top; PoA 2.151 > 2.

p = 4

$x^{OPT}$ = 0.669

C(OPT) = 0.465

PoA = 1 / C(OPT) = 2.151

Step 6 / 10

```
1 def pigou_poa(p):           # top x**p, bottom 1
2   x = (p + 1) ** (-1 / p)   # where C'(x) = 0
▶   opt = x ** (p + 1) + (1 - x)
▶   return 1 / opt            # eq. cost is 1
5
6 for p in [1, 2, 4, 8, 16]:
▶   print(p, round(pigou_poa(p), 3))
```

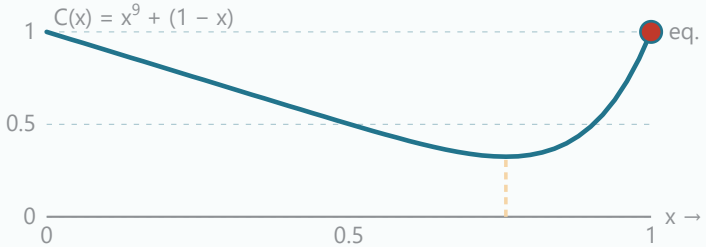
VARIABLES

p = 4 opt = 0.465

OUTPUT

```
1 1.333
2 1.626
4 2.151
```

# Code Trace: `pigou_poa(p)`




p = 8

$x^{OPT} = 0.76$

p = 8: $C(x) = x \cdot x^8 + (1 - x) \cdot 1$, $C'(x) = 9x^8 - 1 = 0 \Rightarrow x^{OPT} = 9^{-1/8} \approx 0.76$.

Step 7 / 10

```
1 def pigou_poa(p):           # top x**p, bottom 1
▶   x = (p + 1) ** (-1 / p)   # where C'(x) = 0
3   opt = x ** (p + 1) + (1 - x)
4   return 1 / opt            # eq. cost is 1
5
▶ for p in [1, 2, 4, 8, 16]:
▶   print(p, round(pigou_poa(p), 3))
```

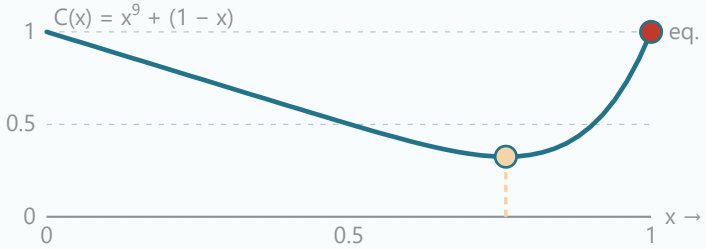
VARIABLES

p = 8 | x = 0.7598

OUTPUT

```
1 1.333
2 1.626
4 2.151
```

# Code Trace: `pigou_poa(p)`




p = 8

$x^{OPT}$ = 0.76

C(OPT) = 0.325

PoA = 1 / C(OPT) = 3.081

PoA 3.081: slow growth, but unbounded.

Step 8 / 10

```
1 def pigou_poa(p):           # top x**p, bottom 1
2   x = (p + 1) ** (-1 / p)   # where C'(x) = 0
▶   opt = x ** (p + 1) + (1 - x)
▶   return 1 / opt            # eq. cost is 1
5
6 for p in [1, 2, 4, 8, 16]:
▶   print(p, round(pigou_poa(p), 3))
```

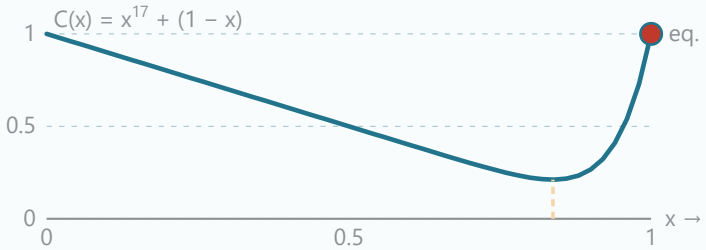
VARIABLES

`p = 8` `opt = 0.3246`

OUTPUT

```
1 1.333
2 1.626
4 2.151
8 3.081
```

# Code Trace: `pigou_poa(p)`




p = 16

$x^{OPT} = 0.838$

p = 16: $C(x) = x \cdot x^{16} + (1 - x) \cdot 1$, $C'(x) = 17x^{16} - 1 = 0 \Rightarrow x^{OPT} = 17^{-1/16} \approx 0.838$.

Step 9 / 10

```
1 def pigou_poa(p):           # top x**p, bottom 1
▶   x = (p + 1) ** (-1 / p)   # where C'(x) = 0
3   opt = x ** (p + 1) + (1 - x)
4   return 1 / opt            # eq. cost is 1
5
▶ for p in [1, 2, 4, 8, 16]:
▶   print(p, round(pigou_poa(p), 3))
```

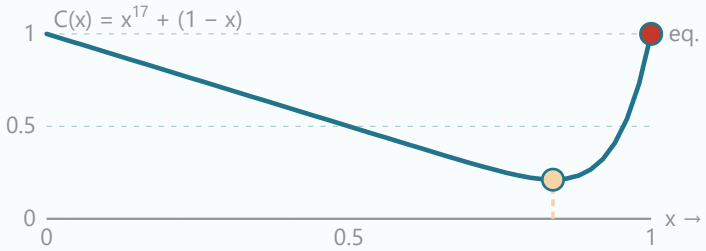
VARIABLES

p = 16 | x = 0.8377

OUTPUT

```
1 1.333
2 1.626
4 2.151
8 3.081
```

# Code Trace: `pigou_poa(p)`




p = 16

$x^{OPT}$ = 0.838

C(OPT) = 0.212

PoA = 1 / C(OPT) = 4.727

p = 16: PoA 4.727. The 4/3 guarantee holds only for linear latency.

Step 10 / 10

```
1 def pigou_poa(p):           # top x**p, bottom 1
2   x = (p + 1) ** (-1 / p)   # where C'(x) = 0
▶   opt = x ** (p + 1) + (1 - x)
▶   return 1 / opt            # eq. cost is 1
5
6 for p in [1, 2, 4, 8, 16]:
▶   print(p, round(pigou_poa(p), 3))
```

VARIABLES

p = 16 opt = 0.2116

OUTPUT

```
1 1.333
2 1.626
4 2.151
8 3.081
16 4.727
```

# The 4/3 Upper Bound for Linear Latency

**Theorem** (Roughgarden–Tardos, 2002). If latencies are linear ($\ell_e(x) = a_e x + b_e$, $a, b \geq 0$), then in *every* network with *every* demand: $C(\text{eq}) \leq \mathbf{4/3} \cdot C(\text{opt})$.

# The 4/3 Upper Bound for Linear Latency

**Theorem** (Roughgarden–Tardos, 2002). If latencies are linear ($\ell_e(x) = a_e x + b_e$, $a, b \geq 0$), then in *every* network with *every* demand: $C(\text{eq}) \leq \mathbf{4/3} \cdot C(\text{opt})$.

**Notation.** $f_e$: equilibrium traffic on edge $e$; $f_e^{OPT}$: optimal traffic. Total delay sums "traffic × delay" over edges: $C(f) = \Sigma\, \ell_e(f_e) \cdot f_e$.

# The 4/3 Upper Bound for Linear Latency

**Theorem** (Roughgarden–Tardos, 2002). If latencies are linear ($\ell_e(x) = a_e x + b_e$, $a, b \geq 0$), then in *every* network with *every* demand: $C(\text{eq}) \leq \mathbf{4/3} \cdot C(\text{opt})$.

**Notation.** $f_e$: equilibrium traffic on edge $e$; $f_e^{OPT}$: optimal traffic. Total delay sums "traffic × delay" over edges: $C(f) = \Sigma\, \ell_e(f_e) \cdot f_e$.

**1. Equilibrium condition.** At equilibrium everyone uses a shortest path. Freeze the delays and move drivers to their optimal paths: nobody finds a shorter one: $\Sigma\, \ell_e(f_e) \cdot f_e \leq \Sigma\, \ell_e(f_e) \cdot f_e^{OPT}$.

# The 4/3 Upper Bound for Linear Latency

**Theorem** (Roughgarden–Tardos, 2002). If latencies are linear ($\ell_e(x) = a_e x + b_e$, $a, b \geq 0$), then in *every* network with *every* demand: $C(\text{eq}) \leq \mathbf{4/3} \cdot C(\text{opt})$.

**Notation.** $f_e$: equilibrium traffic on edge $e$; $f_e^{OPT}$: optimal traffic. Total delay sums "traffic × delay" over edges: $C(f) = \Sigma\, \ell_e(f_e) \cdot f_e$.

**1. Equilibrium condition.** At equilibrium everyone uses a shortest path. Freeze the delays and move drivers to their optimal paths: nobody finds a shorter one: $\Sigma\, \ell_e(f_e) \cdot f_e \leq \Sigma\, \ell_e(f_e) \cdot f_e^{OPT}$.

The left side is the equilibrium's real cost $C(f)$. The right side is a thought experiment: roads as slow as at equilibrium, but traffic spread as in the optimum. It remains to bound this right side by $C(f^{OPT})$.

# The 4/3 Bound: Edge Inequality and Summation

**2. Edge inequality.** For each edge: $\ell_e(f_e)\cdot f_e^{OPT} \le \ell_e(f_e^{OPT})\cdot f_e^{OPT} + \tfrac{1}{4}\cdot\ell_e(f_e)\cdot f_e$. The trial cost exceeds the optimum's cost by at most ¼ of the equilibrium's.

# The 4/3 Bound: Edge Inequality and Summation

**2. Edge inequality.** For each edge: $\ell_e(f_e)\cdot f_e^{OPT} \le \ell_e(f_e^{OPT})\cdot f_e^{OPT} + \tfrac{1}{4}\cdot\ell_e(f_e)\cdot f_e$. The trial cost exceeds the optimum's cost by at most ¼ of the equilibrium's.

**Why is it true?** Put $\ell = ax + b$. The b terms: $b\cdot f^{OPT} \le b\cdot f^{OPT} + \tfrac{1}{4}\cdot b\cdot f$. The a terms need $a\cdot f\cdot f^{OPT} \le a\cdot (f^{OPT})^2 + a\cdot f^2/4$, i.e. $a\cdot (f^{OPT} - f/2)^2 \ge 0$: a square.

# The 4/3 Bound: Edge Inequality and Summation

**2. Edge inequality.** For each edge: $\ell_e(f_e)\cdot f_e^{OPT} \le \ell_e(f_e^{OPT})\cdot f_e^{OPT} + \tfrac{1}{4}\cdot\ell_e(f_e)\cdot f_e$. The trial cost exceeds the optimum's cost by at most ¼ of the equilibrium's.

**Why is it true?** Put $\ell = ax + b$. The b terms: $b\cdot f^{OPT} \le b\cdot f^{OPT} + \tfrac{1}{4}\cdot b\cdot f$. The a terms need $a\cdot f\cdot f^{OPT} \le a\cdot(f^{OPT})^2 + a\cdot f^2/4$, i.e. $a\cdot(f^{OPT} - f/2)^2 \ge 0$: a square.

**Example** (Pigou, the $\ell = x$ road): $f = 1$, $f^{OPT} = \tfrac{1}{2}$. Left: $1\cdot\tfrac{1}{2} = \tfrac{1}{2}$. Right: $\tfrac{1}{2}\cdot\tfrac{1}{2} + \tfrac{1}{4}\cdot 1\cdot 1 = \tfrac{1}{2}$. Equal.

# The 4/3 Bound: Edge Inequality and Summation

**2. Edge inequality.** For each edge: $\ell_e(f_e)\cdot f_e^{OPT} \le \ell_e(f_e^{OPT})\cdot f_e^{OPT} + \tfrac{1}{4}\cdot\ell_e(f_e)\cdot f_e$. The trial cost exceeds the optimum's cost by at most ¼ of the equilibrium's.

**Why is it true?** Put $\ell = ax + b$. The b terms: $b\cdot f^{OPT} \le b\cdot f^{OPT} + \tfrac{1}{4}\cdot b\cdot f$. The a terms need $a\cdot f\cdot f^{OPT} \le a\cdot(f^{OPT})^2 + a\cdot f^2/4$, i.e. $a\cdot(f^{OPT} - f/2)^2 \ge 0$: a square.

**Example** (Pigou, the $\ell = x$ road): $f = 1$, $f^{OPT} = \tfrac{1}{2}$. Left: $1\cdot\tfrac{1}{2} = \tfrac{1}{2}$. Right: $\tfrac{1}{2}\cdot\tfrac{1}{2} + \tfrac{1}{4}\cdot 1\cdot 1 = \tfrac{1}{2}$. Equal.

**3. Sum.** Sum step 2 over edges, use step 1: $C(f) \le C(f^{OPT}) + \tfrac{1}{4}\cdot C(f)$. Move $\tfrac{1}{4}\cdot C(f)$ left: $\tfrac{3}{4}\cdot C(f) \le C(f^{OPT})$ → **$C(f) / C(f^{OPT}) \le 4/3$**.

# The 4/3 Bound: Edge Inequality and Summation

**2. Edge inequality.** For each edge: $\ell_e(f_e)\cdot f_e^{OPT} \le \ell_e(f_e^{OPT})\cdot f_e^{OPT} + \tfrac{1}{4}\cdot\ell_e(f_e)\cdot f_e$. The trial cost exceeds the optimum's cost by at most ¼ of the equilibrium's.

