Chapter 8: Network Design and Routing

Algorithmic Game Theory

Routing basics | Selfish routing | Wardrop equilibrium | Braess's paradox | Congestion games | Rosenthal's potential | Network formation games

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_08_Network_Design_and_Routing.html

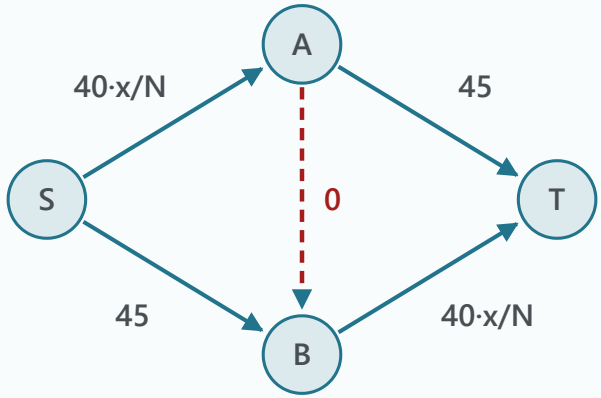
Warm-up: Braess's Paradox

The rules

Each driver picks their own fastest route from S to T; travel time on a narrow road grows with the number of drivers on it, while a wide road always takes 45 minutes.

Think about it

- If a zero-time shortcut from A to B opens, do trips get shorter?
- When everyone takes their own fastest route, is the average time minimized?



N drivers S → T; x: drivers on that road
dashed: new shortcut A → B

Learning Outcomes

Find a Wardrop equilibrium

Compute the equilibrium of a two-route network by equalizing delays; find the system optimum via marginal costs and compare.

Explain Braess's paradox

Show with numbers why a new road can make everyone slower.

Reason with potentials

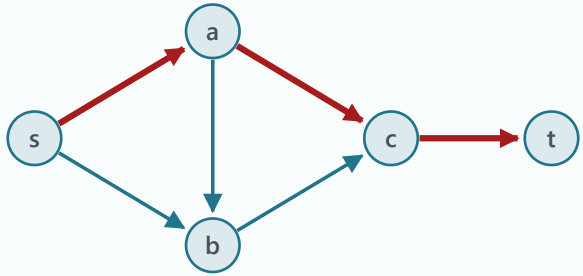
Compute Rosenthal's potential; prove that best-response dynamics must stop.

Model network formation

Check which networks are equilibria when players build the links.

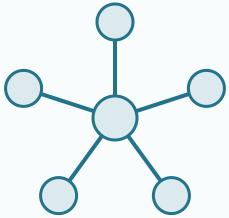
Network Design and Routing

- **Network design** decides the nodes, links, capacities and costs.
- **Routing** decides which paths the flow takes from source to destination.
- Algorithmic game theory studies traffic and communication networks whose users each try to reduce *their own* cost.
- Key applications: city traffic, Internet backbones, power grids.

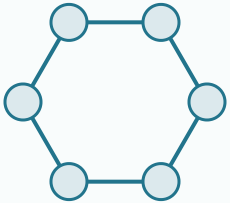


Design: which edges, what capacity?
Routing: which path for the flow? (red)

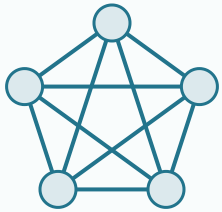
Basic Network Design Concepts



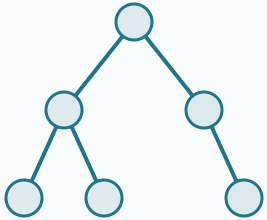
Star



Ring



Full mesh



Tree

Nodes and links

Node: an intersection, router or substation. Link: a road, fiber line or transmission line.

Capacity and bandwidth

The flow a link can carry per unit time; beyond it, delay rises sharply.

Cost

Building cost (design) and usage cost: delay, energy, tolls (routing).

Example

Tree/layered topology in a data center; a near-mesh road network in a city.

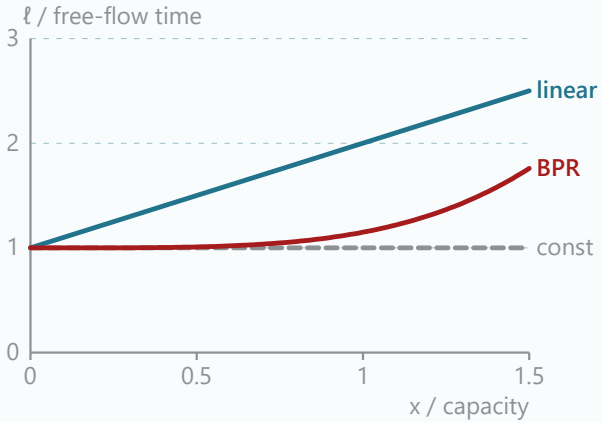
Classic Routing Algorithms

Method	What it does	Note
Dijkstra	Single-source shortest paths with nonnegative weights	O((V + E) log V); the core of OSPF (Open Shortest Path First, a link-state protocol)
Bellman–Ford	Allows negative weights; updates by learning from neighbors	O(V·E); its distributed version is RIP (a distance-vector protocol)
Dynamic routing	Recomputes paths when link states and costs change	Adapts to traffic and failures
Load balancing	Spreads traffic over several paths or servers	E.g. ECMP (equal-cost multipath)

Classic algorithms assume **fixed** edge weights. In reality delay depends on how many people choose a path; when strategic users can deviate from the suggested route, the network equilibrium and the system optimum diverge. This turns routing into a **game**.

Delay Depends on Load: $\ell_e(x)$

- Each edge's delay is a **nondecreasing** function of its flow x: $\ell_e(x)$.
- **Constant**: a wide highway, $\ell = b$.
- **Linear**: $\ell = a \cdot x + b$ (the two-route and Braess examples).
- **BPR** (US Bureau of Public Roads): $\ell = t_0(1 + 0.15\,(x/c)^4)$; t_0: free-flow time, c: road capacity. Standard in traffic planning.
- Queueing delay in networks: $\ell = 1/(c - x)$, only for $\mathbf{0 \le x < c}$ (single-server queue, M/M/1; c: packets served per second, x: packets arriving per second). As x approaches c, delay grows without bound; for $x \ge c$ the queue grows forever.



Selfish Routing

- Each user picks the route that minimizes their own delay.
- With many small users, flow is treated as **splittable** (nonatomic): a single user's effect is negligible.
- Individual route choices can drift away from the **system optimum**, which minimizes total delay.
- In Braess's paradox, adding a link can increase the equilibrium delay.

Model

r units of traffic $s \to t$

f_P: flow on path P

f_e: total flow on edge e

$\ell_P(f) = \Sigma_{e \in P} \ell_e(f_e)$

Total delay

$C(f) = \Sigma_e f_e \cdot \ell_e(f_e)$

A path's delay is the sum of its edges' delays; total delay is flow × delay summed over edges.

Wardrop's Principles (1952)

1st principle: user equilibrium

+ All used paths have equal, minimal delay.
+ Unused paths are no faster.
+ No single user can cut their travel time by switching.

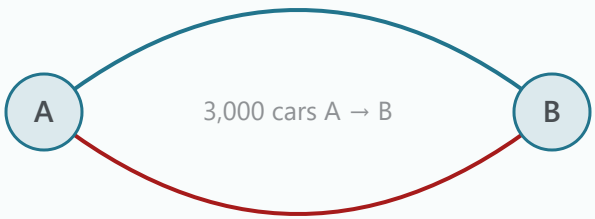
2nd principle: system optimum

– Average (total) delay is minimal.
– The split a central planner would choose.
– Some users may be “sacrificed” to a slower road.

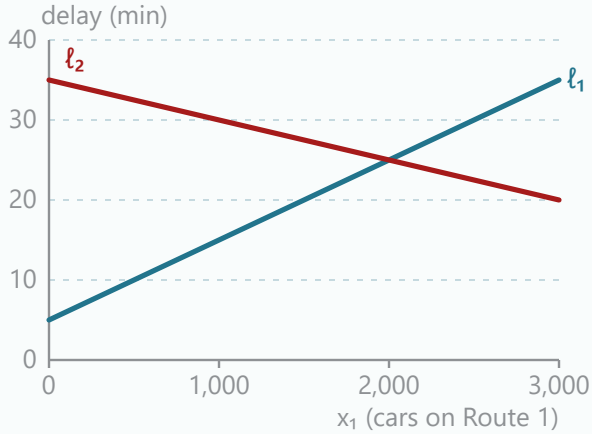
In short: no used path is slower than any other path. Equilibrium explains users' individual choices, but it need not match the system optimum. With linear delays an equilibrium exists, and every equilibrium gives each edge the same delay.

Step by Step: Wardrop Equilibrium on Two Routes

Route 1: $\ell_1 = 5 + x_1/100$



Route 2: $\ell_2 = 20 + x_2/200$

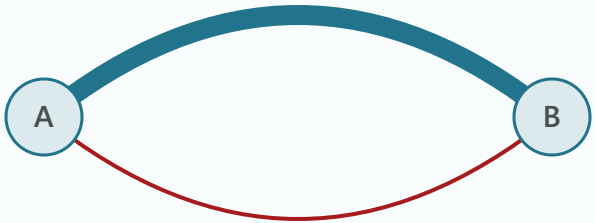


Setup: 3,000 cars travel from A to B. Route 1 is short but narrow (delay rises fast); Route 2 is long but wide.

Step by Step: Wardrop Equilibrium on Two Routes

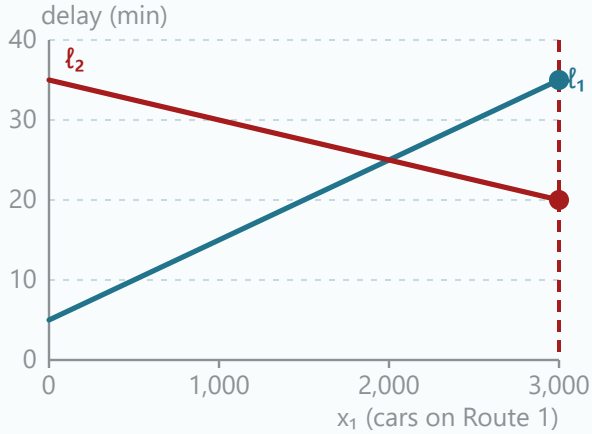
Route 1: $\ell_1 = 5 + x_1/100$

x_1 = 3,000 cars → 35 min



x_2 = 0 cars → 20 min

Route 2: $\ell_2 = 20 + x_2/200$

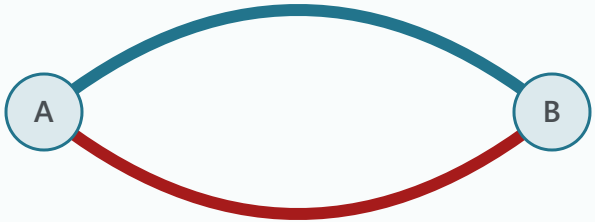


Try 1: everyone on Route 1 → ℓ_1 = 35 min, while the empty Route 2 takes 20. Route 1 drivers **want to switch**: not an equilibrium.