**Why is it true?** Put $\ell = ax + b$. The b terms: $b\cdot f^{OPT} \le b\cdot f^{OPT} + \tfrac{1}{4}\cdot b\cdot f$. The a terms need $a\cdot f\cdot f^{OPT} \le a\cdot(f^{OPT})^2 + a\cdot f^2/4$, i.e. $a\cdot(f^{OPT} - f/2)^2 \ge 0$: a square.

**Example** (Pigou, the $\ell = x$ road): $f = 1$, $f^{OPT} = \tfrac{1}{2}$. Left: $1\cdot\tfrac{1}{2} = \tfrac{1}{2}$. Right: $\tfrac{1}{2}\cdot\tfrac{1}{2} + \tfrac{1}{4}\cdot 1\cdot 1 = \tfrac{1}{2}$. Equal.

**3. Sum.** Sum step 2 over edges, use step 1: $C(f) \le C(f^{OPT}) + \tfrac{1}{4}\cdot C(f)$. Move $\tfrac{1}{4}\cdot C(f)$ left: $\tfrac{3}{4}\cdot C(f) \le C(f^{OPT})$ → **$C(f) / C(f^{OPT}) \le 4/3$**.

Pigou attains it: the bound is tight. With constant latencies in the class, the worst case is always a two-link network like Pigou's (Roughgarden, 2003): topology does not raise the PoA.

# Testing the 4/3 Bound

| Example | Equilibrium | Optimum | Observed ratio | Upper bound |
|---|---|---|---|---|
| Pigou (x and 1) | 1 | 3/4 | **4/3** (tight) | 4/3 |
| Braess (4,000 cars) | 320,000 | 258,750 | 80 / 64.69 ≈ 1.237 | 4/3 |
| Braess normalized (x, 1, 1, x, 0) | 2 | 3/2 | **4/3** (tight) | 4/3 |
| Two routes, 3,000 cars | 75,000 | 71,250 | 1.053 | 4/3 |
| 20,000 random parallel linear networks | — | — | max 1.33 | 4/3 |
| Load Balancing Game, Level 2 | 13 | 12.5 | 1.04 | 5/2 (atomic) |

An upper bound is the **worst case** over a class (e.g. all linear-latency networks); single instances usually do better. In 20,000 random networks (2–4 links, random a, b, demand) the ratio never exceeded 4/3 and neared it in Pigou-like cases.

# One Recipe: Smoothness

**Thought experiment.** Let s be a state (each player has a strategy) and $s^{OPT}$ the optimum. What would player i pay if *only* i switched to their $s^{OPT}$ strategy while the others stay in s? This is i's "trial cost".

# One Recipe: Smoothness

**Thought experiment.** Let s be a state (each player has a strategy) and $s^{OPT}$ the optimum. What would player i pay if *only* i switched to their $s^{OPT}$ strategy while the others stay in s? This is i's "trial cost".

**Notation.** $s_{-i}$: the strategies of players other than i. $C_i(s_i^{OPT}, s_{-i})$: i's trial cost. $C(s) = \Sigma_i C_i(s)$: total cost.

# One Recipe: Smoothness

**Thought experiment.** Let s be a state (each player has a strategy) and $s^{OPT}$ the optimum. What would player i pay if *only* i switched to their $s^{OPT}$ strategy while the others stay in s? This is i's "trial cost".

**Notation.** $s_{-i}$: the strategies of players other than i. $C_i(s_i^{OPT}, s_{-i})$: i's trial cost. $C(s) = \Sigma_i C_i(s)$: total cost.

**Definition** (Roughgarden, 2009). Fix constants $\lambda > 0$ and $0 \le \mu < 1$. A game is **(λ, μ)-smooth** if $\Sigma_i C_i(s_i^{OPT}, s_{-i}) \le \lambda \cdot C(s^{OPT}) + \mu \cdot C(s)$ for every s and $s^{OPT}$.

# One Recipe: Smoothness

**Thought experiment.** Let s be a state (each player has a strategy) and $s^{OPT}$ the optimum. What would player i pay if *only* i switched to their $s^{OPT}$ strategy while the others stay in s? This is i's "trial cost".

**Notation.** $s_{-i}$: the strategies of players other than i. $C_i(s_i^{OPT}, s_{-i})$: i's trial cost. $C(s) = \Sigma_i C_i(s)$: total cost.

**Definition** (Roughgarden, 2009). Fix constants $\lambda > 0$ and $0 \le \mu < 1$. A game is **$(\lambda, \mu)$-smooth** if $\Sigma_i C_i(s_i^{OPT}, s_{-i}) \le \lambda \cdot C(s^{OPT}) + \mu \cdot C(s)$ for every s and $s^{OPT}$.

In words: the trial costs add up to at most λ times the optimum's cost plus μ times the cost of s. The smaller λ and μ, the better the resulting bound.

# From Smoothness to a PoA Bound: Four Steps

**1. Equilibrium.** $s$ is a Nash equilibrium: nobody gains by deviating alone. For every $i$, $C_i(s) \le C_i(s_i^{OPT}, s_{-i})$.

# From Smoothness to a PoA Bound: Four Steps

**1. Equilibrium.** $s$ is a Nash equilibrium: nobody gains by deviating alone. For every $i$, $C_i(s) \le C_i(s_i^{OPT}, s_{-i})$.

**2. Sum.** Add these inequalities over all players: $C(s) \le \Sigma_i\, C_i(s_i^{OPT}, s_{-i})$.

# From Smoothness to a PoA Bound: Four Steps

**1. Equilibrium.** $s$ is a Nash equilibrium: nobody gains by deviating alone. For every $i$, $C_i(s) \le C_i(s_i^{OPT}, s_{-i})$.

**2. Sum.** Add these inequalities over all players: $C(s) \le \sum_i C_i(s_i^{OPT}, s_{-i})$.

**3. Apply smoothness.** The right side is at most $\lambda \cdot C(s^{OPT}) + \mu \cdot C(s)$, so $C(s) \le \lambda \cdot C(s^{OPT}) + \mu \cdot C(s)$.

# From Smoothness to a PoA Bound: Four Steps

**1. Equilibrium.** $s$ is a Nash equilibrium: nobody gains by deviating alone. For every $i$, $C_i(s) \le C_i(s_i^{OPT}, s_{-i})$.

**2. Sum.** Add these inequalities over all players: $C(s) \le \Sigma_i\, C_i(s_i^{OPT}, s_{-i})$.

**3. Apply smoothness.** The right side is at most $\lambda \cdot C(s^{OPT}) + \mu \cdot C(s)$, so $C(s) \le \lambda \cdot C(s^{OPT}) + \mu \cdot C(s)$.

**4. Rearrange.** Move $\mu \cdot C(s)$ left: $(1 - \mu) \cdot C(s) \le \lambda \cdot C(s^{OPT})$. Divide both sides by $(1 - \mu) \cdot C(s^{OPT})$: $C(s) / C(s^{OPT}) \le \lambda / (1 - \mu)$. Since $s$ was any equilibrium, **PoA ≤ λ / (1 − μ)**.

# From Smoothness to a PoA Bound: Four Steps

**1. Equilibrium.** $s$ is a Nash equilibrium: nobody gains by deviating alone. For every $i$, $C_i(s) \le C_i(s_i^{OPT}, s_{-i})$.

**2. Sum.** Add these inequalities over all players: $C(s) \le \Sigma_i\, C_i(s_i^{OPT}, s_{-i})$.

**3. Apply smoothness.** The right side is at most $\lambda \cdot C(s^{OPT}) + \mu \cdot C(s)$, so $C(s) \le \lambda \cdot C(s^{OPT}) + \mu \cdot C(s)$.

**4. Rearrange.** Move $\mu \cdot C(s)$ left: $(1 - \mu) \cdot C(s) \le \lambda \cdot C(s^{OPT})$. Divide both sides by $(1 - \mu) \cdot C(s^{OPT})$: $C(s) / C(s^{OPT}) \le \lambda / (1 - \mu)$. Since $s$ was any equilibrium, **PoA $\le \lambda / (1 - \mu)$**.

**Familiar cases.** The edge inequality in the 4/3 proof means $\lambda = 1$, $\mu = ¼$: $1 / (1 - ¼) =$ **4/3**. In atomic (indivisible-player) linear congestion games $\lambda = 5/3$, $\mu = 1/3$: $(5/3) / (2/3) =$ **5/2**.

# From Smoothness to a PoA Bound: Four Steps

**1. Equilibrium.** $s$ is a Nash equilibrium: nobody gains by deviating alone. For every $i$, $C_i(s) \le C_i(s_i^{OPT}, s_{-i})$.

**2. Sum.** Add these inequalities over all players: $C(s) \le \Sigma_i\, C_i(s_i^{OPT}, s_{-i})$.

**3. Apply smoothness.** The right side is at most $\lambda \cdot C(s^{OPT}) + \mu \cdot C(s)$, so $C(s) \le \lambda \cdot C(s^{OPT}) + \mu \cdot C(s)$.

**4. Rearrange.** Move $\mu \cdot C(s)$ left: $(1 - \mu) \cdot C(s) \le \lambda \cdot C(s^{OPT})$. Divide both sides by $(1 - \mu) \cdot C(s^{OPT})$: $C(s) / C(s^{OPT}) \le \lambda / (1 - \mu)$. Since $s$ was any equilibrium, **PoA ≤ λ / (1 − μ)**.

**Familiar cases.** The edge inequality in the 4/3 proof means $\lambda = 1$, $\mu = ¼$: $1 / (1 - ¼) =$ **4/3**. In atomic (indivisible-player) linear congestion games $\lambda = 5/3$, $\mu = 1/3$: $(5/3) / (2/3) =$ **5/2**.

**Why does it matter?** The proof uses only “nobody deviates alone”, so the same bound holds for weaker equilibrium notions too (e.g. mixed equilibria): the **robust PoA**.

# Atomic Congestion Games: PoA ≤ 5/2

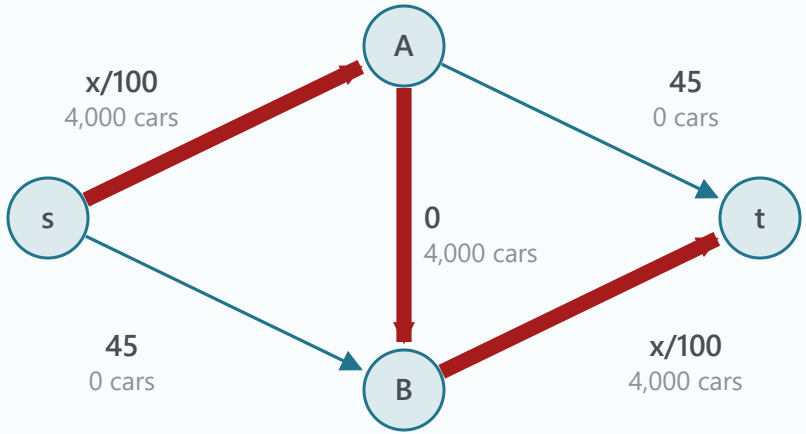
| y ($s^{OPT}$) \ z (equilibrium load) | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | **0** | 1 | 4 | 9 | 16 | 25 |
| 1 | 2 | **0** | **0** | 2 | 6 | 12 |
| 2 | 14 | 9 | 6 | 5 | 6 | 9 |
| 3 | 36 | 28 | 22 | 18 | 16 | 16 |
| 4 | 68 | 57 | 48 | 41 | 36 | 33 |
| 5 | 110 | 96 | 84 | 74 | 66 | 60 |

Cell: 3 × (right − left), avoiding fractions. No negative cells: it holds (checked for y, z ≤ 60); orange 0s mark equality.

- Each player is one **unit** of load; edge cost $c_e(x) = a_e x + b_e$.
- $z_e$, $y_e$: players on e at equilibrium and at the optimum. A lone switcher to their optimal path meets at most $z_e + 1$ players on e.
- Key inequality (integers): $\mathbf{y(z + 1) \le 5/3 \cdot y^2 + 1/3 \cdot z^2}$. With c(x) = x: left is the trial cost; $y^2$, $z^2$ the optimal and equilibrium costs.
- ⇒ (5/3, 1/3)-smooth ⇒ PoA ≤ (5/3)/(2/3) = **5/2**; the bound is tight (Awerbuch–Azar–Epstein; Christodoulou–Koutsoupias, 2005).

Splittable traffic gives 4/3, indivisible players 5/2: the delay a player adds itself (the "+1") enlarges the loss.

# Step by Step: PoA in the Braess Network




Equilibrium with the new road

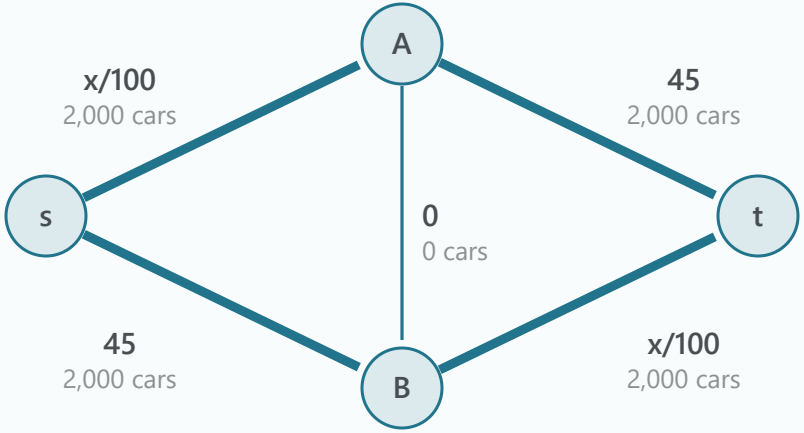
all s→A→B→t: 80 min

C(eq) = 4,000 · 80

= 320,000 car·min

Once A→B opens everyone zigzags, 80 min each. Total 320,000 car·min.