Step by Step: Wardrop Equilibrium on Two Routes

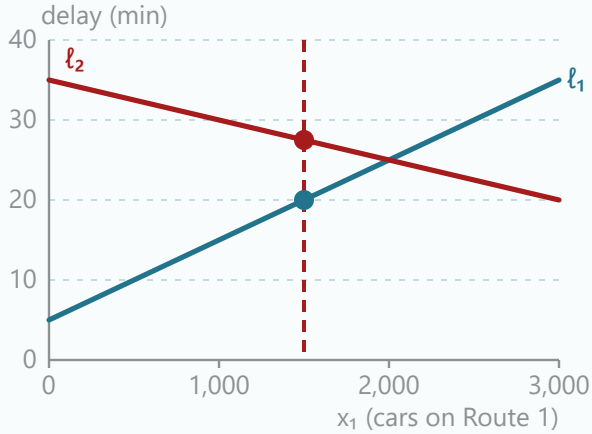
Route 1: $\ell_1 = 5 + x_1/100$

x_1 = 1,500 cars → 20 min



x_2 = 1,500 cars → 27.5 min

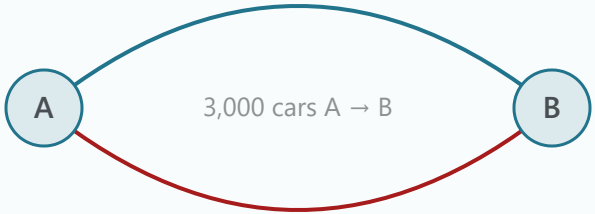
Route 2: $\ell_2 = 20 + x_2/200$



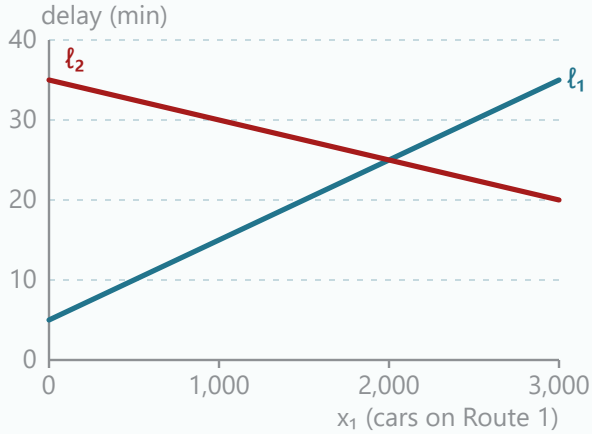
Try 2: 1,500 / 1,500 → $\ell_1 = 20$, $\ell_2 = 27.5$ min. Now Route 2 drivers move to Route 1: still not an equilibrium.

Step by Step: Wardrop Equilibrium on Two Routes

Route 1: $\ell_1 = 5 + x_1/100$



Route 2: $\ell_2 = 20 + x_2/200$

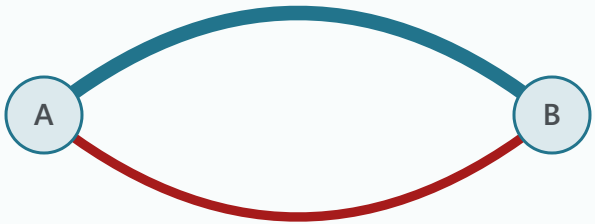


Equilibrium condition: if both routes are used, their delays must be equal: $5 + x_1/100 = 20 + (3000 - x_1)/200$.

Step by Step: Wardrop Equilibrium on Two Routes

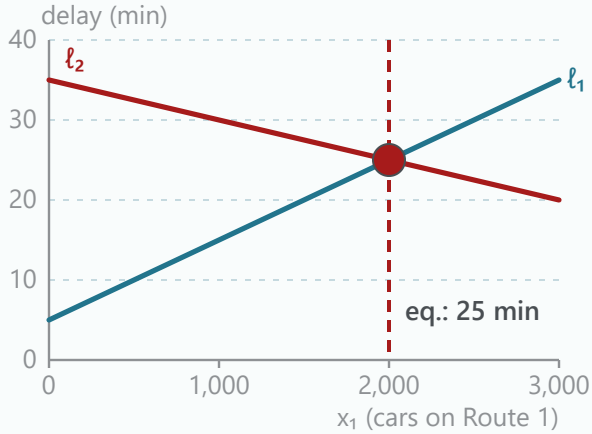
Route 1: $\ell_1 = 5 + x_1/100$

x_1 = 2,000 cars → 25 min



x_2 = 1,000 cars → 25 min

Route 2: $\ell_2 = 20 + x_2/200$

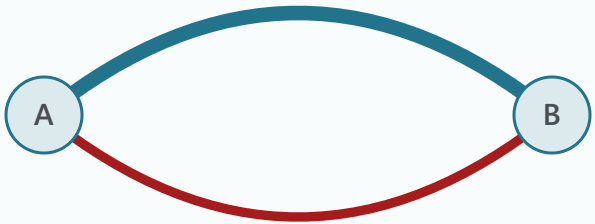


×200: $1000 + 2x_1 = 4000 + 3000 - x_1 \rightarrow 3x_1 = 6000 \rightarrow$ **$x_1 = 2{,}000$, $x_2 = 1{,}000$**; both take 25 min.

Step by Step: Wardrop Equilibrium on Two Routes

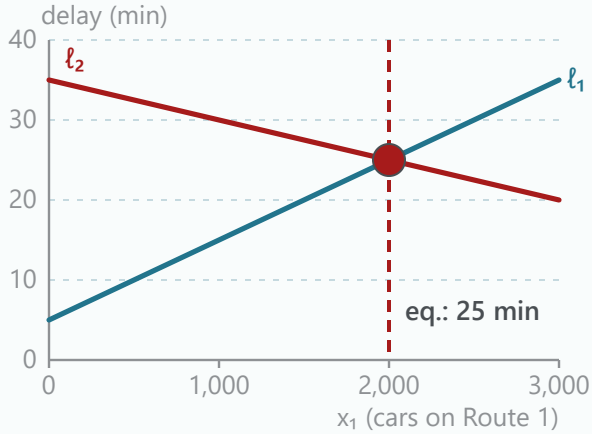
Route 1: $\ell_1 = 5 + x_1/100$

x_1 = 2,000 cars → 25 min



x_2 = 1,000 cars → 25 min

Route 2: $\ell_2 = 20 + x_2/200$

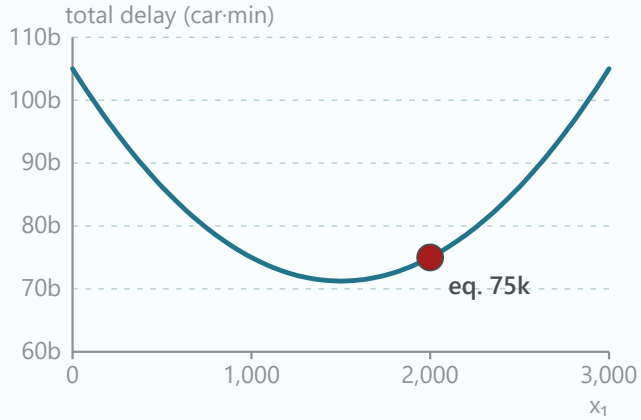


Check: a driver who switches still faces ≈ 25 min → nobody gains. Total delay 3,000 × 25 = **75,000 car·min**.

Equilibrium Is Not the System Optimum

Total delay

$C = x_1 \cdot \ell_1(x_1) + x_2 \cdot \ell_2(x_2)$



At the equilibrium $x_1 = 2{,}000$ the total is 75,000 car·min. Is this the bottom of the curve?

Equilibrium Is Not the System Optimum

Total delay

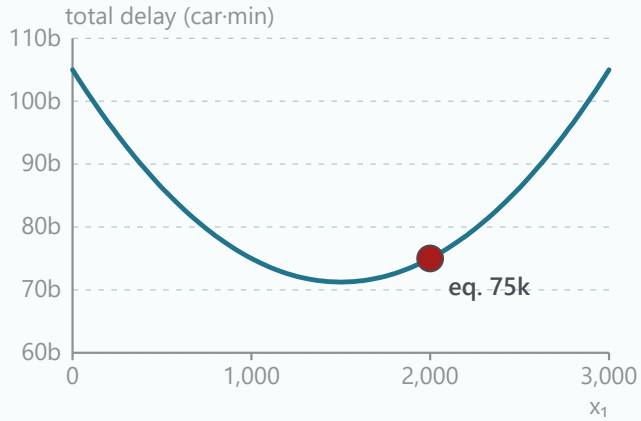
$C = x_1 \cdot \ell_1(x_1) + x_2 \cdot \ell_2(x_2)$

Marginal cost

: the extra total delay from adding one more car

Route 1: $x_1 \cdot \ell_1 = 5x_1 + x_1^2/100$ → derivative $5 + x_1/50$

Route 2: $x_2 \cdot \ell_2 = 20x_2 + x_2^2/200$ → derivative $20 + x_2/100$



One more car adds its own delay plus the $x \cdot \ell'$ delay it imposes on *everyone else*.

Equilibrium Is Not the System Optimum

Total delay

$C = x_1 \cdot \ell_1(x_1) + x_2 \cdot \ell_2(x_2)$

Marginal cost

: the extra total delay from adding one more car

Route 1: $x_1 \cdot \ell_1 = 5x_1 + x_1^2/100$ → derivative $5 + x_1/50$

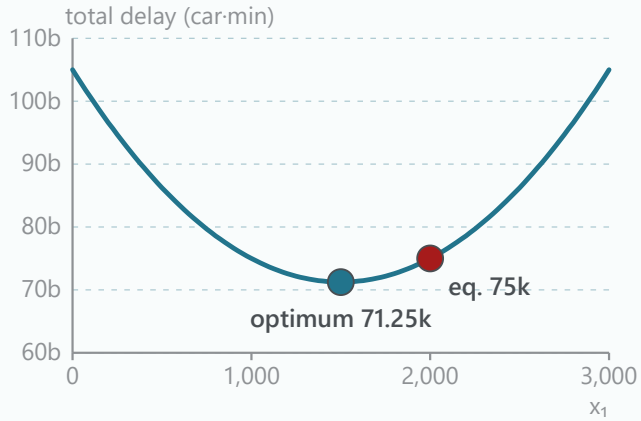
Route 2: $x_2 \cdot \ell_2 = 20x_2 + x_2^2/200$ → derivative $20 + x_2/100$

Equate:

$5 + x_1/50 = 20 + (3000 - x_1)/100$

$\rightarrow 3x_1 = 4500 \rightarrow$

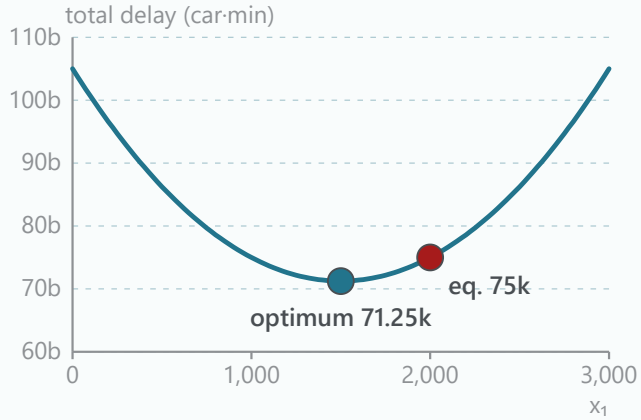
$x_1 = x_2 = 1{,}500$



At the optimum, marginal costs are equal: traffic splits 1,500 / 1,500, total 71,250.

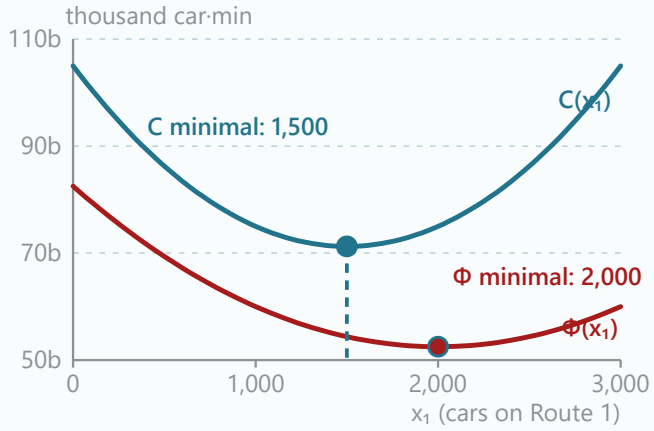
Equilibrium Is Not the System Optimum

	x_1 / x_2	ℓ_1 / ℓ_2 (min)	Total
Equilibrium	2,000 / 1,000	25 / 25	75,000
Optimum	1,500 / 1,500	20 / 27.5	71,250



At the optimum, Route 1 drivers take 20 min and Route 2 drivers 27.5; Route 2 drivers want to leave → **the optimum is not an equilibrium**.