# Step by Step: PoA in the Braess Network




Before the new road: 65 min

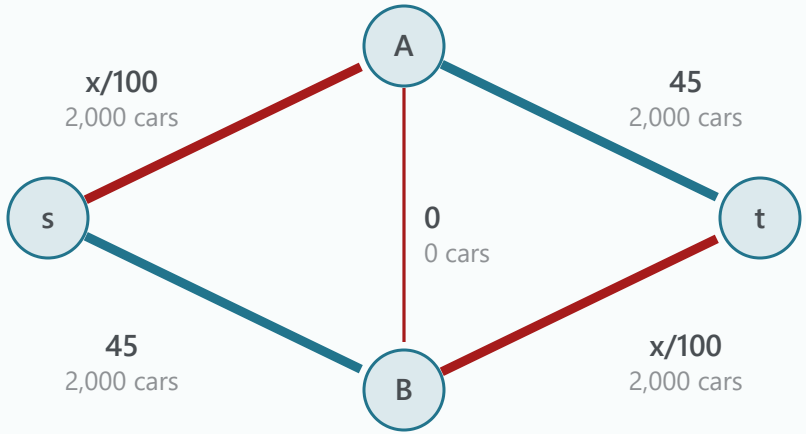
80 / 65 ≈ 1.231

This is the “Braess loss”;

not the PoA: other network

A common comparison is 80/65 ≈ 1.23. But the PoA's denominator is the optimum of **the same network**; can the new network do better than 65?

# Step by Step: PoA in the Braess Network




Marginal costs at z = 0:

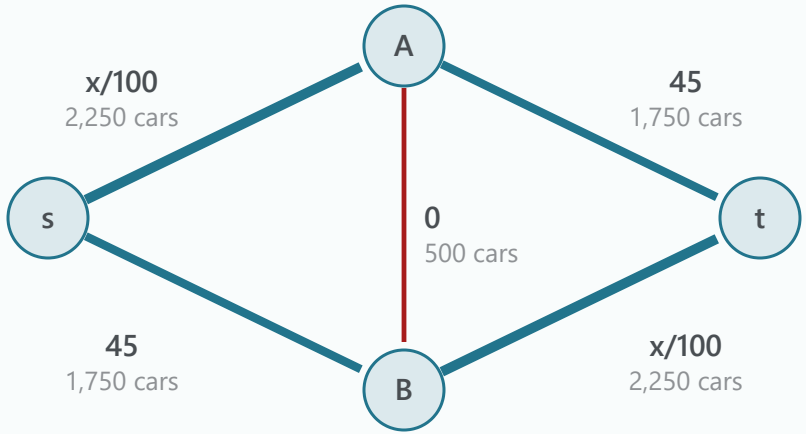
zigzag: 2·20 + 2·20 = 80

old path: 2·20 + 45 = 85

→ some zigzag helps!

On s→A with 2,000 cars the delay is 20 min; marginal cost $\ell + x \cdot \ell' = 20 + 2{,}000 \cdot (1/100) = 40$ min. The zigzag uses two such edges: 40 + 0 + 40 = 80; an old path 40 + 45 = 85. 80 < 85: even a planner sends some cars on the zigzag.

# Step by Step: PoA in the Braess Network




Symmetric: a cars per old path

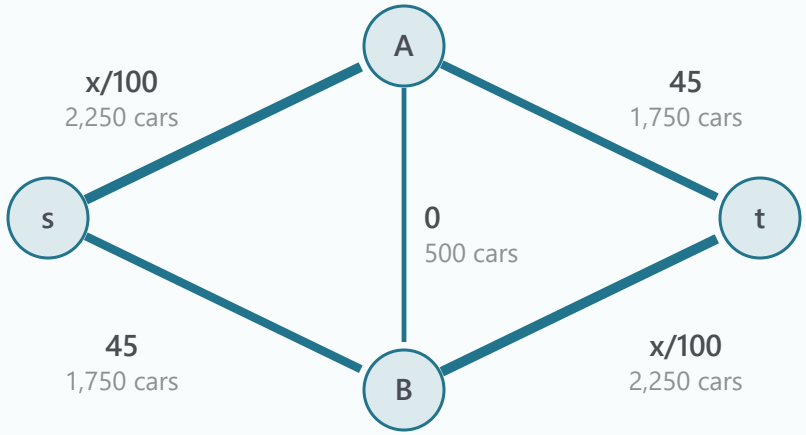
$C(a) = 2(4000 - a)^2/100 + 90a$

$C' = 0 \rightarrow a = 1{,}750,\ z = 500$

$C(\text{opt}) = 258{,}750$

Optimum: 1,750 on each old path, 500 on the zigzag (marginal costs equal at 90). Average ≈ 64.69 min; the script searched all splits.

# Step by Step: PoA in the Braess Network




PoA = 320,000 / 258,750

≈ 1.237

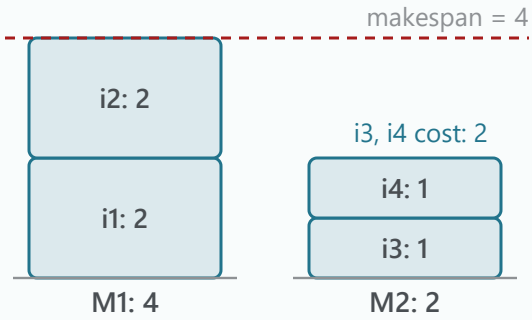
≤ 4/3 ✓

normalized: exactly 4/3

**PoA = 320,000 / 258,750 ≈ 1.237** ≤ 4/3. In the classic normalized Braess network (x, 1, 1, x, 0) the ratio is exactly 2 / (3/2) = 4/3.

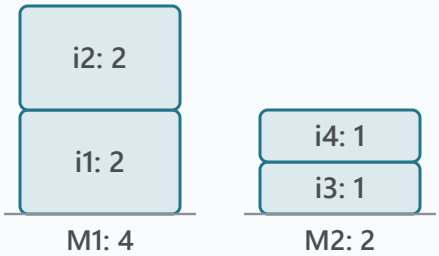
# The Load Balancing Game

- n jobs (players), m identical machines; job j has size $w_j$.
- Each job picks a machine; its **cost** is that machine's total load (its completion time).
- Social cost: the **makespan** (maximum load), i.e. when all jobs are done.
- Applications: server clusters, cloud task placement, processor cores. Koutsoupias–Papadimitriou first defined the PoA in this model (1999).




This game always has a pure Nash equilibrium: when a job moves to a less loaded machine, the load list sorted in decreasing order gets lexicographically “smaller” (a smaller number at the first difference). This cannot go on forever; the process stops at a Nash equilibrium.

# Step by Step: PoA in the Load Balancing Game




4 jobs: sizes 2, 2, 1, 1

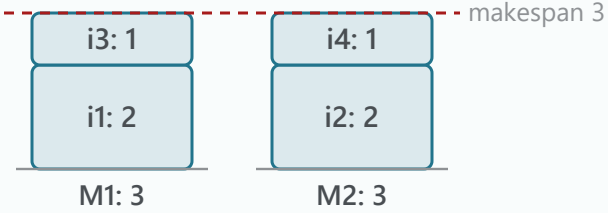
2 identical machines

Job cost: its machine's load

Social cost: makespan

The load balancing game: each job (player) picks a machine and wants to minimize its completion time, i.e. its machine's total load.

# Step by Step: PoA in the Load Balancing Game



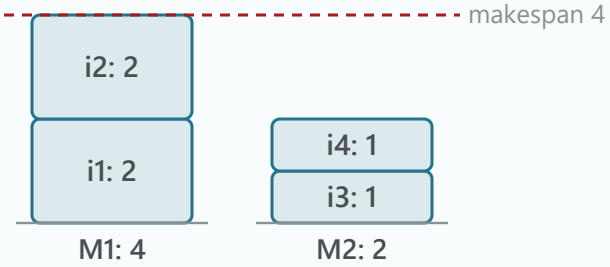

Optimum: {2, 1} | {2, 1}

loads 3 and 3

makespan = 3

Social optimum: total work 6 on two machines → best possible 3 and 3. Makespan 3.

# Step by Step: PoA in the Load Balancing Game



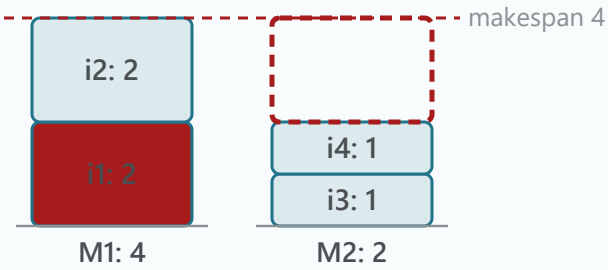

**Assignment: {2, 2} | {1, 1}**

**loads 4 and 2**

**makespan = 4**

Another assignment: both big jobs on M1 (load 4), both small jobs on M2 (load 2). Can this be an equilibrium?

# Step by Step: PoA in the Load Balancing Game




If i1 (2) moves to M2:

2 + 2 = 4, now 4

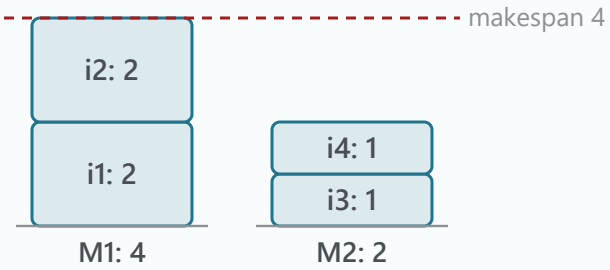
→ no gain

If i3 (1) moves to M1: 4 + 1 = 5 > 2

Deviation check: a big job moving to M2 still waits 4; a small job moving to M1 waits 5.

**Nobody gains → a Nash equilibrium.**

# Step by Step: PoA in the Load Balancing Game




Worst equilibrium: 4

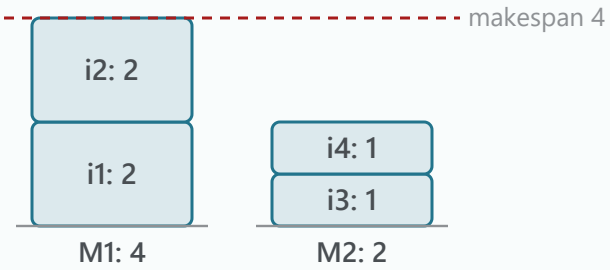
Best equilibrium: 3 (optimum is an eq.)

PoA = 4/3, PoS = 1

Script: 6 of 16 assignments are eq.

Here **PoA = 4/3** and PoS = 1: the optimum is an equilibrium, but so is a bad assignment.

# Step by Step: PoA in the Load Balancing Game




**m identical machines, pure eq.:**

**PoA = 2 − 2/(m + 1)**

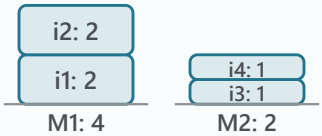
**m = 2 → 4/3 (this example is tight)**

**Mixed equilibria: grows very slowly in m**

General result (Finn–Horowitz, 1979; Koutsoupias–Papadimitriou, 1999): for pure equilibria PoA = 2 − 2/(m+1) < 2. For mixed equilibria the PoA grows very slowly but without bound in m (Koutsoupias–Mavronicolas–Spirakis, 2003; Czumaj–Vöcking, 2002).

# Code Trace: `is_nash(sizes, assign, m)`

i1, i2 → M1; i3, i4 → M2




Call: job sizes and each job's machine (0 = M1, 1 = M2).

Step 1 / 11

```
▶ def is_nash(sizes, assign, m):
2   load = [0] * m
3   for j, k in enumerate(assign):
4     load[k] += sizes[j]
5   for j, k in enumerate(assign):
6     for h in range(m):
7       if load[h] + sizes[j] < load[k]:
8         return False      # j moves
9   return True             # nobody moves
```

**VARIABLES**

`sizes = [2, 2, 1, 1]`

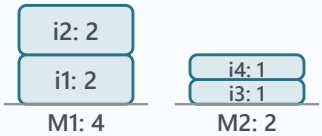
`assign = [0, 0, 1, 1]` `m = 2`

**OUTPUT**

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

# Code Trace: `is_nash(sizes, assign, m)`

load = [0, 0]




The load array is zeroed.

Step 2 / 11

```
1 def is_nash(sizes, assign, m):
▶   load = [0] * m
3   for j, k in enumerate(assign):
4     load[k] += sizes[j]
5   for j, k in enumerate(assign):
6     for h in range(m):
7       if load[h] + sizes[j] < load[k]:
8         return False      # j moves
9   return True             # nobody moves
```

**VARIABLES**

`sizes = [2, 2, 1, 1]`

`assign = [0, 0, 1, 1]` `m = 2`

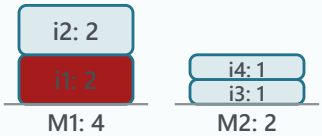
`load = [0, 0]`

**OUTPUT**

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

# Code Trace: `is_nash(sizes, assign, m)`

load = [2, 0]




j = 0: i1 (size 2) is added to M1 → load = [2, 0].

Step 3 / 11

```
1 def is_nash(sizes, assign, m):
2   load = [0] * m
▶   for j, k in enumerate(assign):
▶     load[k] += sizes[j]
5   for j, k in enumerate(assign):
6     for h in range(m):
7       if load[h] + sizes[j] < load[k]:
8         return False      # j moves
9   return True             # nobody moves
```

**VARIABLES**

`sizes = [2, 2, 1, 1]`

`assign = [0, 0, 1, 1]` `m = 2`

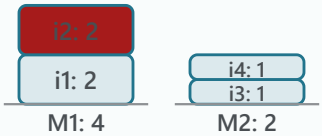
`load = [2, 0]` `j = 0` `k = 0`

**OUTPUT**

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

# Code Trace: `is_nash(sizes, assign, m)`

load = [4, 0]




j = 1: i2 (size 2) is added to M1 → load = [4, 0].

Step 4 / 11

```
1 def is_nash(sizes, assign, m):
2   load = [0] * m
▶   for j, k in enumerate(assign):
▶     load[k] += sizes[j]
5   for j, k in enumerate(assign):
6     for h in range(m):
7       if load[h] + sizes[j] < load[k]:
8         return False      # j moves
9   return True             # nobody moves
```

**VARIABLES**

`sizes = [2, 2, 1, 1]`

`assign = [0, 0, 1, 1]` `m = 2`

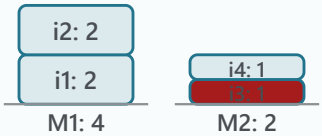
`load = [4, 0]` `j = 1` `k = 0`

**OUTPUT**

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

# Code Trace: `is_nash(sizes, assign, m)`

load = [4, 1]




j = 2: i3 (size 1) is added to M2 → load = [4, 1].

Step 5 / 11

```
1 def is_nash(sizes, assign, m):
2   load = [0] * m
▶   for j, k in enumerate(assign):
▶     load[k] += sizes[j]
5   for j, k in enumerate(assign):
6     for h in range(m):
7       if load[h] + sizes[j] < load[k]:
8         return False      # j moves
9   return True             # nobody moves
```

**VARIABLES**

`sizes = [2, 2, 1, 1]`

`assign = [0, 0, 1, 1]` `m = 2`

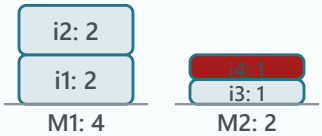
`load = [4, 1]` `j = 2` `k = 1`

**OUTPUT**

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

# Code Trace: `is_nash(sizes, assign, m)`

load = [4, 2]




j = 3: i4 (size 1) is added to M2 → load = [4, 2].

Step 6 / 11

```
1 def is_nash(sizes, assign, m):
2   load = [0] * m
▶   for j, k in enumerate(assign):
▶     load[k] += sizes[j]
5   for j, k in enumerate(assign):
6     for h in range(m):
7       if load[h] + sizes[j] < load[k]:
8         return False      # j moves
9   return True             # nobody moves
```

**VARIABLES**

`sizes = [2, 2, 1, 1]`

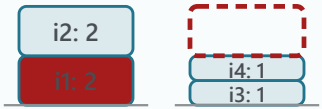
`assign = [0, 0, 1, 1]` `m = 2`

`load = [4, 2]` `j = 3` `k = 1`

**OUTPUT**

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

# Code Trace: `is_nash(sizes, assign, m)`




load[1] + sizes[0] = 4
load[0] = 4 → stays

j = 0, h = 1: if i1 moves to M2, 2 + 2 = 4; now 4. 4 < 4? **No**.



```
1 def is_nash(sizes, assign, m):
2   load = [0] * m
3   for j, k in enumerate(assign):
4     load[k] += sizes[j]
▶   for j, k in enumerate(assign):
▶     for h in range(m):
▶       if load[h] + sizes[j] < load[k]:
8         return False      # j moves
9   return True             # nobody moves
```

**VARIABLES**

`sizes = [2, 2, 1, 1]`
`assign = [0, 0, 1, 1]` `m = 2`
`load = [4, 2]` `j = 0` `k = 0` `h = 1`

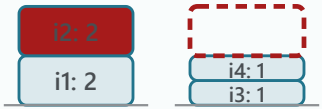
**OUTPUT**

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

# Code Trace: `is_nash(sizes, assign, m)`

load[1] + sizes[1] = 4

load[0] = 4 → stays




j = 1, h = 1: if i2 moves to M2, 2 + 2 = 4; now 4. 4 < 4? **No**.

Step 8 / 11

```
1 def is_nash(sizes, assign, m):
2   load = [0] * m
3   for j, k in enumerate(assign):
4     load[k] += sizes[j]
▶   for j, k in enumerate(assign):
▶     for h in range(m):
▶       if load[h] + sizes[j] < load[k]:
8         return False      # j moves
9   return True             # nobody moves
```

**VARIABLES**

`sizes = [2, 2, 1, 1]`

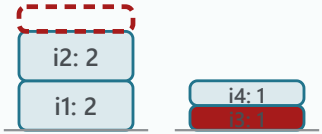
`assign = [0, 0, 1, 1]` `m = 2`

`load = [4, 2]` `j = 1` `k = 0` `h = 1`

**OUTPUT**

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

# Code Trace: `is_nash(sizes, assign, m)`




load[0] + sizes[2] = 5
load[1] = 2 → stays

j = 2, h = 0: if i3 moves to M1, 4 + 1 = 5; now 2. 5 < 2? **No**.



```
1 def is_nash(sizes, assign, m):
2   load = [0] * m
3   for j, k in enumerate(assign):
4     load[k] += sizes[j]
▶   for j, k in enumerate(assign):
▶     for h in range(m):
▶       if load[h] + sizes[j] < load[k]:
8         return False      # j moves
9   return True             # nobody moves
```

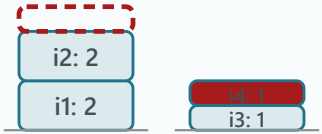
**VARIABLES**

`sizes = [2, 2, 1, 1]`
`assign = [0, 0, 1, 1]` `m = 2`
`load = [4, 2]` `j = 2` `k = 1` `h = 0`

**OUTPUT**

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

# Code Trace: `is_nash(sizes, assign, m)`




load[0] + sizes[3] = 5
load[1] = 2 → stays

j = 3, h = 0: if i4 moves to M1, 4 + 1 = 5; now 2. 5 < 2? **No**.

Step 10 / 11

```
1 def is_nash(sizes, assign, m):
2   load = [0] * m
3   for j, k in enumerate(assign):
4     load[k] += sizes[j]
▶   for j, k in enumerate(assign):
▶     for h in range(m):
▶       if load[h] + sizes[j] < load[k]:
8         return False      # j moves
9   return True             # nobody moves
```

VARIABLES

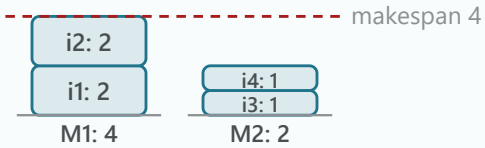
`sizes = [2, 2, 1, 1]`
`assign = [0, 0, 1, 1]` `m = 2`
`load = [4, 2]` `j = 3` `k = 1` `h = 0`

OUTPUT

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

# Code Trace: `is_nash(sizes, assign, m)`

is_nash → True




No job could escape to a better machine → **True**: {2, 2} | {1, 1} is a Nash equilibrium (makespan 4).

Step 11 / 11

```
1  def is_nash(sizes, assign, m):
2    load = [0] * m
3    for j, k in enumerate(assign):
4      load[k] += sizes[j]
5    for j, k in enumerate(assign):
6      for h in range(m):
7        if load[h] + sizes[j] < load[k]:
8          return False      # j moves
▶    return True             # nobody moves
```

**VARIABLES**

`sizes = [2, 2, 1, 1]`

`assign = [0, 0, 1, 1]` `m = 2`

`load = [4, 2]`

**OUTPUT**

```
True
```

For h = k (its own machine) the condition is always false; the steps show only h ≠ k.

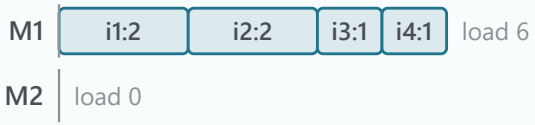
# Brute Force: All 16 Assignments

| Split (M1 \| M2, symmetric) | Assignments | Makespan | Nash? |
|---|---|---|---|
| {2, 1} \| {2, 1} | 4 | 3 | ✓ eq. |
| {2, 1, 1} \| {2} | 4 | 4 | ✗ |
| {2, 2} \| {1, 1} | 2 | 4 | ✓ eq. |
| {2, 2, 1} \| {1} | 4 | 5 | ✗ |
| {2, 2, 1, 1} \| { } | 2 | 6 | ✗ |

The script tried all $2^4 = 16$ assignments and every job's deviation in each. The equilibria are those with makespan 3 and 4 → **PoA = 4/3** (worst equilibrium), **PoS = 1** (best equilibrium = optimum).

Best-response dynamics in this game always stop at an equilibrium: every profitable move shrinks the sorted load list. Which equilibrium is reached depends on the start and the order of moves.

# Code Trace: PoA and PoS by Trying All Assignments




1/16 assign = (0, 0, 0, 0)

makespan 6 · Nash? ✗

opt = 6 · eqs = []

(0, 0, 0, 0): all on M1, loads [6, 0]; i1 moving to M2 gets 2 < 6 ⇒ not an equilibrium. opt is 6 for now.

Step 1 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

assign = (0, 0, 0, 0)

load = [6, 0]  opt = 6

eqs = []

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




2/16 assign = (0, 0, 0, 1)

makespan 5 · Nash? ✗

opt = 5 · eqs = []

(0, 0, 0, 1): loads [5, 1]; i1 moving to the other machine gets 3 < 5 ⇒ not an equilibrium.

Step 2 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

assign = (0, 0, 0, 1)

load = [5, 1]  opt = 5

eqs = []

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




3/16 assign = (0, 0, 1, 0)

makespan 5 · Nash? ✗

opt = 5 · eqs = []

(0, 0, 1, 0): loads [5, 1]; i1 moving to the other machine gets 3 < 5 ⇒ not an equilibrium.

Step 3 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

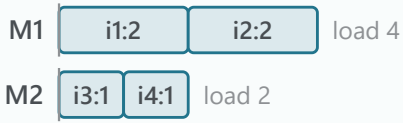
assign = (0, 0, 1, 0)

load = [5, 1]  opt = 5

eqs = []

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




4/16 assign = (0, 0, 1, 1)

makespan 4 · Nash? ✓

opt = 4 · eqs = [4]