Equilibrium and Optimum: Two Different Minimizations



Wardrop equilibrium

Minimizes the Beckmann potential (Beckmann, McGuire, Winsten, 1956):

$\Phi(f) = \Sigma_e \int_0^{f_e} \ell_e(t)\, dt$

Derivative ℓ_e: the delay seen by the "last driver in".

System optimum

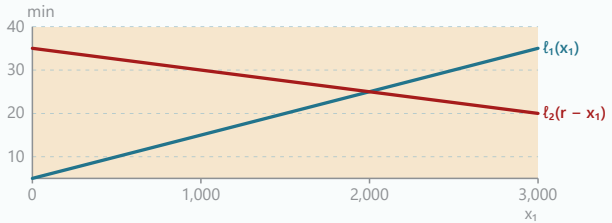
Minimizes total delay:

$C(f) = \Sigma_e\, f_e \cdot \ell_e(f_e)$

Derivative $\ell_e + f_e \cdot \ell'_e$: the marginal cost.

The two curves bottom out in different places: Φ at x_1 = 2,000 (equilibrium, Φ = 52,500), C at x_1 = 1,500 (optimum, C = 71,250). The gap comes from one term: a driver ignores the **x·ℓ′** externality their presence imposes on others.

Code Trace: `find_eq(l1, l2, r)`



lo = 0
hi = 3,000
width = 3,000

Call: the two-route network ($\ell_1 = 5 + x/100$, $\ell_2 = 20 + x/200$), 3,000 cars. Goal: the x with $\ell_1(x) = \ell_2(r - x)$.

Step 1 / 9

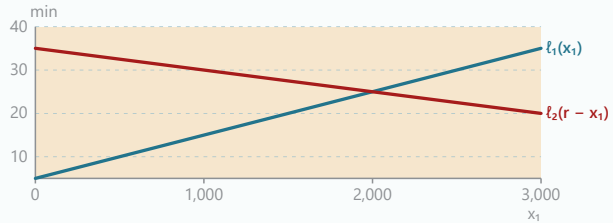
```
▶ def find_eq(l1, l2, r, eps=50):
2   if l1(0) >= l2(r): return 0   # route 1 unused
3   if l1(r) <= l2(0): return r   # all on route 1
4   lo, hi = 0, r
5   while hi - lo > eps:
6     x = (lo + hi) / 2
7     if l1(x) < l2(r - x):
8       lo = x      # route 1 faster: raise
9     else:
10      hi = x      # route 1 slower: lower
11  return (lo + hi) / 2
```

VARIABLES

`r = 3000` `eps = 50`

Since $\ell_1(x) - \ell_2(r - x)$ is nondecreasing, its zero is the equilibrium; the first two lines catch the corner cases. In general networks the Beckmann potential is minimized by convex optimization (e.g. Frank–Wolfe).

Code Trace: `find_eq(l1, l2, r)`



lo = 0
hi = 3,000
width = 3,000

Search interval [0, 3,000]: the equilibrium must lie here (at x = 0 Route 1 is faster, at x = r slower).

Step 2 / 9

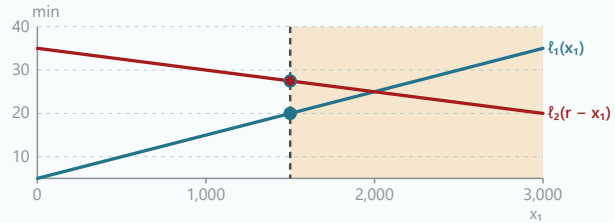
```
1  def find_eq(l1, l2, r, eps=50):
▶    if l1(0) >= l2(r): return 0   # route 1 unused
▶    if l1(r) <= l2(0): return r   # all on route 1
▶    lo, hi = 0, r
5    while hi - lo > eps:
6      x = (lo + hi) / 2
7      if l1(x) < l2(r - x):
8        lo = x      # route 1 faster: raise
9      else:
10       hi = x      # route 1 slower: lower
11   return (lo + hi) / 2
```

VARIABLES

`r = 3000` `eps = 50` `lo = 0` `hi = 3000`

Since $\ell_1(x) - \ell_2(r - x)$ is nondecreasing, its zero is the equilibrium; the first two lines catch the corner cases. In general networks the Beckmann potential is minimized by convex optimization (e.g. Frank–Wolfe).

Code Trace: `find_eq(l1, l2, r)`



lo = 1,500
hi = 3,000
width = 1,500
x = 1,500
$\ell_1 = 20 \cdot \ell_2 = 27.5$

Round 1: x = 1,500 → $\ell_1 = 20 < \ell_2 = 27.5$. Route 1 is still faster, so more cars move to it: **lo = 1,500**.

Step 3 / 9

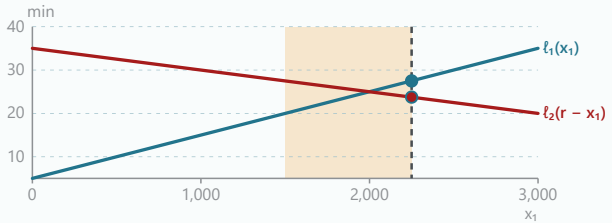
```
1  def find_eq(l1, l2, r, eps=50):
2    if l1(0) >= l2(r): return 0   # route 1 unused
3    if l1(r) <= l2(0): return r   # all on route 1
4    lo, hi = 0, r
5    while hi - lo > eps:
▶      x = (lo + hi) / 2
▶      if l1(x) < l2(r - x):
▶        lo = x      # route 1 faster: raise
9      else:
10       hi = x      # route 1 slower: lower
11   return (lo + hi) / 2
```

VARIABLES

`r = 3000` `eps = 50` `lo = 1500` `hi = 3000` `x = 1500`

Since $\ell_1(x) - \ell_2(r - x)$ is nondecreasing, its zero is the equilibrium; the first two lines catch the corner cases. In general networks the Beckmann potential is minimized by convex optimization (e.g. Frank–Wolfe).

Code Trace: `find_eq(l1, l2, r)`



lo = 1,500
hi = 2,250
width = 750
x = 2,250
$\ell_1 = 27.5 \cdot \ell_2 = 23.75$

Round 2: x = 2,250 → $\ell_1 = 27.5 \geq \ell_2 = 23.75$. Route 1 is too crowded: **hi = 2,250**.

Step 4 / 9

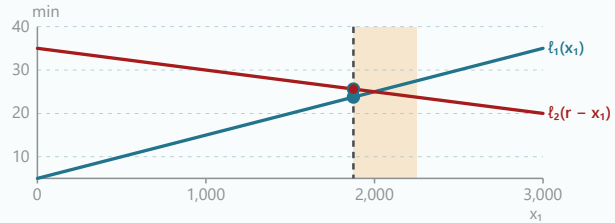
```
1  def find_eq(l1, l2, r, eps=50):
2    if l1(0) >= l2(r): return 0   # route 1 unused
3    if l1(r) <= l2(0): return r   # all on route 1
4    lo, hi = 0, r
5    while hi - lo > eps:
▶      x = (lo + hi) / 2
▶      if l1(x) < l2(r - x):
8        lo = x      # route 1 faster: raise
9      else:
▶        hi = x      # route 1 slower: lower
11   return (lo + hi) / 2
```

VARIABLES

`r = 3000` `eps = 50` `lo = 1500` `hi = 2250` `x = 2250`

Since $\ell_1(x) - \ell_2(r - x)$ is nondecreasing, its zero is the equilibrium; the first two lines catch the corner cases. In general networks the Beckmann potential is minimized by convex optimization (e.g. Frank–Wolfe).

Code Trace: `find_eq(l1, l2, r)`



lo = 1,875
hi = 2,250
width = 375
x = 1,875
$\ell_1 = 23.75 \cdot \ell_2 = 25.63$

Round 3: x = 1,875 → ℓ_1 = 23.75 < ℓ_2 = 25.63. Route 1 is still faster, so more cars move to it: **lo = 1,875**.

Step 5 / 9

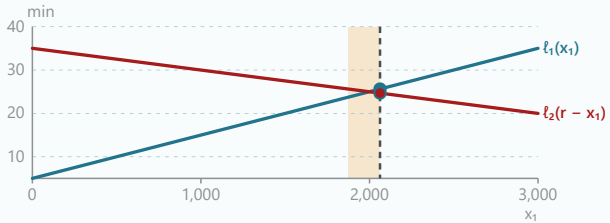
```
1  def find_eq(l1, l2, r, eps=50):
2    if l1(0) >= l2(r): return 0   # route 1 unused
3    if l1(r) <= l2(0): return r   # all on route 1
4    lo, hi = 0, r
5    while hi - lo > eps:
▶      x = (lo + hi) / 2
▶      if l1(x) < l2(r - x):
▶        lo = x      # route 1 faster: raise
9      else:
10       hi = x      # route 1 slower: lower
11   return (lo + hi) / 2
```

VARIABLES

`r = 3000` `eps = 50` `lo = 1875` `hi = 2250` `x = 1875`

Since $\ell_1(x) - \ell_2(r - x)$ is nondecreasing, its zero is the equilibrium; the first two lines catch the corner cases. In general networks the Beckmann potential is minimized by convex optimization (e.g. Frank–Wolfe).

Code Trace: `find_eq(l1, l2, r)`



lo = 1,875
hi = 2,062.5
width = 187.5
x = 2,062.5
$\ell_1 = 25.63 \cdot \ell_2 = 24.69$

Round 4: x = 2,062.5 → ℓ_1 = 25.63 ≥ ℓ_2 = 24.69. Route 1 is too crowded: **hi = 2,062.5**.

Step 6 / 9

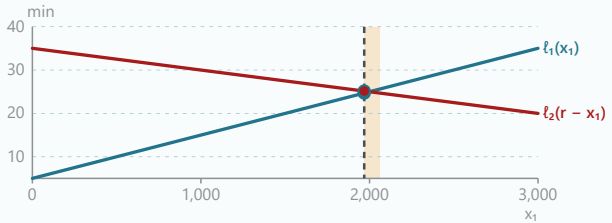
```
1  def find_eq(l1, l2, r, eps=50):
2    if l1(0) >= l2(r): return 0   # route 1 unused
3    if l1(r) <= l2(0): return r   # all on route 1
4    lo, hi = 0, r
5    while hi - lo > eps:
▶      x = (lo + hi) / 2
▶      if l1(x) < l2(r - x):
8        lo = x      # route 1 faster: raise
9      else:
▶        hi = x      # route 1 slower: lower
11   return (lo + hi) / 2
```

VARIABLES

`r = 3000` `eps = 50` `lo = 1875` `hi = 2062.5` `x = 2062.5`

Since $\ell_1(x) - \ell_2(r - x)$ is nondecreasing, its zero is the equilibrium; the first two lines catch the corner cases. In general networks the Beckmann potential is minimized by convex optimization (e.g. Frank–Wolfe).

Code Trace: `find_eq(l1, l2, r)`



lo = 1,968.75
hi = 2,062.5
width = 93.75
x = 1,968.75
$\ell_1 = 24.69 \cdot \ell_2 = 25.16$

Round 5: x = 1,968.75 → $\ell_1 = 24.69 < \ell_2 = 25.16$. Route 1 is still faster, so more cars move to it: **lo = 1,968.75**.

Step 7 / 9

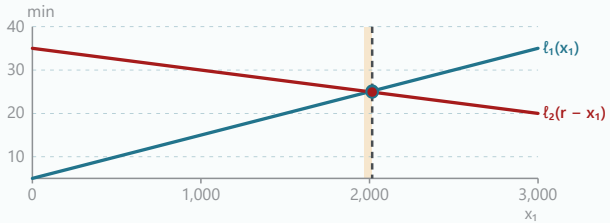
```
1  def find_eq(l1, l2, r, eps=50):
2    if l1(0) >= l2(r): return 0   # route 1 unused
3    if l1(r) <= l2(0): return r   # all on route 1
4    lo, hi = 0, r
5    while hi - lo > eps:
▶      x = (lo + hi) / 2
▶      if l1(x) < l2(r - x):
▶        lo = x      # route 1 faster: raise
9      else:
10       hi = x      # route 1 slower: lower
11   return (lo + hi) / 2
```

VARIABLES

`r = 3000` `eps = 50` `lo = 1968.75` `hi = 2062.5` `x = 1968.75`

Since $\ell_1(x) - \ell_2(r - x)$ is nondecreasing, its zero is the equilibrium; the first two lines catch the corner cases. In general networks the Beckmann potential is minimized by convex optimization (e.g. Frank–Wolfe).

Code Trace: `find_eq(l1, l2, r)`



lo = 1,968.75
hi = 2,015.63
width = 46.88
x = 2,015.63
$\ell_1 = 25.16 \cdot \ell_2 = 24.92$

Round 6: x = 2,015.63 → ℓ_1 = 25.16 ≥ ℓ_2 = 24.92. Route 1 is too crowded: **hi = 2,015.63**.

Step 8 / 9

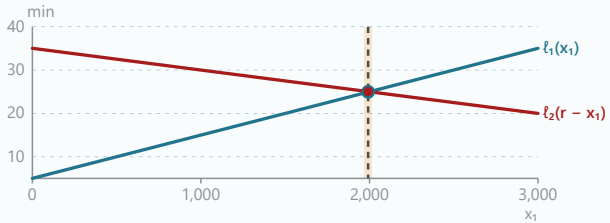
```
1  def find_eq(l1, l2, r, eps=50):
2    if l1(0) >= l2(r): return 0   # route 1 unused
3    if l1(r) <= l2(0): return r   # all on route 1
4    lo, hi = 0, r
5    while hi - lo > eps:
▶      x = (lo + hi) / 2
▶      if l1(x) < l2(r - x):
8        lo = x      # route 1 faster: raise
9      else:
▶        hi = x      # route 1 slower: lower
11   return (lo + hi) / 2
```

VARIABLES

`r = 3000` `eps = 50` `lo = 1968.75` `hi = 2015.625` `x = 2015.625`

Since $\ell_1(x) - \ell_2(r - x)$ is nondecreasing, its zero is the equilibrium; the first two lines catch the corner cases. In general networks the Beckmann potential is minimized by convex optimization (e.g. Frank–Wolfe).

Code Trace: `find_eq(l1, l2, r)`



lo = 1,968.75
hi = 2,015.63
width = 46.88
x = 1,992.19
$\ell_1 = 24.92 \cdot \ell_2 = 25.04$

Width 46.88 ≤ 50 → stop. Result ≈ **1,992.19** (true equilibrium 2,000). Each round halves the interval: after k rounds the error is ≤ $r/2^k$.

Step 9 / 9

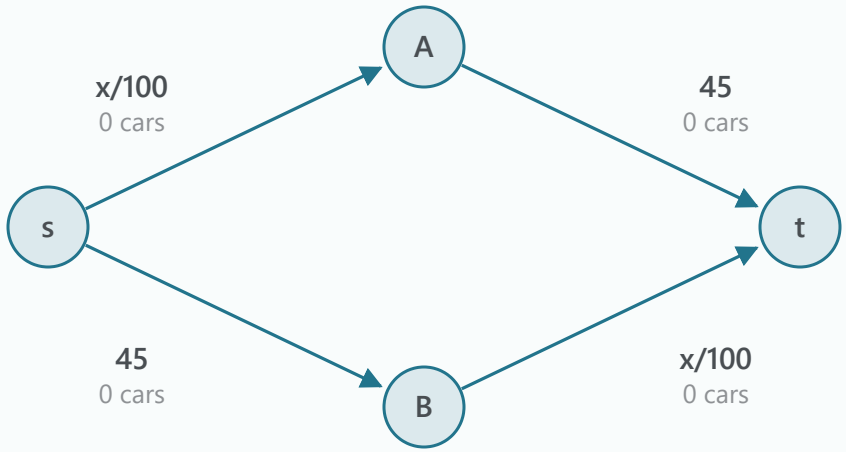
```
1  def find_eq(l1, l2, r, eps=50):
2    if l1(0) >= l2(r): return 0   # route 1 unused
3    if l1(r) <= l2(0): return r   # all on route 1
4    lo, hi = 0, r
5    while hi - lo > eps:
6      x = (lo + hi) / 2
7      if l1(x) < l2(r - x):
8        lo = x      # route 1 faster: raise
9      else:
10       hi = x      # route 1 slower: lower
▶    return (lo + hi) / 2
```

VARIABLES

`r = 3000` `eps = 50` `lo = 1968.75` `hi = 2015.625`

Since $\ell_1(x) - \ell_2(r - x)$ is nondecreasing, its zero is the equilibrium; the first two lines catch the corner cases. In general networks the Beckmann potential is minimized by convex optimization (e.g. Frank–Wolfe).

Step by Step: Braess's Paradox (4,000 Cars)



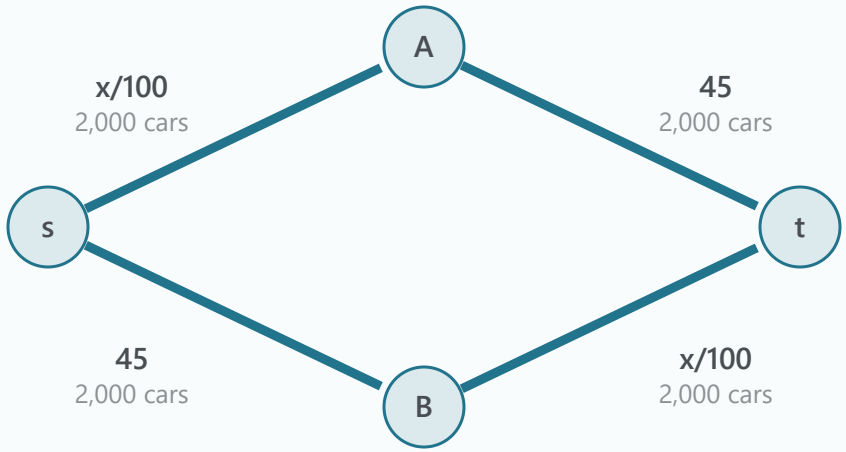
Path	cars	time
s→A→t	0	—
s→B→t	0	—
s→A→B→t	—	—

4,000 cars s → t

x: cars on the edge

Setup: 4,000 cars from s to t. s→A and B→t are narrow roads (x/100 min); A→t and s→B are wide highways (always 45 min).

Step by Step: Braess's Paradox (4,000 Cars)



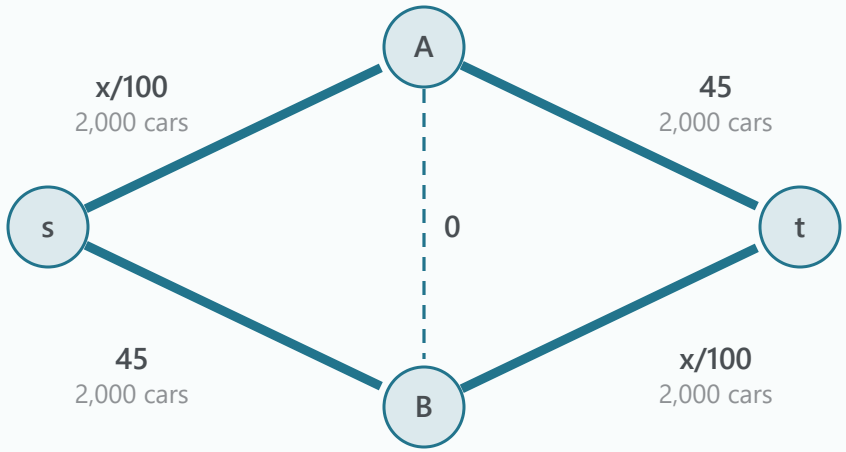
Path	cars	time
s→A→t	2,000	65 min
s→B→t	2,000	65 min
s→A→B→t	—	—

Average: 65 min

Total: 260,000 car·min

Symmetry: 2,000 / 2,000 → both paths take 2000/100 + 45 = **65 min**. Switching doesn't help: a Wardrop equilibrium (and the optimum).

Step by Step: Braess's Paradox (4,000 Cars)



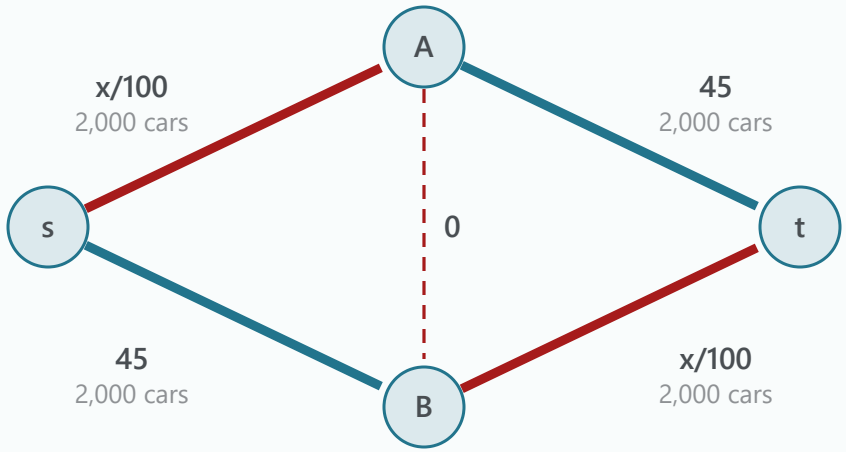
Path	cars	time
s→A→t	2,000	65 min
s→B→t	2,000	65 min
s→A→B→t	0	—

Average: 65 min

Total: 260,000 car·min

The city opens a very short, **0-minute** link road from A to B. At first nobody uses it.

Step by Step: Braess's Paradox (4,000 Cars)



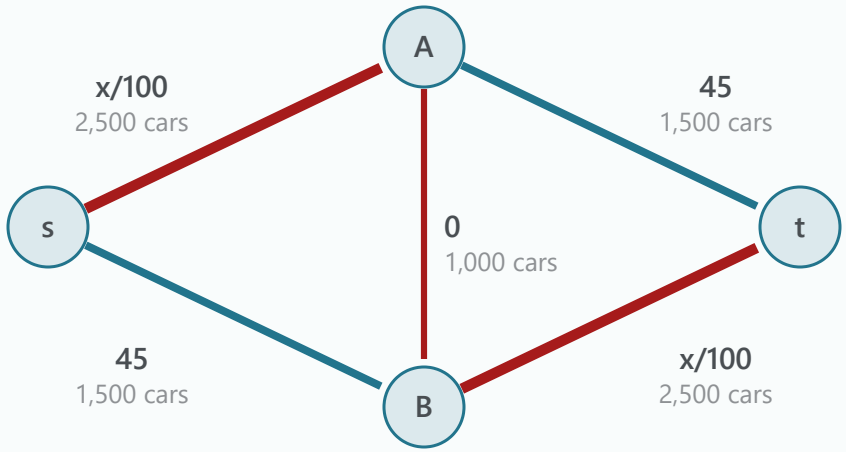
Path	cars	time
s→A→t	2,000	65 min
s→B→t	2,000	65 min
s→A→B→t	0	40 min

Zigzag: 20 + 0 + 20

= 40 < 65 min

One driver tries s→A→B→t: 20 + 0 + 20 = **40 min** < 65. Very profitable individually → they switch.