(0, 0, 1, 1): loads [4, 2], nobody can escape ⇒ **equilibrium**; makespan 4 is added to the list.



```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
▶        eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

`assign = (0, 0, 1, 1)`

`load = [4, 2]` `opt = 4`

`eqs = [4]`

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




5/16 assign = (0, 1, 0, 0)

makespan 4 · Nash? ✗

opt = 4 · eqs = [4]

(0, 1, 0, 0): loads [4, 2]; i3 moving to the other machine gets 3 < 4 ⇒ not an equilibrium.

Step 5 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

assign = (0, 1, 0, 0)

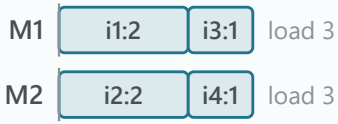
load = [4, 2]

opt = 4

eqs = [4]

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




6/16 assign = (0, 1, 0, 1)

makespan 3 · Nash? ✓

opt = 3 · eqs = [4, 3]

(0, 1, 0, 1): loads [3, 3], nobody can escape ⇒ **equilibrium**; makespan 3 is added to the list.

Step 6 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
▶        eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

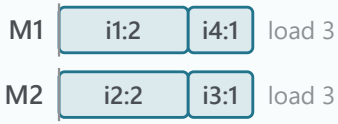
`assign = (0, 1, 0, 1)`

`load = [3, 3]` `opt = 3`

`eqs = [4, 3]`

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




7/16 assign = (0, 1, 1, 0)

makespan 3 · Nash? ✓

opt = 3 · eqs = [4, 3, 3]

(0, 1, 1, 0): loads [3, 3], nobody can escape ⇒ **equilibrium**; makespan 3 is added to the list.

Step 7 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
▶        eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

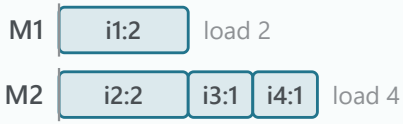
assign = (0, 1, 1, 0)

load = [3, 3]  opt = 3

eqs = [4, 3, 3]

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




8/16 assign = (0, 1, 1, 1)

makespan 4 · Nash? ✗

opt = 3 · eqs = [4, 3, 3]

(0, 1, 1, 1): loads [2, 4]; i3 moving to the other machine gets 3 < 4 ⇒ not an equilibrium.

Step 8 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

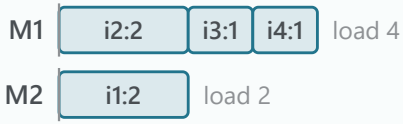
assign = (0, 1, 1, 1)

load = [2, 4]  opt = 3

eqs = [4, 3, 3]

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




9/16 assign = (1, 0, 0, 0)

makespan 4 · Nash? ✗

opt = 3 · eqs = [4, 3, 3]

(1, 0, 0, 0): loads [4, 2]; i3 moving to the other machine gets 3 < 4 ⇒ not an equilibrium.

Step 9 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

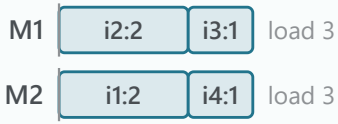
assign = (1, 0, 0, 0)

load = [4, 2]  opt = 3

eqs = [4, 3, 3]

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




10/16 assign = (1, 0, 0, 1)

makespan 3 · Nash? ✓

opt = 3 · eqs = [4, 3, 3, 3]

(1, 0, 0, 1): loads [3, 3], nobody can escape ⇒ **equilibrium**; makespan 3 is added to the list.

Step 10 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
▶        eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

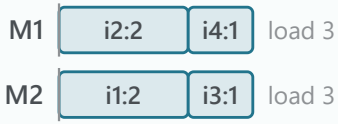
`assign = (1, 0, 0, 1)`

`load = [3, 3]` `opt = 3`

`eqs = [4, 3, 3, 3]`

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




11/16 assign = (1, 0, 1, 0)

makespan 3 · Nash? ✓

opt = 3 · eqs = [4, 3, 3, 3, 3]

(1, 0, 1, 0): loads [3, 3], nobody can escape ⇒ **equilibrium**; makespan 3 is added to the list.

Step 11 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
▶        eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

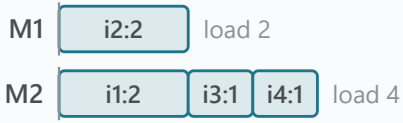
`assign = (1, 0, 1, 0)`

`load = [3, 3]` `opt = 3`

`eqs = [4, 3, 3, 3, 3]`

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




12/16 assign = (1, 0, 1, 1)

makespan 4 · Nash? ✗

opt = 3 · eqs = [4, 3, 3, 3, 3]

(1, 0, 1, 1): loads [2, 4]; i3 moving to the other machine gets 3 < 4 ⇒ not an equilibrium.

Step 12 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

assign = (1, 0, 1, 1)

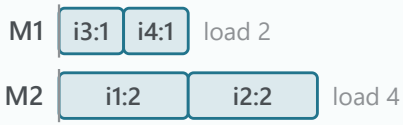
load = [2, 4]

opt = 3

eqs = [4, 3, 3, 3, 3]

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




13/16 assign = (1, 1, 0, 0)

makespan 4 · Nash? ✓

opt = 3 · eqs = [4, 3, 3, 3, 3, 4]

(1, 1, 0, 0): loads [2, 4], nobody can escape ⇒ **equilibrium**; makespan 4 is added to the list.

Step 13 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
▶        eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

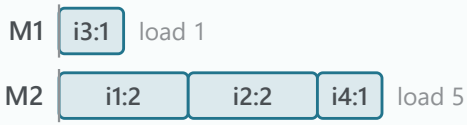
assign = (1, 1, 0, 0)

load = [2, 4]  opt = 3

eqs = [4, 3, 3, 3, 3, 4]

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




14/16 assign = (1, 1, 0, 1)

makespan 5 · Nash? ✗

opt = 3 · eqs = [4, 3, 3, 3, 3, 4]

(1, 1, 0, 1): loads [1, 5]; i1 moving to the other machine gets 3 < 5 ⇒ not an equilibrium.

Step 14 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

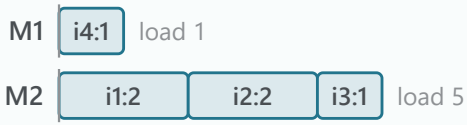
assign = (1, 1, 0, 1)

load = [1, 5]  opt = 3

eqs = [4, 3, 3, 3, 3, 4]

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




15/16 assign = (1, 1, 1, 0)

makespan 5 · Nash? ✗

opt = 3 · eqs = [4, 3, 3, 3, 3, 4]

(1, 1, 1, 0): loads [1, 5]; i1 moving to the other machine gets 3 < 5 ⇒ not an equilibrium.

Step 15 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

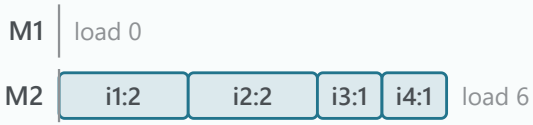
assign = (1, 1, 1, 0)

load = [1, 5]  opt = 3

eqs = [4, 3, 3, 3, 3, 4]

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




16/16 assign = (1, 1, 1, 1)

makespan 6 · Nash? ✗

opt = 3 · eqs = [4, 3, 3, 3, 3, 4]

(1, 1, 1, 1): loads [0, 6]; i1 moving to the other machine gets 2 < 6 ⇒ not an equilibrium.

Step 16 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
▶    for assign in product(range(m), repeat=len(sizes)):
▶      load = [0] * m
▶      for j, k in enumerate(assign):
▶        load[k] += sizes[j]
▶      opt = min(opt, max(load))
▶      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
11   return max(eqs) / opt, min(eqs) / opt
12 r = poa_pos([2, 2, 1, 1], 2)
13 print(round(r[0], 3), r[1])
```

VARIABLES

assign = (1, 1, 1, 1)

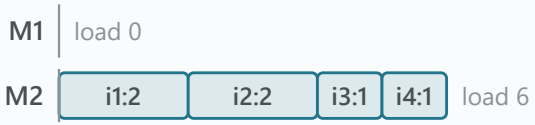
load = [0, 6]

opt = 3

eqs = [4, 3, 3, 3, 3, 4]

OUTPUT

# Code Trace: PoA and PoS by Trying All Assignments




16/16 assign = (1, 1, 1, 1)

makespan 6 · Nash? ✗

opt = 3 · eqs = [4, 3, 3, 3, 3, 4]

Equilibria [4, 3, 3, 3, 3, 4] ⇒ PoA = 4/3, PoS = 3/3 = 1. There are $m^n$ assignments: exponential for large games.

Step 17 / 17

```
1  from itertools import product
2  def poa_pos(sizes, m):
3    opt, eqs = float("inf"), []
4    for assign in product(range(m), repeat=len(sizes)):
5      load = [0] * m
6      for j, k in enumerate(assign):
7        load[k] += sizes[j]
8      opt = min(opt, max(load))
9      if is_nash(sizes, assign, m):    # eq. test
10       eqs.append(max(load))
▶    return max(eqs) / opt, min(eqs) / opt
▶  r = poa_pos([2, 2, 1, 1], 2)
▶  print(round(r[0], 3), r[1])
```

VARIABLES

opt = 3

eqs = [4, 3, 3, 3, 3, 4]

OUTPUT

```
1.333 1.0
```

# Machines with Different Speeds (Related Machines)

- Machine i has speed $s_i$; a job's completion time is its machine's total load divided by $s_i$. A pure Nash equilibrium still always exists.
- Example: speeds A = 2, B = 1, C = 1; job sizes 2, 1, 1. Social cost: the maximum completion time.

| Assignment | Times (A, B, C) | Max | Nash? |
|---|---|---|---|
| A {2}, B {1}, C {1} | 1, 1, 1 | 1 | yes (optimum) |
| A {1, 1}, B {2}, C { } | 1, 2, 0 | **2** | yes |

In the second assignment, the size-2 job would wait (2 + 2)/2 = 2 on A or 2/1 = 2 on C; a size-1 job moving to C still waits 1: nobody gains. Ratio 2/1 = 2, above the bound 2 − 2/4 = 1.5 for three identical machines.

On identical machines the loss of a pure equilibrium stays below 2 (2 − 2/(m+1)). With different speeds the loss grows without bound (if very slowly) as machines are added (Czumaj–Vöcking, 2002). Levels 2 and 3 of the Load Balancing game use this model.

# Back to the Warm-up: The Load Balancing Game

### The game measures something else

Here social cost is the sum of completion times, not the makespan. Level 2, CPU Scheduling: 7 units, speeds 2, 1, 1.

| Loads (A, B, C) | Total time | Nash? |
|---|---|---|
| (3, 2, 2) | 3·1.5 + 2·2 + 2·2 = **12.5** | yes (optimum) |
| (4, 2, 1) | 4·2 + 2·2 + 1·1 = **13** | yes (worst) |

PoA = 13 / 12.5 = **1.04**. In (4, 2, 1), a job moving from A to C still waits 2: nobody gains. The mirror (4, 1, 2) costs 13 too.

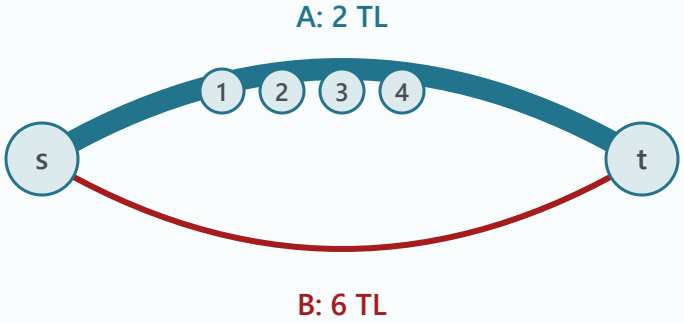
### Where does selfish placement lead?

If each arriving job picks its fastest server (ties alphabetical), the order is A, A, B, C, A, A, B: exactly the worst equilibrium (4, 2, 1). The game's selfish view uses this greedy placement.

Level 1 (6 jobs, identical servers): the only equilibrium (2, 2, 2) is optimal, PoA = 1. Loss appears once servers differ.

The 4/3 for {2, 2, 1, 1} measures makespan, the game total time: the PoA depends on the social cost used.

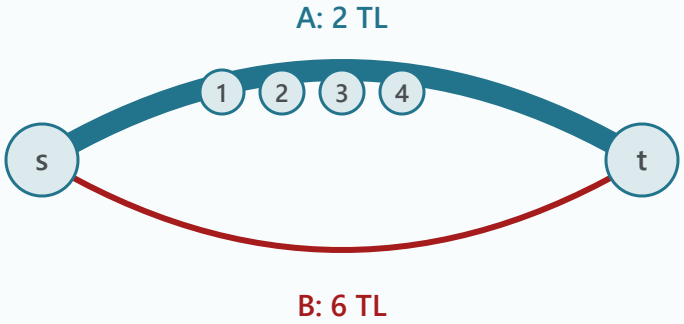
# PoA ≠ PoS in the Same Game




link cost is split equally among its users

4 players rent a link from s to t: A costs 2 TL, B costs 6 TL; users **split the cost equally** (fair cost sharing). Thickness: number of players on the link.

# PoA ≠ PoS in the Same Game




all on A: 0.5 TL each

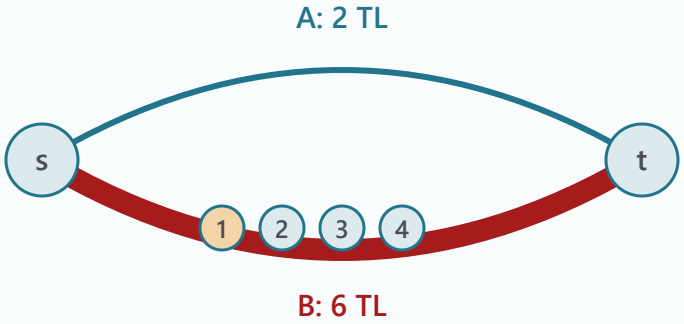
moving alone to B: 6 TL

total 2 = optimum

All on A: each pays 0.5; one moving alone to B would pay 6 ⇒ nobody moves.

**Equilibrium, total 2 = optimum.**

# PoA ≠ PoS in the Same Game




all on B: 1.5 TL each

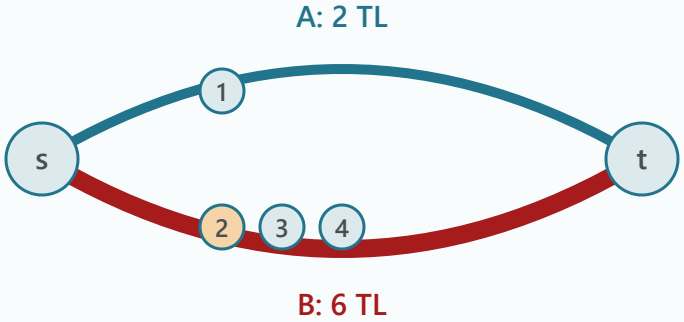
moving alone to A: 2 TL

total 6

All on B: each pays 1.5; one moving alone to A would pay 2 (2 > 1.5) ⇒ nobody moves.

**Also an equilibrium, total 6.**

# PoA ≠ PoS in the Same Game



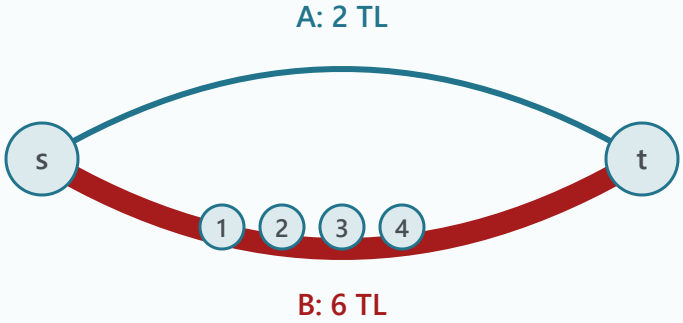

the 1 on A: 2 TL

the 3 on B: 2 TL

B to A: 1 TL ⇒ moves

Split profile (1 A, 3 B): a player on B moving to A pays 1 < 2 ⇒ not an equilibrium. The script checked all 16 profiles: only "all A" and "all B" are equilibria.

# PoA ≠ PoS in the Same Game

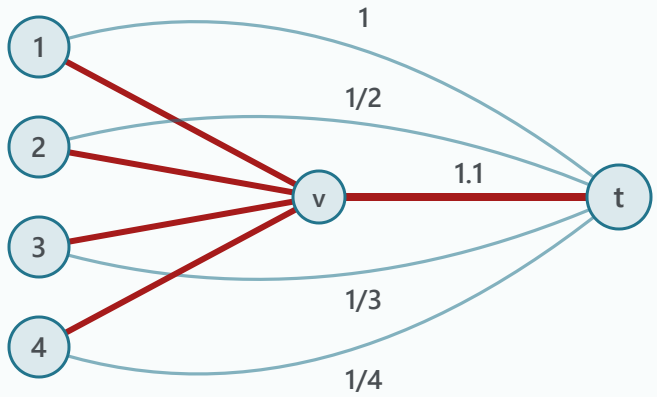



PoA = 6/2 = 3

PoS = 2/2 = 1

**PoA = 6/2 = 3** (worst equilibrium), **PoS = 1** (best equilibrium). A coordination problem: if players can be steered to the good equilibrium there is no loss; the PoS measures this optimistic scenario.

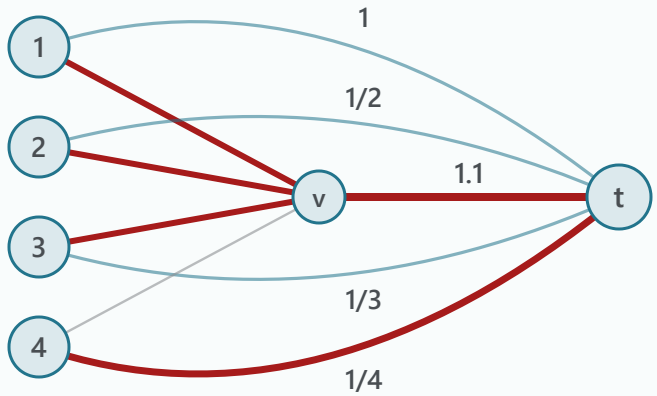
# Step by Step: The Price of Stability (The $H_n$ Example)




| player | path | pays |
|---|---|---|
| 1 | shared | 0.275 |
| 2 | shared | 0.275 |
| 3 | shared | 0.275 |
| 4 | shared | 0.275 |
| total | | 1.1 |

Each player reaches t by one of two paths: a private path of cost 1/i used only by them, or a shared path of cost 1 + ε = 1.1 (ε: a small extra cost) whose cost users split equally. Optimum: everyone shares, total **1.1**, 0.275 each.

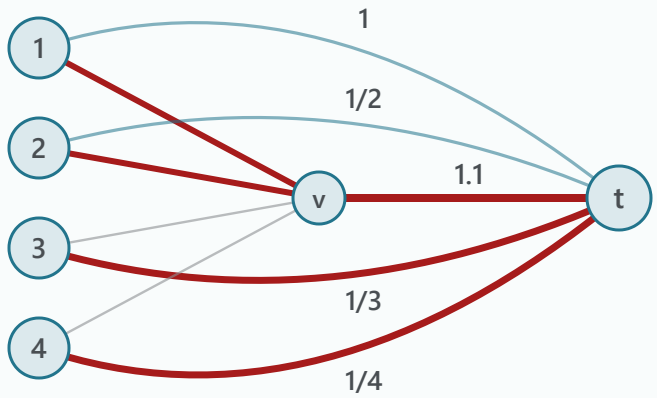
# Step by Step: The Price of Stability (The $H_n$ Example)




| player | path | pays |
|---|---|---|
| 1 | shared | 0.367 |
| 2 | shared | 0.367 |
| 3 | shared | 0.367 |
| 4 | own | 0.25 |
| total | | 1.35 |

Player 4: own path 1/4 = 0.25 < 0.275 → **leaves**. The remaining three now pay 1.1/3 ≈ 0.367.

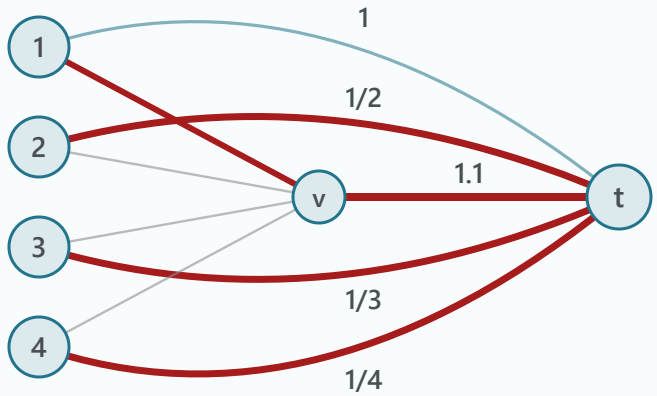
# Step by Step: The Price of Stability (The $H_n$ Example)




| player | path | pays |
|---|---|---|
| 1 | shared | 0.55 |
| 2 | shared | 0.55 |
| 3 | own | 0.333 |
| 4 | own | 0.25 |
| total | | 1.683 |

Player 3: 1/3 ≈ 0.333 < 0.367 → leaves. The remaining two pay 0.55.

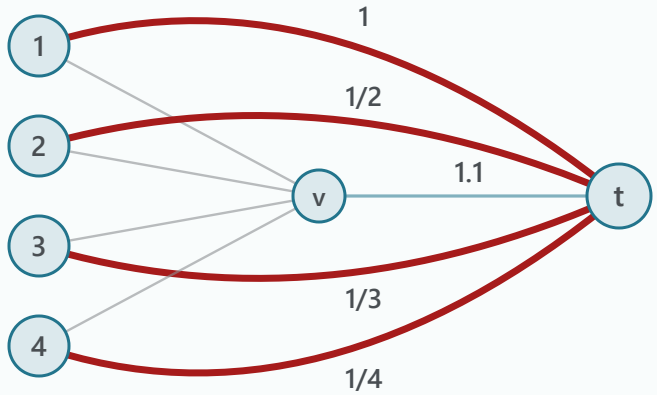
# Step by Step: The Price of Stability (The $H_n$ Example)




| player | path | pays |
|---|---|---|
| 1 | shared | 1.1 |
| 2 | own | 0.5 |
| 3 | own | 0.333 |
| 4 | own | 0.25 |
| total | | 2.183 |

Player 2: 1/2 = 0.5 < 0.55 → leaves. Player 1 pays for the shared path alone: 1.1.

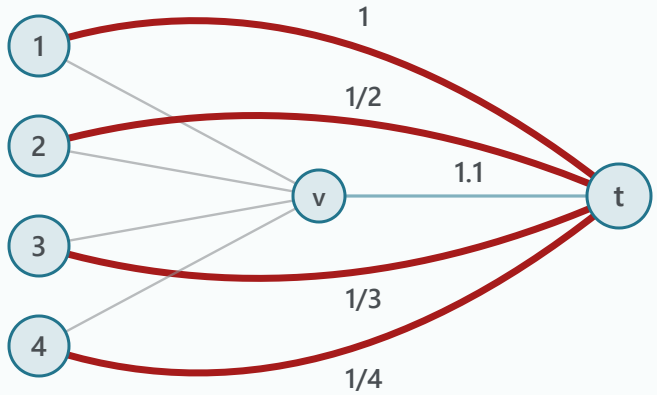
# Step by Step: The Price of Stability (The $H_n$ Example)




| player | path | pays |
|---|---|---|
| 1 | own | 1 |
| 2 | own | 0.5 |
| 3 | own | 0.333 |
| 4 | own | 0.25 |
| total | | 2.083 |

Player 1: own path 1 < 1.1 → leaves. Everyone on their own path: 1 + 1/2 + 1/3 + 1/4 = **25/12 ≈ 2.083**.

# Step by Step: The Price of Stability (The $H_n$ Example)




| player | path | pays |
|---|---|---|
| 1 | own | 1 |
| 2 | own | 0.5 |
| 3 | own | 0.333 |
| 4 | own | 0.25 |
| total | | 2.083 |

Script: this is the **only** equilibrium of the 16 profiles → PoS = PoA = (25/12) / 1.1 = 125/66 ≈ 1.894. As ε → 0 with n players, PoS → $H_n$ = 1 + 1/2 + … + 1/n.

# Why PoS ≤ $H_n$? Potential and Cost

**Notation.** $c_e$: cost of link e; $n_e$: number of players using e (each pays $c_e/n_e$). H(k) = 1 + 1/2 + … + 1/k; e.g. H(1) = 1, H(4) = 25/12 ≈ 2.08. $H_n$ = H(n). Total cost C(s): the sum of the costs of the links in use.

# Why PoS ≤ $H_n$? Potential and Cost

**Notation.** $c_e$: cost of link e; $n_e$: number of players using e (each pays $c_e/n_e$). H(k) = 1 + 1/2 + … + 1/k; e.g. H(1) = 1, H(4) = 25/12 ≈ 2.08. $H_n$ = H(n). Total cost C(s): the sum of the costs of the links in use.

**Rosenthal's potential.** Imagine players joining a link one at a time: the 1st writes $c_e$, the 2nd $c_e/2$, …, the kth $c_e/k$. The link contributes $c_e \cdot H(n_e)$: $\mathbf{\Phi(s) = \Sigma_e\, c_e \cdot H(n_e)}$.

# Why PoS ≤ $H_n$? Potential and Cost

**Notation.** $c_e$: cost of link e; $n_e$: number of players using e (each pays $c_e/n_e$). H(k) = 1 + 1/2 + … + 1/k; e.g. H(1) = 1, H(4) = 25/12 ≈ 2.08. $H_n$ = H(n). Total cost C(s): the sum of the costs of the links in use.

**Rosenthal's potential.** Imagine players joining a link one at a time: the 1st writes $c_e$, the 2nd $c_e/2$, …, the kth $c_e/k$. The link contributes $c_e \cdot H(n_e)$: $\mathbf{\Phi(s) = \Sigma_e\, c_e \cdot H(n_e)}$.

**Example** (the $H_n$ example, four players). All on the shared path: C = 1.1 and Φ = 1.1·H(4) = 1.1·25/12 ≈ 2.29. All on their own paths: each path has 1 user, H(1) = 1, so Φ = C = 25/12 ≈ 2.08.

# Why PoS ≤ $H_n$? Potential and Cost

**Notation.** $c_e$: cost of link e; $n_e$: number of players using e (each pays $c_e/n_e$). H(k) = 1 + 1/2 + … + 1/k; e.g. H(1) = 1, H(4) = 25/12 ≈ 2.08. $H_n$ = H(n). Total cost C(s): the sum of the costs of the links in use.

**Rosenthal's potential.** Imagine players joining a link one at a time: the 1st writes $c_e$, the 2nd $c_e/2$, …, the kth $c_e/k$. The link contributes $c_e \cdot H(n_e)$: $\mathbf{\Phi(s) = \Sigma_e\, c_e \cdot H(n_e)}$.

**Example** (the $H_n$ example, four players). All on the shared path: C = 1.1 and Φ = 1.1·H(4) = 1.1·25/12 ≈ 2.29. All on their own paths: each path has 1 user, H(1) = 1, so Φ = C = 25/12 ≈ 2.08.

**The potential is between 1 and $H_n$ times the cost.** On every link in use, $1 \le H(n_e) \le H_n$ (there are at most n users). Multiply each term by $c_e$ and sum: for every s, $\mathbf{C(s) \le \Phi(s) \le H_n \cdot C(s)}$.

# PoS ≤ $H_n$: Walking from the Optimum to an Equilibrium

**1. Start at the optimum.** Run best-response dynamics from $s^{OPT}$: at each step one player switches to a path that is cheaper for them.

# PoS ≤ $H_n$: Walking from the Optimum to an Equilibrium

**1. Start at the optimum.** Run best-response dynamics from $s^{OPT}$: at each step one player switches to a path that is cheaper for them.

**2. Φ drops at every move.** In a potential game the deviator's change in cost equals the change in Φ; if the player gains, Φ falls. There are finitely many profiles, so the dynamics stop at an equilibrium s with **Φ(s) ≤ Φ(OPT)**.

# PoS ≤ $H_n$: Walking from the Optimum to an Equilibrium

**1. Start at the optimum.** Run best-response dynamics from $s^{OPT}$: at each step one player switches to a path that is cheaper for them.

**2. Φ drops at every move.** In a potential game the deviator's change in cost equals the change in Φ; if the player gains, Φ falls. There are finitely many profiles, so the dynamics stop at an equilibrium s with **Φ(s) ≤ Φ(OPT)**.

**3. Chain it.** C(s) ≤ Φ(s) (potential is at least the cost) ≤ Φ(OPT) (step 2) ≤ $H_n$·C(OPT) (potential is at most $H_n$ times the cost). In the example: 2.08 ≤ 2.08 ≤ 2.29 ≤ 2.29.

# PoS ≤ $H_n$: Walking from the Optimum to an Equilibrium

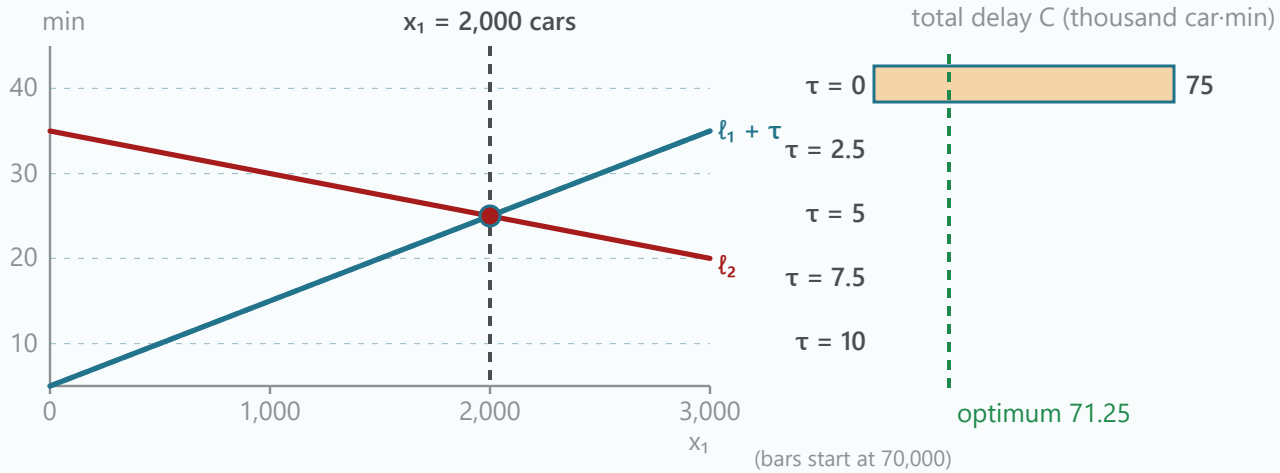
**1. Start at the optimum.** Run best-response dynamics from $s^{OPT}$: at each step one player switches to a path that is cheaper for them.