Step by Step: Braess's Paradox (4,000 Cars)



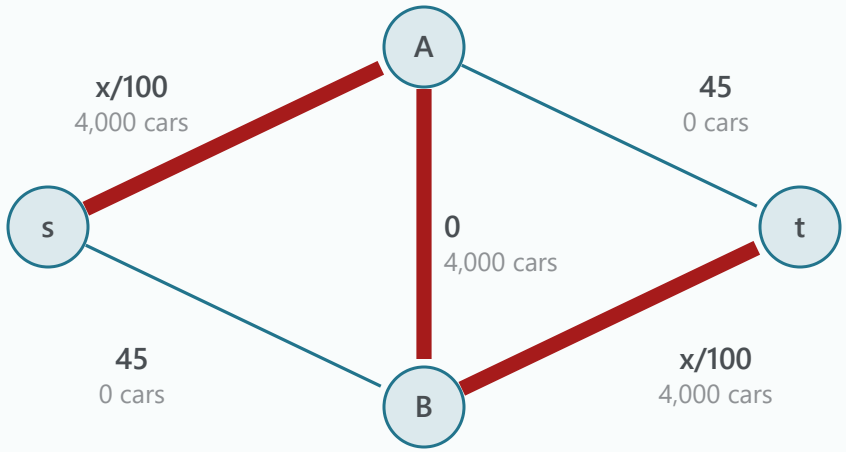
Path	cars	time
s→A→t	1,500	70 min
s→B→t	1,500	70 min
s→A→B→t	1,000	50 min

Zigzag still fastest

50 < 70 min

1,000 drivers have switched: 2,500 cars on the narrow roads → 25 min. Old paths 70, zigzag 50 min: **switching continues**.

Step by Step: Braess's Paradox (4,000 Cars)



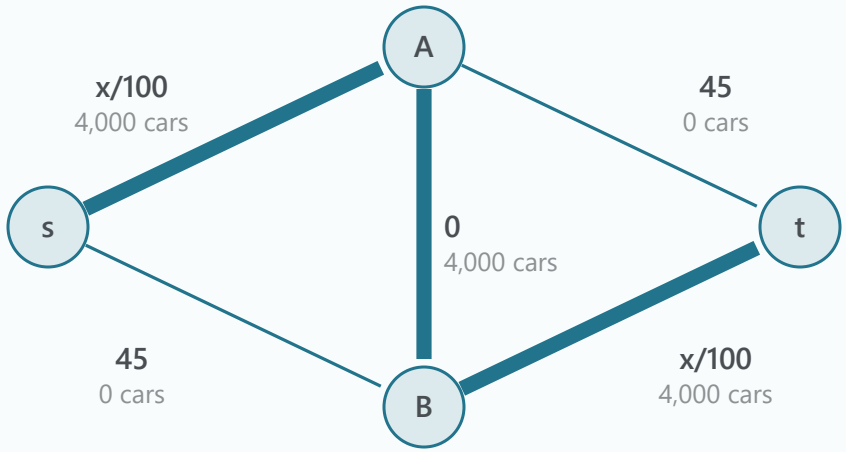
Path	cars	time
s→A→t	0	85 min
s→B→t	0	85 min
s→A→B→t	4,000	80 min

Average: 80 min

Total: 320,000 car·min

Everyone zigzags: 40 + 0 + 40 = **80 min**. Going back to an old path costs 40 + 45 = 85 min → nobody goes back: a new equilibrium.

Step by Step: Braess's Paradox (4,000 Cars)



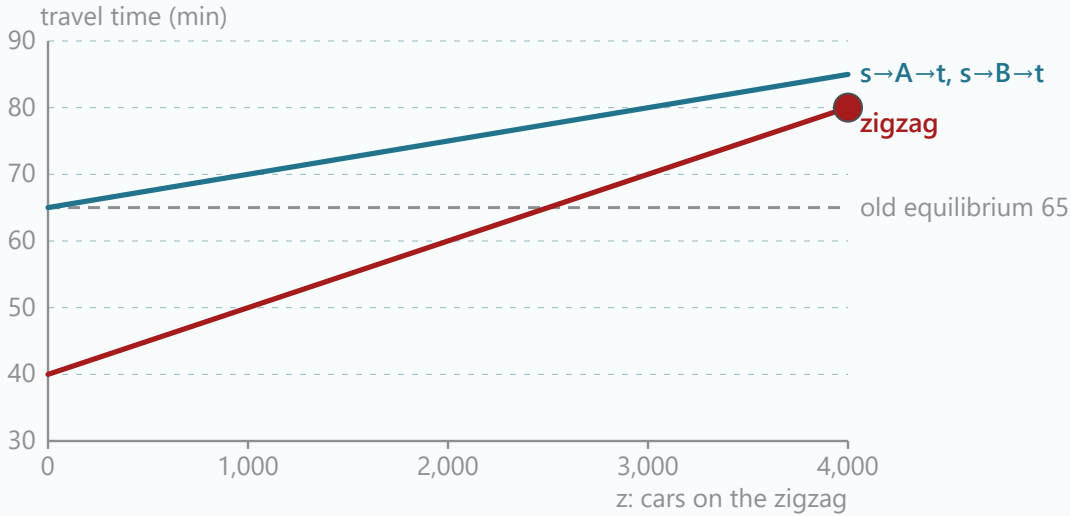
Path	cars	time
s→A→t	0	85 min
s→B→t	0	85 min
s→A→B→t	4,000	80 min

65 → 80 min

260k → 320k car·min

Result: a road was added and **everyone lost 15 min**. More capacity + selfish choice = a worse equilibrium. The system optimum with the shortcut is ≈ 64.7 min (only 500 cars on the zigzag).

Why Does Everyone Zigzag?



If the rest of the traffic splits evenly over the two old paths, they take 65 + z/200 and the zigzag 40 + z/100 min. The zigzag is faster **for every z** (B→t is at most 40 min < 45), so it is a dominant choice for everyone. But each switching driver loads the narrow roads: at z = 4,000 everyone takes 80 min.

Braess's Paradox: Lessons and Examples

- Adding a link to a network can increase the delay of **all** users because of selfish route choice (Braess, 1968).
- The new road looks attractive individually, but the traffic pattern users create together moves to a worse equilibrium.
- Conversely, *closing* a road can sometimes speed everyone up.

Stuttgart (1969)

Traffic that worsened after a new road investment is often said to have improved once a street was closed (Knödel, 1969).

New York (1990)

When 42nd Street was closed on Earth Day, the expected gridlock reportedly never came (Kolata, 1990).

Mechanical / electrical

A similar paradox appears in networks of springs and resistors (Cohen and Horowitz, 1991).

Tolls, access restrictions or central routing can reduce the effect.

Back to the Warm-up: Braess's Paradox

The theory's answer: no

Average time	Selfish equilibrium	Optimum
No shortcut	20 + 45 = 65	65
With shortcut	**40 + 0 + 40 = 80**	≈ 64.7

With the shortcut everyone picks S→A→B→T. Going back to an old road costs 40 + 45 = 85, so nobody does: equilibrium 80. The optimum with the shortcut gives the zigzag a small share (1,750 / 1,750 / 500 cars → 258,750 / 4,000 ≈ 64.7).

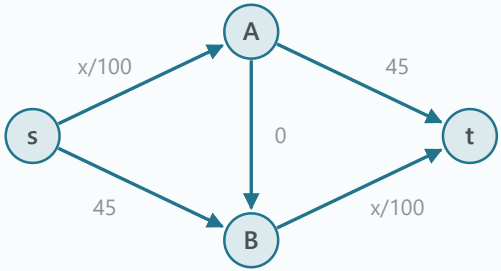
In practice

In laboratory experiments, too, subjects drifted over time toward the slower equilibrium the paradox predicts once the shortcut was added (Rapoport et al., 2009).

In cities: besides Stuttgart and New York, removing the elevated highway over Seoul's Cheonggyecheon in the early 2000s is often said not to have worsened traffic as feared. Real networks are far more complex; these cases are not proof, only signs that the paradox is possible.

Building or closing a road should be judged by the equilibrium drivers will reach, not by capacity alone.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	
s→A	
A→t	
s→B	
B→t	
A→B	

Call: the Braess network, 5 edges, 4,000 cars. Question: *closing* which edge cuts the equilibrium time the most?

Step 1 / 13

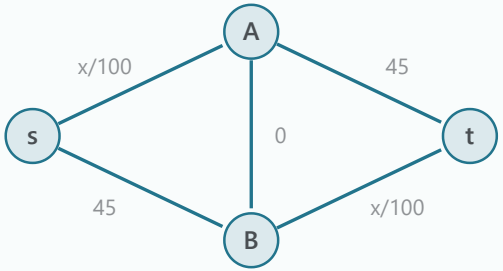
```
▶ def best_subnet(net, r):
2   best, cut = eq_time(net, r), None
3   for e in net:
4     sub = [k for k in net if k != e]
5     d = eq_time(sub, r)   # inf if no path
6     if d < best:
7       best, cut = d, e
8   return cut, best
```

VARIABLES

`r = 4000`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	
A→t	
s→B	
B→t	
A→B	

best
80 min
cut = None

Equilibrium of the full network: everyone zigzags, **80 min**. Removing nothing is the initial candidate.

Step 2 / 13

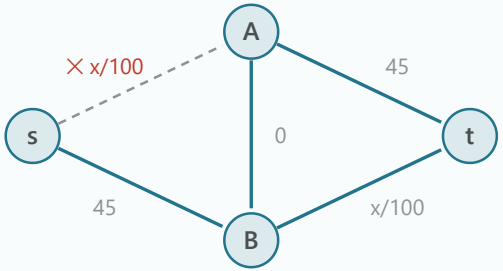
```
1 def best_subnet(net, r):
▶   best, cut = eq_time(net, r), None
3   for e in net:
4     sub = [k for k in net if k != e]
5     d = eq_time(sub, r)   # inf if no path
6     if d < best:
7       best, cut = d, e
8   return cut, best
```

VARIABLES

`r = 4000` `best = 80` `cut = None`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	
s→B	
B→t	
A→B	

best
80 min
cut = None

Close s→A: only s→B→t remains: 40 + 45 = 85 min.

Step 3 / 13

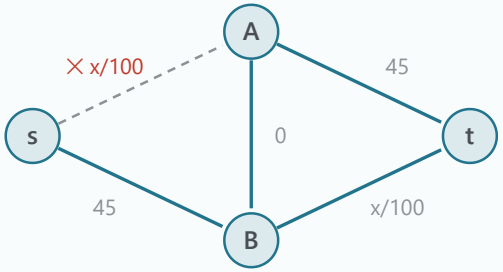
```
1  def best_subnet(net, r):
2    best, cut = eq_time(net, r), None
▶    for e in net:
▶      sub = [k for k in net if k != e]
▶      d = eq_time(sub, r)   # inf if no path
6      if d < best:
7        best, cut = d, e
8    return cut, best
```

VARIABLES

`r = 4000` `best = 80` `cut = None` `e = sA` `d = 85`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	
s→B	
B→t	
A→B	

best
80 min
cut = None

85 < 80 is false → candidate unchanged.