**2. Φ drops at every move.** In a potential game the deviator's change in cost equals the change in Φ; if the player gains, Φ falls. There are finitely many profiles, so the dynamics stop at an equilibrium s with **Φ(s) ≤ Φ(OPT)**.

**3. Chain it.** C(s) ≤ Φ(s) (potential is at least the cost) ≤ Φ(OPT) (step 2) ≤ $H_n$·C(OPT) (potential is at most $H_n$ times the cost). In the example: 2.08 ≤ 2.08 ≤ 2.29 ≤ 2.29.

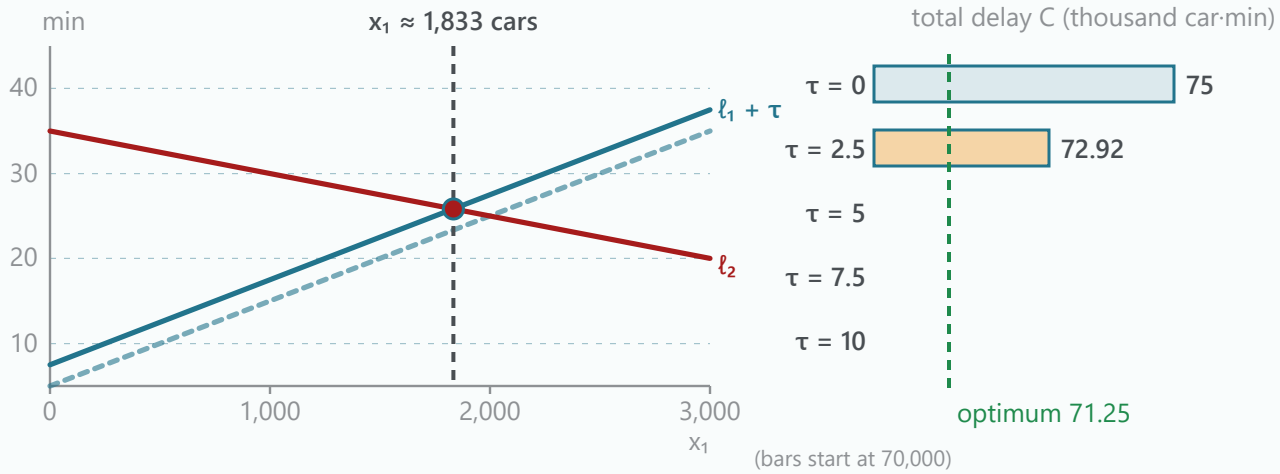
Some equilibrium costs at most $H_n$ times the optimum: **PoS ≤ $H_n$** (Anshelevich et al., 2004). $H_n$ grows very slowly ($H_{100}$ ≈ 5.19; roughly ln n). The $H_n$ example (a directed network) shows the bound is tight. The PoA, by contrast, can be as large as n.

# Step by Step: A Congestion Toll Shifts the Equilibrium



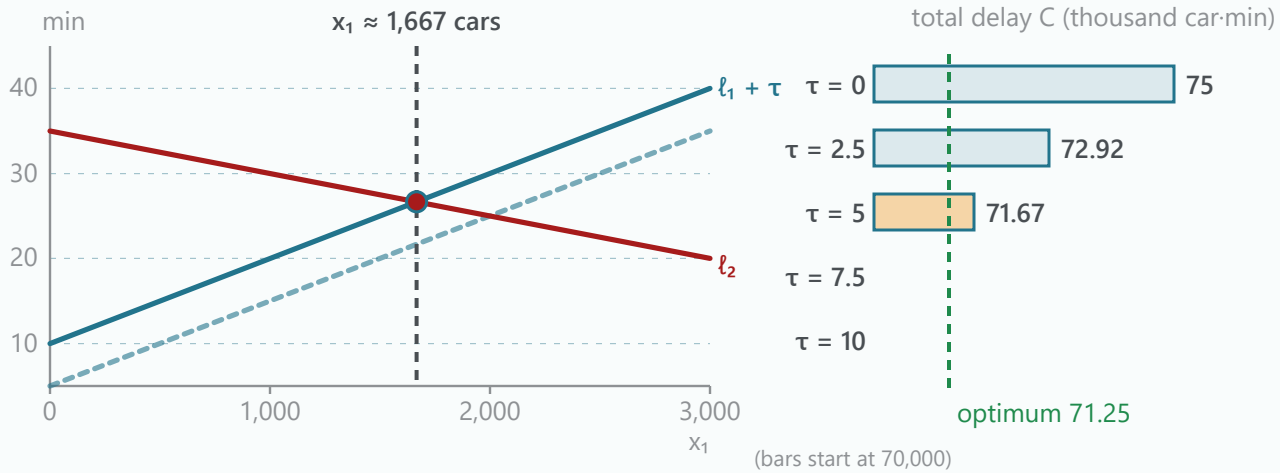

Two-route network: $\ell_1 = 5 + x_1/100$, $\ell_2 = 20 + x_2/200$, 3,000 cars in total. No toll ($\tau = 0$): equilibrium $\ell_1 = \ell_2 \rightarrow x_1 = 2{,}000$, total 75,000 car·min. Drivers count the toll as an extra cost in minutes.

# Step by Step: A Congestion Toll Shifts the Equilibrium



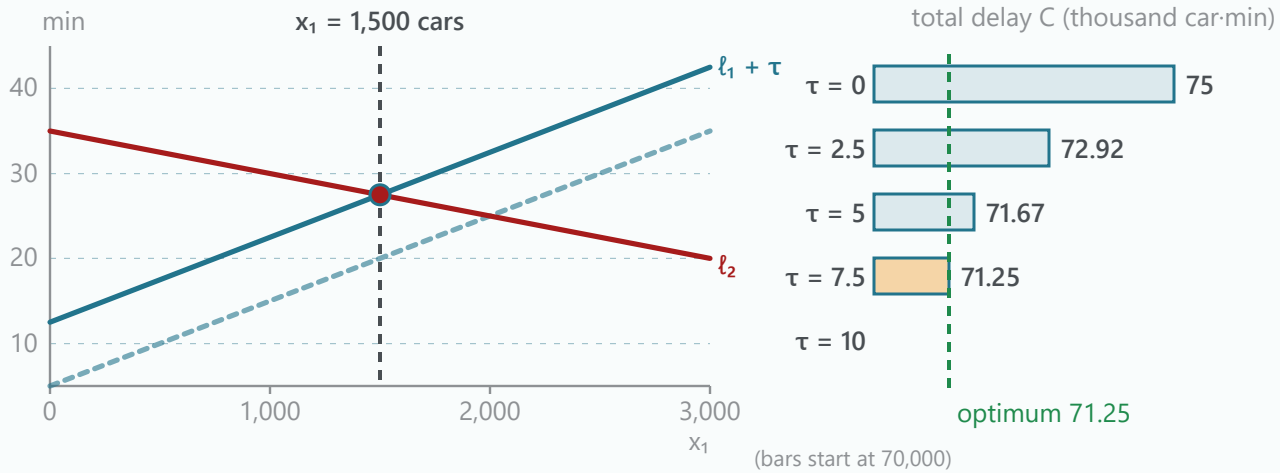

$\tau = 2.5$: the Route 1 curve shifts up and the crossing moves left: $x_1 \approx 1{,}833$, total $\approx 72{,}917$. A toll is a *transfer*; it is not added to total delay.

# Step by Step: A Congestion Toll Shifts the Equilibrium



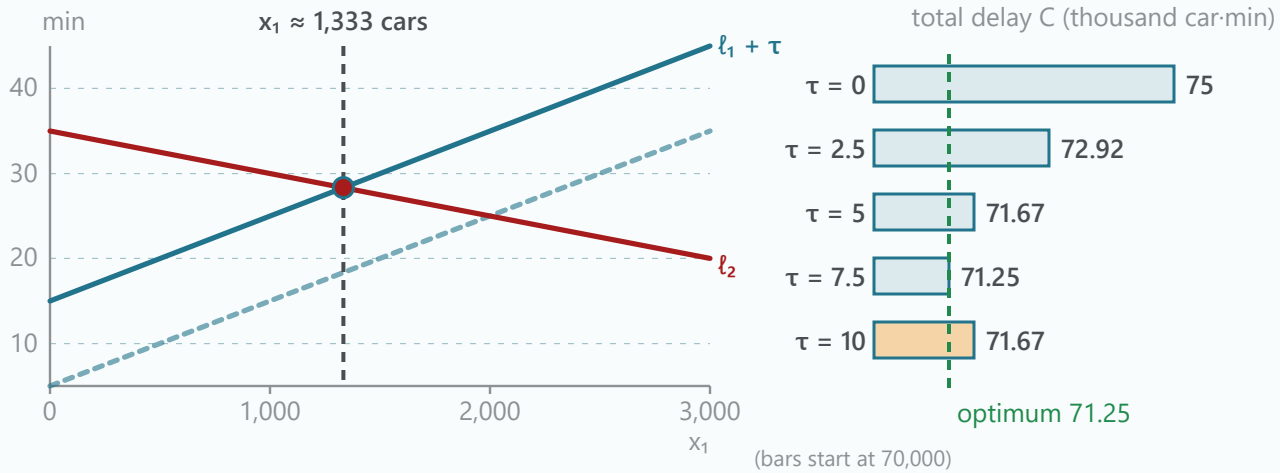

τ = 5: $x_1$ ≈ 1,667, total ≈ 71,667. Each increase pushes some drivers onto the wide Route 2.

# Step by Step: A Congestion Toll Shifts the Equilibrium



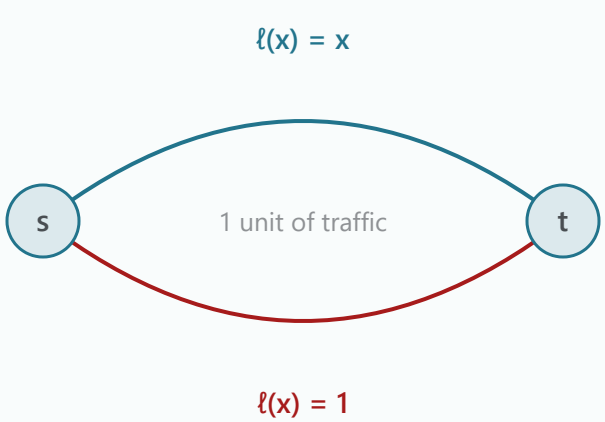

τ = 7.5: $x_1$ = 1,500 → the **system optimum** (71,250) is now an equilibrium. Marginal-cost tolls are $x_1\ell'_1$ = 15 min on Route 1 and $x_2\ell'_2$ = 7.5 on Route 2; what matters is the difference: 7.5.

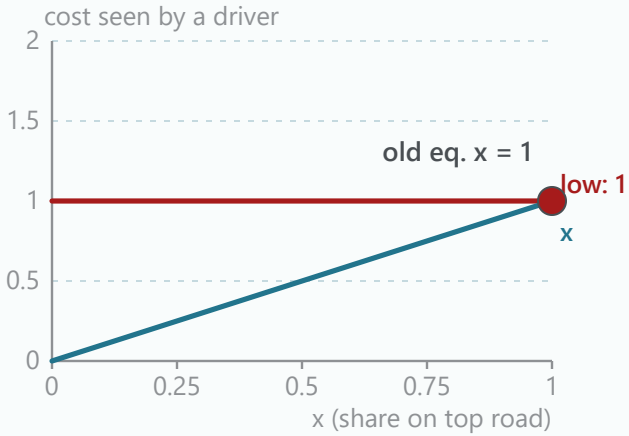
# Step by Step: A Congestion Toll Shifts the Equilibrium




τ = 10: too high a toll overshoots: $x_1 \approx 1{,}333$, total again ≈ 71,667. The right toll equals the externality; it is not “the more the better”.