Step 4 / 13

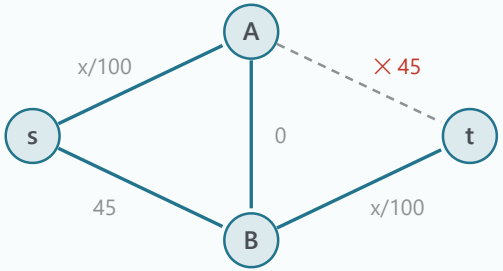
```
1  def best_subnet(net, r):
2    best, cut = eq_time(net, r), None
3    for e in net:
4      sub = [k for k in net if k != e]
5      d = eq_time(sub, r)   # inf if no path
▶      if d < best:
7        best, cut = d, e
8    return cut, best
```

VARIABLES

`r = 4000` `best = 80` `cut = None` `e = sA` `d = 85`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	80 min
s→B	
B→t	
A→B	

best
80 min
cut = None

Close A→t: the zigzag and the lower path remain; the zigzag dominates: 80 min.

Step 5 / 13

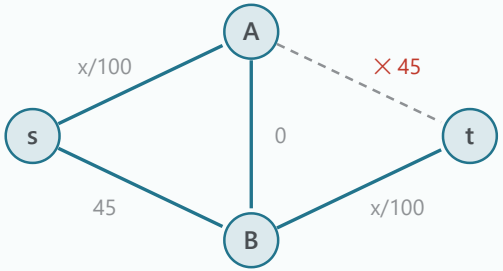
```
1 def best_subnet(net, r):
2   best, cut = eq_time(net, r), None
▶   for e in net:
▶     sub = [k for k in net if k != e]
▶     d = eq_time(sub, r)   # inf if no path
6     if d < best:
7       best, cut = d, e
8   return cut, best
```

VARIABLES

`r = 4000` `best = 80` `cut = None` `e = At` `d = 80`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	80 min
s→B	
B→t	
A→B	

best
80 min
cut = None

80 < 80 is false → candidate unchanged.

Step 6 / 13

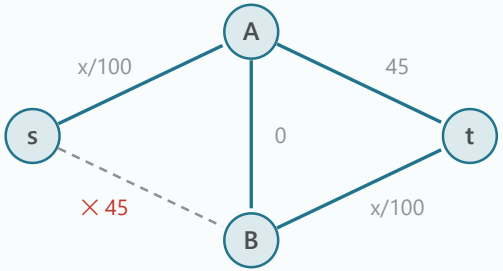
```
1 def best_subnet(net, r):
2   best, cut = eq_time(net, r), None
3   for e in net:
4     sub = [k for k in net if k != e]
5     d = eq_time(sub, r)   # inf if no path
▶     if d < best:
7       best, cut = d, e
8   return cut, best
```

VARIABLES

`r = 4000` `best = 80` `cut = None` `e = At` `d = 80`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	80 min
s→B	80 min
B→t	
A→B	

best
80 min
cut = None

Close s→B: the upper path and the zigzag remain; the zigzag dominates: 80 min.

Step 7 / 13

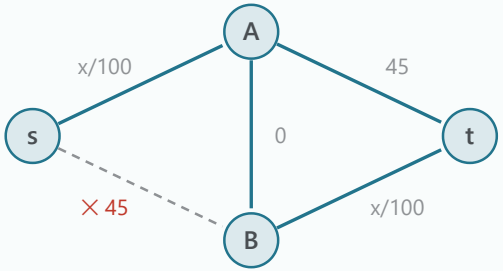
```
1 def best_subnet(net, r):
2   best, cut = eq_time(net, r), None
▶   for e in net:
▶     sub = [k for k in net if k != e]
▶     d = eq_time(sub, r)   # inf if no path
6     if d < best:
7       best, cut = d, e
8   return cut, best
```

VARIABLES

`r = 4000` `best = 80` `cut = None` `e = sB` `d = 80`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	80 min
s→B	80 min
B→t	
A→B	

best
80 min
cut = None

80 < 80 is false → candidate unchanged.

Step 8 / 13

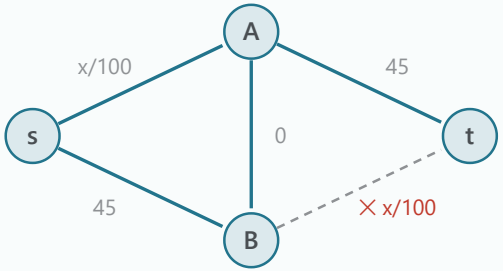
```
1 def best_subnet(net, r):
2   best, cut = eq_time(net, r), None
3   for e in net:
4     sub = [k for k in net if k != e]
5     d = eq_time(sub, r)   # inf if no path
▶     if d < best:
7       best, cut = d, e
8   return cut, best
```

VARIABLES

`r = 4000` `best = 80` `cut = None` `e = sB` `d = 80`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	80 min
s→B	80 min
B→t	85 min
A→B	

best
80 min
cut = None

Close B→t: only s→A→t remains: 40 + 45 = 85 min.

Step 9 / 13

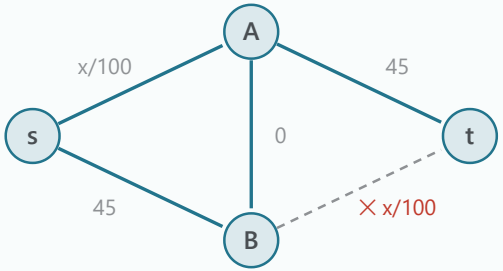
```
1 def best_subnet(net, r):
2   best, cut = eq_time(net, r), None
▶   for e in net:
▶     sub = [k for k in net if k != e]
▶     d = eq_time(sub, r)   # inf if no path
6     if d < best:
7       best, cut = d, e
8   return cut, best
```

VARIABLES

`r = 4000` `best = 80` `cut = None` `e = Bt` `d = 85`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	80 min
s→B	80 min
B→t	85 min
A→B	

best
80 min
cut = None

85 < 80 is false → candidate unchanged.

Step 10 / 13

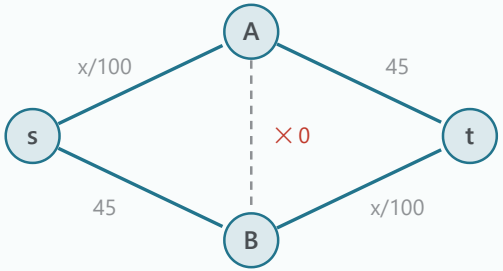
```
1 def best_subnet(net, r):
2   best, cut = eq_time(net, r), None
3   for e in net:
4     sub = [k for k in net if k != e]
5     d = eq_time(sub, r)   # inf if no path
▶     if d < best:
7       best, cut = d, e
8   return cut, best
```

VARIABLES

`r = 4000` `best = 80` `cut = None` `e = Bt` `d = 85`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	80 min
s→B	80 min
B→t	85 min
A→B	65 min

best
80 min
cut = None

Close A→B: the pre-Braess network: 2,000 / 2,000, 20 + 45 = 65 min.

Step 11 / 13

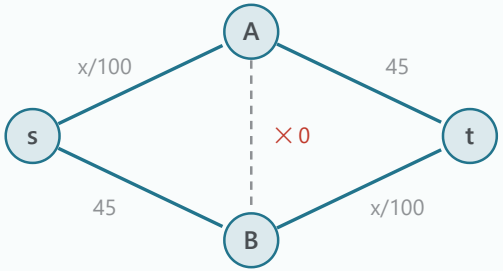
```
1 def best_subnet(net, r):
2   best, cut = eq_time(net, r), None
▶   for e in net:
▶     sub = [k for k in net if k != e]
▶     d = eq_time(sub, r)   # inf if no path
6     if d < best:
7       best, cut = d, e
8   return cut, best
```

VARIABLES

`r = 4000` `best = 80` `cut = None` `e = AB` `d = 65`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	80 min
s→B	80 min
B→t	85 min
A→B	65 min

best
65 min
cut = A→B

65 < previous best → **update**: new best subnetwork "A→B closed".

Step 12 / 13

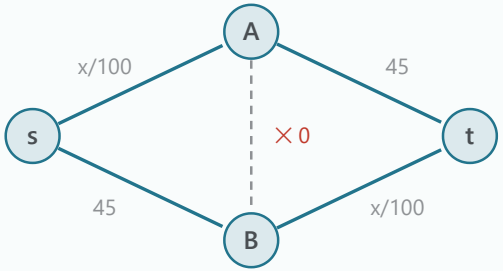
```
1 def best_subnet(net, r):
2   best, cut = eq_time(net, r), None
3   for e in net:
4     sub = [k for k in net if k != e]
5     d = eq_time(sub, r)   # inf if no path
▶     if d < best:
▶       best, cut = d, e
8   return cut, best
```

VARIABLES

`r = 4000` `best = 65` `cut = AB` `e = AB` `d = 65`

eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Code Trace: `best_subnet(net, r)`



removed	eq. time
—	80 min
s→A	85 min
A→t	80 min
s→B	80 min
B→t	85 min
A→B	65 min

best
65 min
cut = A→B

Result: **close A→B**; equilibrium 80 → 65 min. The designer asks "which edges should stay open?"; in general networks this choice is hard.

Step 13 / 13

```
1 def best_subnet(net, r):
2   best, cut = eq_time(net, r), None
3   for e in net:
4     sub = [k for k in net if k != e]
5     d = eq_time(sub, r)   # inf if no path
6     if d < best:
7       best, cut = d, e
▶   return cut, best
```

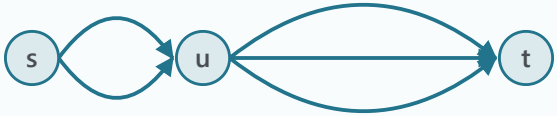
VARIABLES

`r = 4000` `best = 65` `cut = AB`

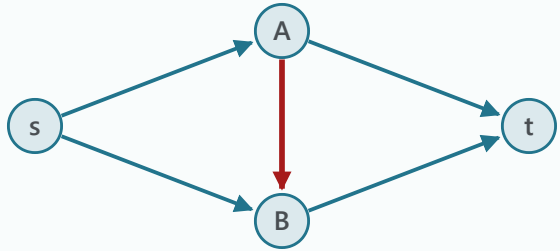
eq_time(net, r): the common time of the used paths in the subnetwork's Wardrop equilibrium (computed in the script by minimizing the Beckmann potential). The code only tries removing one edge; a full search needs 2^m subnetworks.

Network Design for Selfish Users

Series-parallel: no paradox



built from series/parallel joins



Wheatstone (Braess) substructure

Problem

Which subnetwork of a given network, left open, minimizes the Wardrop equilibrium time? *Closing* a road, not only building one, is a design decision.

Further reading: hardness

Choosing the best edges to close is hard in general: with linear delays, closing nothing is at most 4/3 times worse than the best choice, and guaranteeing anything better efficiently is NP-hard (Roughgarden, 2006).

Further reading: topology

In networks built from parallel paths and their series connections (series-parallel), Braess's paradox never occurs; other networks can show it with suitable delays (Milchtaich, 2006).

Congestion Games

- n players, a set of resources E (edges). Each player's strategy is a subset of resources (e.g. an s–t path).
- A resource's cost $c_e(n_e)$ depends only on the **number** of players using it and grows with that load.
- A player's cost: the sum of the costs of the chosen resources.
- As players try to cut their own delay, the total network cost can rise.

	Atomic	**Nonatomic**
Players	finite, n players	continuous flow
Effect	one player adds 1 to the load	one player is negligible
Equilibrium	pure Nash	Wardrop
Potential	Rosenthal (sum)	Beckmann (integral)

Rosenthal (1973): every finite congestion game has a potential function, so a pure Nash equilibrium always exists. No mixed strategies needed.

Rosenthal's Potential Function

$$\Phi(s) = \Sigma_e \left[c_e(1) + c_e(2) + \ldots + c_e(n_e) \right]$$

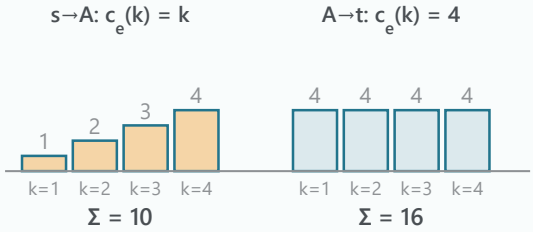
On each edge, add the costs from 1 up to the number of users n_e (this is not the total cost).

Rosenthal's Potential Function

$$\Phi(s) = \Sigma_e \left[c_e(1) + c_e(2) + \ldots + c_e(n_e) \right]$$

On each edge, add the costs from 1 up to the number of users n_e (this is not the total cost).

Example: all 4 drivers on s→A→t. s→A: $c_e(k) = k$ → 1 + 2 + 3 + 4 = 10; A→t: $c_e(k) = 4$ → 16; other edges empty. **Φ = 26** (total cost 4 × 8 = 32).

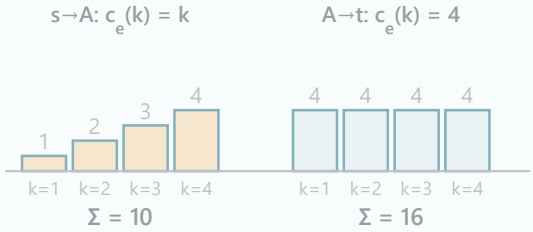


Rosenthal's Potential Function

$$\Phi(s) = \Sigma_e \left[c_e(1) + c_e(2) + \ldots + c_e(n_e) \right]$$

On each edge, add the costs from 1 up to the number of users n_e (this is not the total cost).

Example: all 4 drivers on s→A→t. s→A: $c_e(k) = k$ → 1 + 2 + 3 + 4 = 10; A→t: $c_e(k) = 4$ → 16; other edges empty. **Φ = 26** (total cost 4 × 8 = 32).



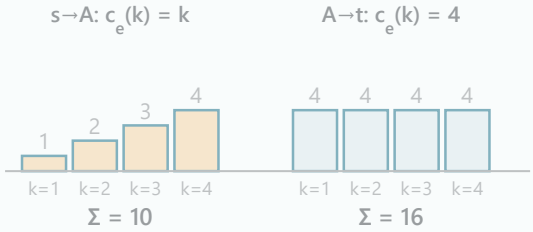
Key identity. When driver i leaves an edge, the last term $c_e(n_e)$ of that edge's sum disappears (that is what i paid there); entering a new edge adds $c_e(n_e + 1)$ (i's new cost). So the change in Φ is exactly i's change in cost: **$\Delta\Phi = c_i(\text{new}) - c_i(\text{old})$**.

Rosenthal's Potential Function

$$\Phi(s) = \Sigma_e \left[c_e(1) + c_e(2) + \ldots + c_e(n_e) \right]$$

On each edge, add the costs from 1 up to the number of users n_e (this is not the total cost).

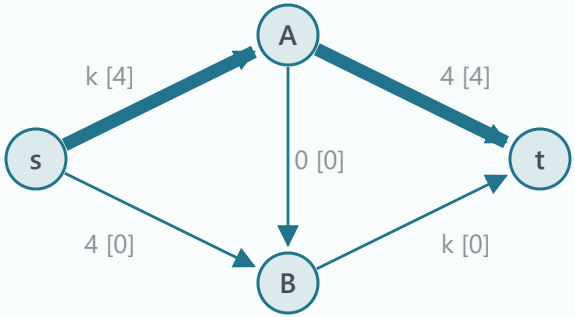
Example: all 4 drivers on s→A→t. s→A: $c_e(k) = k$ → 1 + 2 + 3 + 4 = 10; A→t: $c_e(k) = 4$ → 16; other edges empty. **Φ = 26** (total cost 4 × 8 = 32).



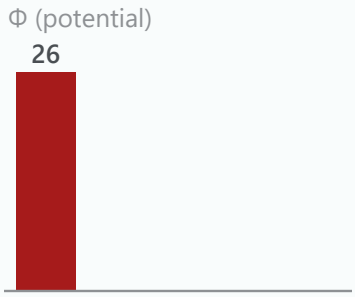
Key identity. When driver i leaves an edge, the last term $c_e(n_e)$ of that edge's sum disappears (that is what i paid there); entering a new edge adds $c_e(n_e + 1)$ (i's new cost). So the change in Φ is exactly i's change in cost: **$\Delta\Phi = c_i(\text{new}) - c_i(\text{old})$**.

Consequence. Every improving move strictly decreases Φ; Φ takes finitely many values → the dynamics stop. Where they stop, nobody can improve → **a pure Nash equilibrium exists** (Rosenthal, 1973).

Step by Step: Best-Response Dynamics and Φ

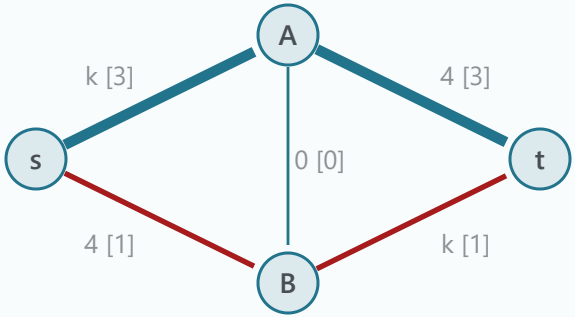


driver	path	cost
1	top	8
2	top	8
3	top	8
4	top	8

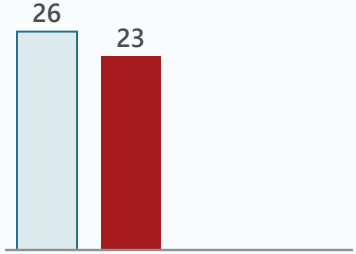


Edge costs: k (number of users) on s→A and B→t, 4 on s→B and A→t, 0 on A→B. Everyone on s→A→t: cost 4 + 4 = 8, **Φ = 26**.

Step by Step: Best-Response Dynamics and Φ

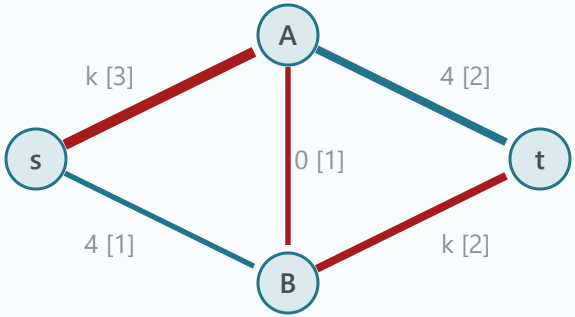


driver	path	cost
1	bottom	5
2	top	7
3	top	7
4	top	7

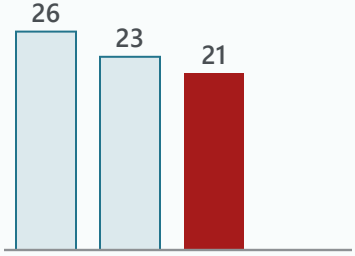


Driver 1's options: top 8, bottom 4 + 1 = 5, zig 4 + 0 + 1 = 5. Ties go to the first → **bottom** (8 → 5). Φ: 26 → 23 (Δ = −3).

Step by Step: Best-Response Dynamics and Φ

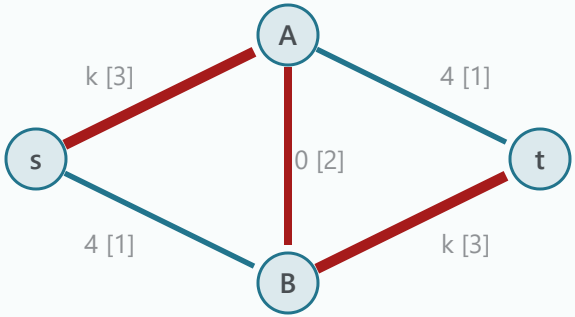


driver	path	cost
1	bottom	6
2	zig	5
3	top	7
4	top	7

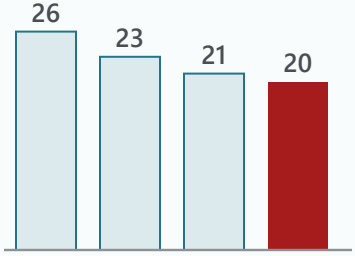


Driver 2: top 7, bottom 4 + 2 = 6, zig 3 + 0 + 2 = 5 → **zig** (7 → 5). Φ: 23 → 21 (Δ = −2).

Step by Step: Best-Response Dynamics and Φ

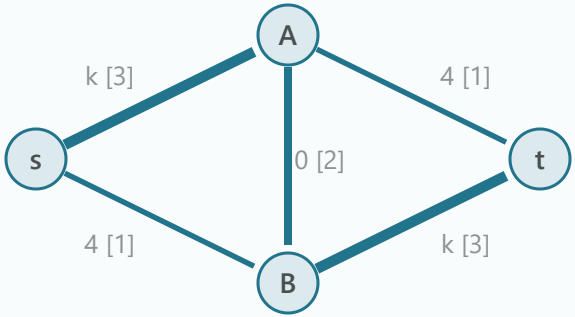


driver	path	cost
1	bottom	7
2	zig	6
3	zig	6
4	top	7

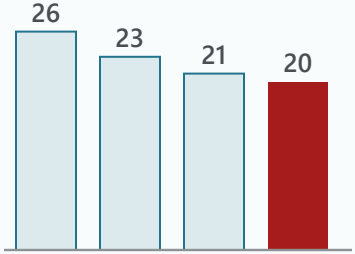


Driver 3: top 7, bottom 4 + 3 = 7, zig 3 + 0 + 3 = 6 → **zig** (7 → 6). Φ: 21 → 20 (Δ = −1).

Step by Step: Best-Response Dynamics and Φ

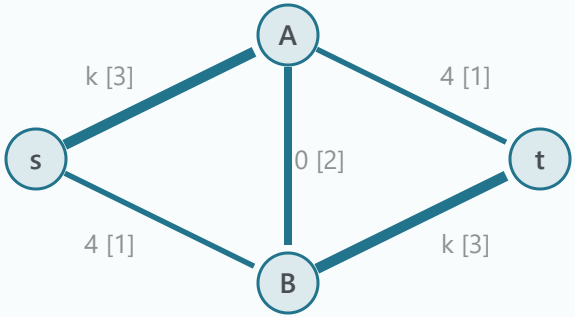


driver	path	cost
1	bottom	7
2	zig	6
3	zig	6
4	top	7

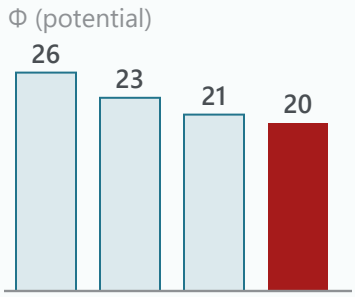


Driver 4 (top, 7): bottom 4 + 4 = 8, zig 3 + 0 + 4 = 7 → nothing better, **stays**. Each drop in Φ equaled the mover's gain.

Step by Step: Best-Response Dynamics and Φ



driver	path	cost
1	bottom	7
2	zig	6
3	zig	6
4	top	7



In the second round nobody can improve → **pure Nash equilibrium** (Φ = 20). Total 7 + 6 + 6 + 7 = 26; without A→B a 2+2 split would give 24: atomic Braess!

Potential Games: What Do We Gain?

Existence

A state minimizing Φ is a pure Nash equilibrium; an equilibrium always exists.

Convergence

No sequence of improving moves is infinite (the finite improvement property): best-response dynamics with strict improvements always stop (a player already best-responding stays put).

Speed (further reading)

Stopping is guaranteed, speed is not: for some congestion games no fast method for finding an equilibrium is known (Fabrikant, Papadimitriou, Talwar, 2004).

Generality

Every (exact) potential game is equivalent to a congestion game (Monderer and Shapley, 1996).