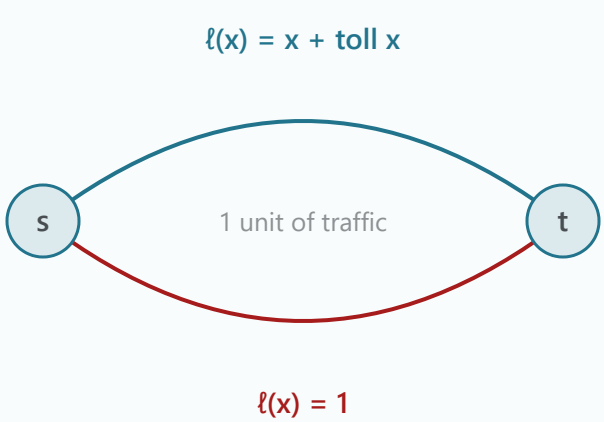
# Step by Step: A Congestion Toll Fixes Pigou

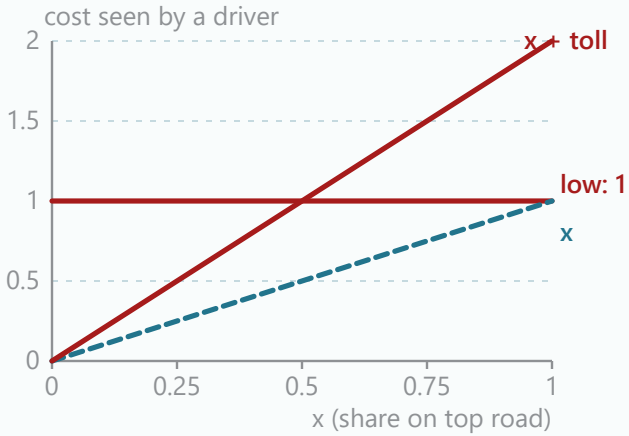



Start: no toll, everyone on top (x = 1), total delay 1. Goal: make the optimum (x = 1/2) an equilibrium.

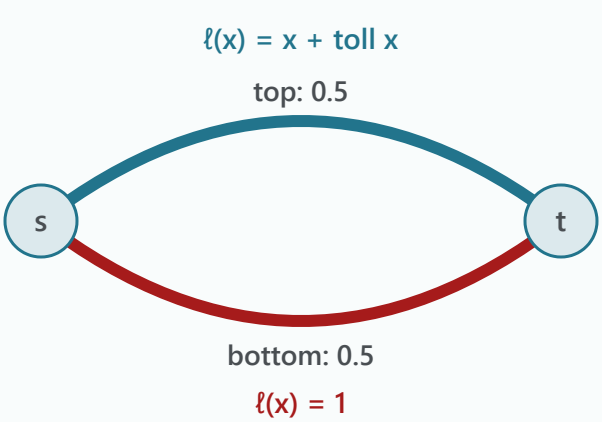
# Step by Step: A Congestion Toll Fixes Pigou

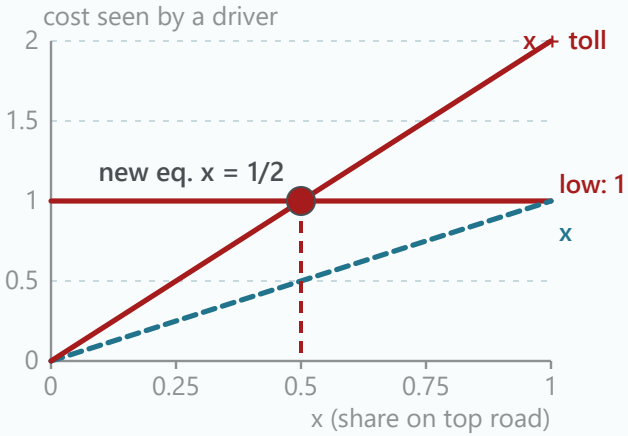



Marginal-cost toll $\tau_e = f_e \cdot \ell'_e(f_e)$: on top $\ell' = 1 \rightarrow \boldsymbol{\tau = x}$; on the bottom $\ell' = 0 \rightarrow$ no toll. A driver on top sees x + x = 2x.

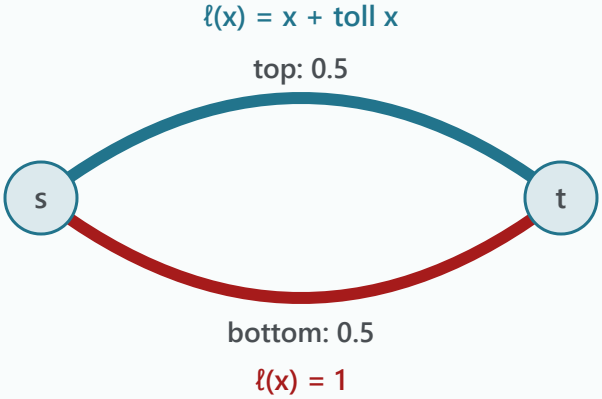
# Step by Step: A Congestion Toll Fixes Pigou

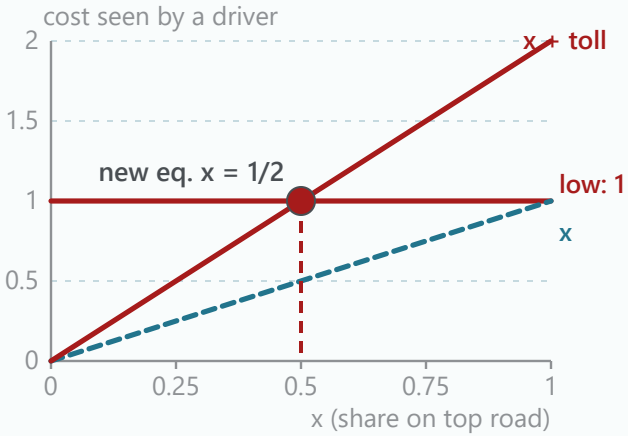



New equilibrium: 2x = 1 → **x = 1/2**. Selfish drivers choose the optimal split on their own; delay 3/4.

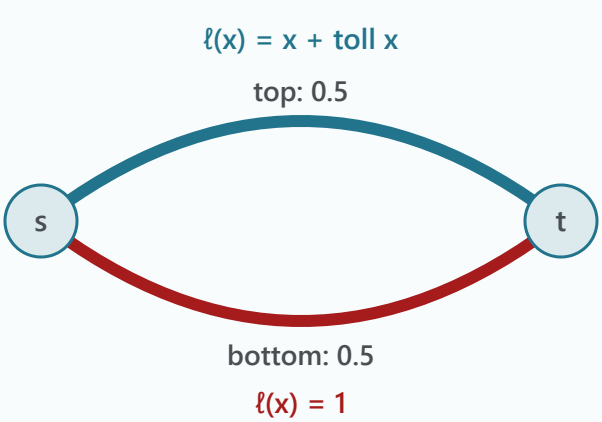
# Step by Step: A Congestion Toll Fixes Pigou

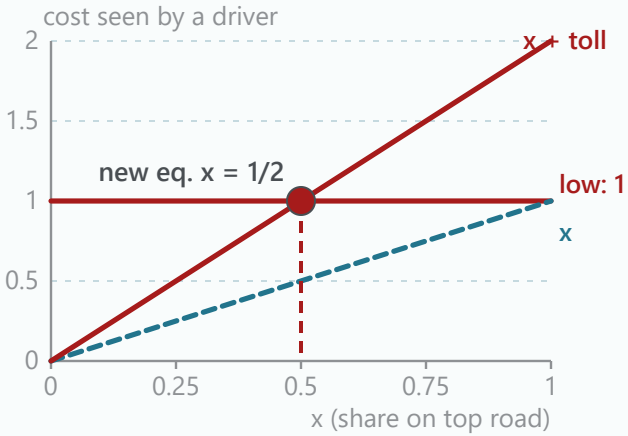



Tolls collected: 1/2 · 1/2 = 1/4. Drivers pay 3/4 + 1/4 = 1 in total: unless the revenue is returned, drivers gain nothing and the gain goes to society.

# Step by Step: A Congestion Toll Fixes Pigou




A fixed toll gives the same result: $\tau = 1/2$ on top → $x + 1/2 = 1$ → $x = 1/2$. General theorem: marginal-cost tolls make the optimum an equilibrium in *every* network.

# Other Ways to Reduce the PoA

### Congestion pricing

Brings the external cost users impose on others into their decision (London, Stockholm, Singapore).

### Central (Stackelberg) routing

A center routes part of the traffic, the rest choose selfishly; as the controlled share grows, the ratio approaches 1.

### Capacity planning

Roughgarden–Tardos: the selfish equilibrium for r units of traffic costs no more than the optimal routing of 2r units. Adding capacity can substitute for tolls.

### Route advice and information

Recommendations can steer the equilibrium to a lower total cost, but compliance is voluntary.

### Mechanism design

Set the rules so that the good outcome is the unique equilibrium.

### Counting the cost

Weigh an intervention's information, enforcement and fairness costs against the efficiency it gains.

# Real Life and Discussion

Toll lanes in traffic networks, inter-domain route selection on the Internet, time-of-use tariffs in power grids: the question is the same in all three. How bad is the selfish equilibrium, and which intervention brings it closer to the optimum?

### Discussion questions

- Why don't individual interests always reach the social optimum? What externality is at work in Pigou's example?
- Which incentives could reduce the PoA (tolls, bonuses, central routing)? Which is fairer?
- Where might Braess's paradox show up in daily life? When does closing a road make sense?

REVIEW

# Summary

| Example | Equilibrium | Optimum | Ratio |
|---|---|---|---|
| Two routes, 3,000 cars | 75,000 | 71,250 | PoA ≈ 1.053 |
| Pigou (x and 1) | 1 | 3/4 | PoA = 4/3 |
| Linear latency, any network | — | — | PoA ≤ 4/3 (tight) |
| Pigou, $\ell = x^p$ | 1 | → 0 | PoA → ∞ |
| Atomic congestion, linear | — | — | PoA ≤ 5/2 (tight) |
| Braess, 4,000 cars | 320,000 | 258,750 | PoA ≈ 1.237 |
| Load balancing {2, 2, 1, 1} | 4 (worst) | 3 | PoA = 4/3, PoS = 1 |
| Fair sharing, $H_n$ example | 25/12 | 1.1 | PoS ≈ 1.894 (≤ $H_n$) |
| Pigou + toll $\tau = x$ | 3/4 | 3/4 | 1 |

REVIEW

# Sample Problems

1. In Pigou's network let the top road's delay be $x^2$. What are the equilibrium cost, the optimum and the PoA?
2. Jobs {3, 3, 2, 2, 2}, 2 identical machines. What is the optimal makespan? Find a Nash equilibrium with makespan 7.
3. What is the PoA in the normalized Braess network (s→A: x, A→t: 1, s→B: 1, B→t: x, A→B: 0; 1 unit of traffic)?
4. Build the $H_n$ example for n = 3 (small ε). What does the PoS approach?
5. In a Pigou network with top road 2x and bottom road 1, what is the PoA, and what fixed toll makes the optimum an equilibrium?

▶ **Show answers**

REVIEW

# Exercises: Traces and New Topics

1. What does `pigou_poa(3)` return? ($x^{OPT} = 4^{-1/3}$)
2. Jobs (4, 3, 3, 2) start on M1; each step, the first job that gains moves to the least loaded machine. Final makespan? Optimum?
3. (1, ½)-smooth: what PoA bound? Compare with Pigou's (1, ¼).
4. PoA ≠ PoS game, A costs 3 TL, B 8 TL, 4 players: equilibria, PoA, PoS?
5. Verify $y(z + 1) \le 5/3 \cdot y^2 + 1/3 \cdot z^2$ for $y = 2$, $z = 3$.
6. What does `poa_pos([3, 2, 2, 1], 2)` return?

▶ **Show answers**

# References


- T. Roughgarden, *Twenty Lectures on Algorithmic Game Theory*, Cambridge University Press, 2016 (Lectures 11–15).
- N. Nisan, T. Roughgarden, É. Tardos, V. Vazirani (eds.), *Algorithmic Game Theory*, Cambridge University Press, 2007 (Chapters 17–20).
- E. Koutsoupias, C. Papadimitriou, "Worst-case Equilibria", *STACS*, 1999.
- T. Roughgarden, É. Tardos, "How Bad Is Selfish Routing?", *Journal of the ACM*, 2002.
- T. Roughgarden, "The Price of Anarchy Is Independent of the Network Topology", *J. Computer and System Sciences*, 2003.
- E. Anshelevich et al., "The Price of Stability for Network Design with Fair Cost Allocation", *FOCS*, 2004.
- T. Roughgarden, "Intrinsic Robustness of the Price of Anarchy", *STOC*, 2009.
- G. Christodoulou, E. Koutsoupias, "The Price of Anarchy of Finite Congestion Games", *STOC*, 2005.
- B. Awerbuch, Y. Azar, A. Epstein, "The Price of Routing Unsplittable Flow", *STOC*, 2005.
- A. C. Pigou, *The Economics of Welfare*, Macmillan, 1920.
- A. R. Karlin, Y. Peres, *Game Theory, Alive*, AMS, 2017 (Chapter 8: Price of Anarchy); D. Easley, J. Kleinberg, *Networks, Crowds, and Markets*, Cambridge, 2010 (Chapter 8: traffic and Braess).
- Course page: **T. Roughgarden, CS364A (Stanford, 2013)**, notes for Lectures 11–15.

# References (2/2)


- C. Papadimitriou, “Algorithms, Games, and the Internet”, *STOC*, 2001.
- R. L. Graham, “Bounds for Certain Multiprocessing Anomalies”, *Bell System Technical Journal* 45(9), 1966.
- G. Finn, E. Horowitz, “A Linear Time Approximation Algorithm for Multiprocessor Scheduling”, *BIT* 19, 1979.
- E. Koutsoupias, M. Mavronicolas, P. Spirakis, “Approximate Equilibria and Ball Fusion”, *Theory of Computing Systems* 36(6), 2003.
- A. Czumaj, B. Vöcking, “Tight Bounds for Worst-Case Equilibria”, *SODA*, 2002.
- T. Roughgarden, “Stackelberg Scheduling Strategies”, *SIAM Journal on Computing* 33(2), 2004.

# The End

## Chapter 9: The Price of Anarchy

Previous: Chapter 8: Network Design and Routing · Next: Chapter 10: Multi-Agent Systems