Learning

If players best-respond in turn, they reach an equilibrium.

Efficiency

The equilibrium minimizing Φ is one way to select a good equilibrium among many, but it is no guarantee.

Network Externalities: Negative and Positive

- **Externality**: a user's decision changes others' costs or benefits, but the user ignores this in their own calculation.

Negative: congestion

- Each extra driver imposes a total delay of $x \cdot \ell'(x)$ on the x others on the same road.
- In the two-route example, with 2,000 cars on Route 1: $2{,}000 \times 1/100 = 20$ car·min.
- Result: the busy road is overused; the remedy is a congestion toll.

Positive: network effects

+ The value of joining grows with the number of members: phones, messaging, social networks.
+ n users have $n(n-1)/2$ possible connections; the claim that value grows like n^2 (Metcalfe) is a heuristic.
+ A new user ignores the benefit they bring others: adoption stays slow until a *critical mass* is reached.

Network formation games have both: a link shortens distances for others as well as its builder (a positive externality), but only the buyer pays for it.

Network Formation Games

- So far the network was *given*; now the players build it themselves: Internet peering, social networks, P2P networks.
- Every link has a building cost; being well connected (short distances) is valuable. The tension: **link cost ↔ reach**.

Fabrikant et al. (2003): unilateral

Player i buys links to any nodes (α each).
Cost: α·(links bought) + Σ_j dist(i, j).
Equilibrium: Nash.

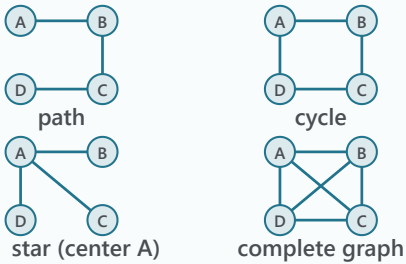
Jackson–Wolinsky (1996): bilateral

A link needs both sides' consent to form; either side can cut it. Solution concept: **pairwise stability**.

Questions: Which networks are equilibria? Which is socially best? How far apart can they be?

Bilateral Links: Pairwise Stability (n = 4)

- Benefit: 4 from each player at distance 1, 2.2 at distance 2, 1 at distance 3.
- Each link costs 2 to **each** of its two endpoints.
- **Pairwise stable**: (i) no player gains by cutting one of their links; (ii) no missing link would benefit both endpoints *together*.

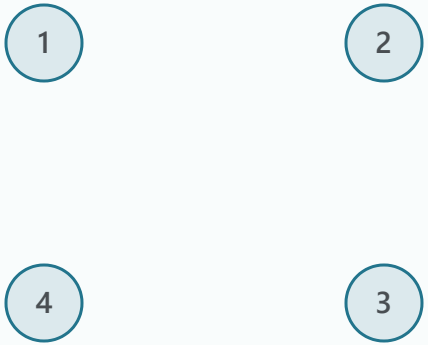


Network	Payoffs	Total	Pairwise stable?
path	5.2/6.2/6.2/5.2	22.8	No: A–D is added (+3 benefit, −2 cost each)
cycle	6.2 (all)	24.8	Yes
star	6/6.4/6.4/6.4	**25.2**	Yes; highest total
complete graph	6 (all)	24	No: cutting a link gains 2, loses 1.8

In the star, two leaves linking would each gain 1.8 but pay 2, so they don't.

There can be several pairwise stable networks, and they need not match the one with the highest total: the cycle is stable but trails the star by 0.4.

Step by Step: A Network Formation Game (n = 4, α = 1.5)



4 players, link price α = 1.5

cost_i = α · (links bought)

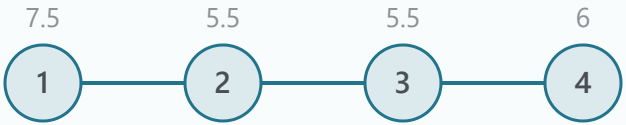
+ Σ_j dist(i, j)

Social cost = Σ cost_i

Disconnected → cost ∞

Each player chooses which links to buy *themselves*; everyone may use a link, whoever bought it.

Step by Step: A Network Formation Game (n = 4, α = 1.5)



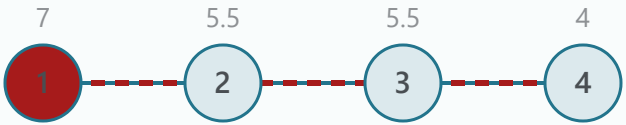
Path 1–2–3–4 (i buys i+1)

costs 7.5 · 5.5 · 5.5 · 6

social cost 24.5

Path network: 1 buys one link (1.5) and its distances are 1 + 2 + 3 = 6 → 7.5. The end players are far from everyone.

Step by Step: A Network Formation Game (n = 4, α = 1.5)



If 1 buys a link to 4:

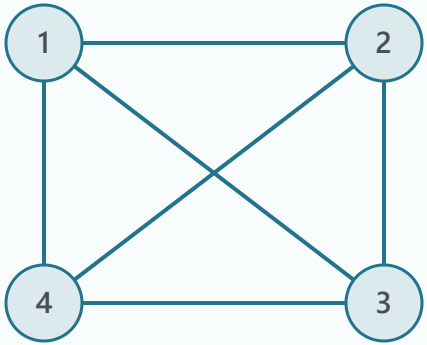
$2 \cdot 1.5 + (1 + 2 + 1) = 7$

$7 < 7.5 \rightarrow$ gain

The path is not an equilibrium

If player 1 buys a second link to 4, its distances drop from 6 to 4: $3 + 4 = 7 < 7.5 \rightarrow$ **a profitable deviation**.

Step by Step: A Network Formation Game (n = 4, α = 1.5)



Complete graph: 6 links

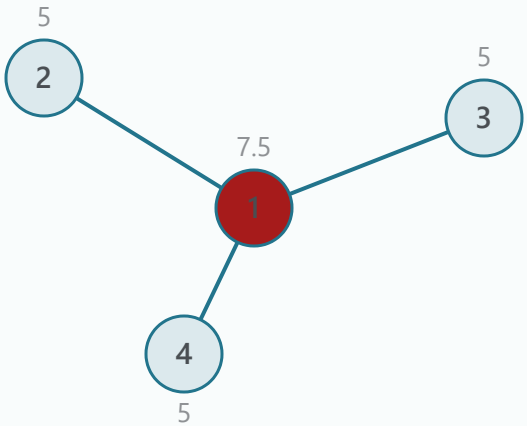
social: 6 · 1.5 + 12 = 21

Dropping a link: −1.5 + 1

→ gains, not an equilibrium

The complete graph is socially cheapest (21). But a player who bought a link saves 1.5 by dropping it and loses only 1 in distance → not an equilibrium.

Step by Step: A Network Formation Game (n = 4, α = 1.5)



Center 1 buys three links

center 4.5 + 3 = 7.5; leaf 5

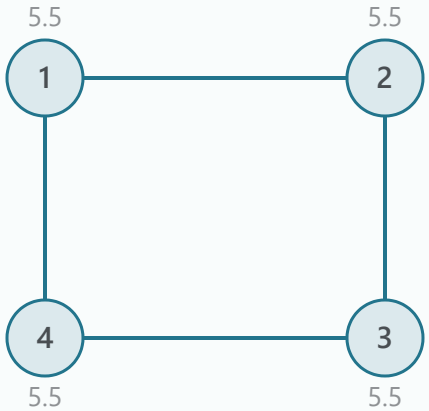
A leaf buying a link: 1.5 + 4 = 5.5

Center drops a link → split

→ Nash equilibrium (22.5)

Star: a new link saves a leaf only 1 in distance (1 < 1.5); if the center drops a link the network splits → **nobody deviates**.

Step by Step: A Network Formation Game (n = 4, α = 1.5)



4-cycle (one link each): 5.5

social cost 22

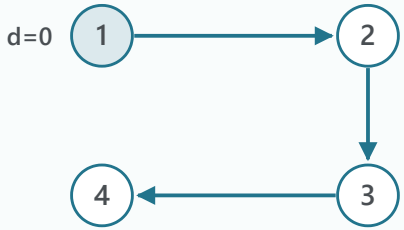
Script: all 4,096 profiles checked

equilibria: star 22.5 · cycle 22

optimum: complete graph 21

Brute-force search: at α = 1.5 the equilibria are the star (22.5) and the 4-cycle (22); the optimum (21) is not an equilibrium.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: [1]

dist: 1:0

Σ dist = 0

arrow: buyer → other end · red: new link

Path network 1–2–3–4; 1 has bought only link (1, 2). BFS starts at i = 1: dist = {1: 0}.

Step 1 / 12

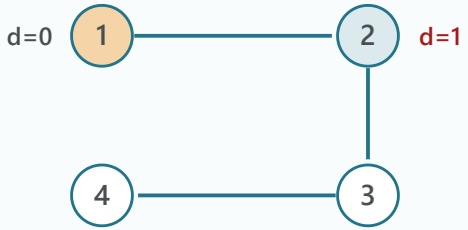
```
1  def cost(i, links, alpha):
▶    dist = {i: 0}; queue = [i]
3    while queue:
4      u = queue.pop(0)
5      for v in nbrs(links, u):
6        if v not in dist:
7          dist[v] = dist[u] + 1
8          queue.append(v)
9    bought = sum(1 for (a, b) in links if a == i)
10   return alpha * bought + sum(dist.values())
```

VARIABLES

`dist = {1: 0}` `queue = [1]`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: [2]

dist: 1:0 2:1

Σ dist = 1

arrow: buyer → other end · red: new link

u = 1 leaves the queue; new neighbor 2 → distance 1.

Step 2 / 12

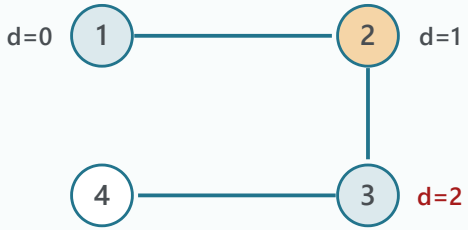
```
1  def cost(i, links, alpha):
2    dist = {i: 0}; queue = [i]
3    while queue:
▶      u = queue.pop(0)
▶      for v in nbrs(links, u):
▶        if v not in dist:
▶          dist[v] = dist[u] + 1
▶          queue.append(v)
9    bought = sum(1 for (a, b) in links if a == i)
10   return alpha * bought + sum(dist.values())
```

VARIABLES

`dist = {1: 0, 2: 1}` `queue = [2]` `u = 1`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: [3]

dist: 1:0 2:1 3:2

Σ dist = 3

arrow: buyer → other end · red: new link

u = 2 leaves the queue; new neighbor 3 → distance 2.

Step 3 / 12

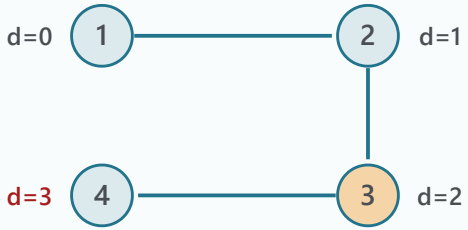
```
1  def cost(i, links, alpha):
2    dist = {i: 0}; queue = [i]
3    while queue:
▶      u = queue.pop(0)
▶      for v in nbrs(links, u):
▶        if v not in dist:
▶          dist[v] = dist[u] + 1
▶          queue.append(v)
9    bought = sum(1 for (a, b) in links if a == i)
10   return alpha * bought + sum(dist.values())
```

VARIABLES

`dist = {1: 0, 2: 1, 3: 2}` `queue = [3]` `u = 2`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: [4]

dist: 1:0 2:1 3:2 4:3

Σ dist = 6

arrow: buyer → other end · red: new link

u = 3 leaves the queue; new neighbor 4 → distance 3.

Step 4 / 12

```
1  def cost(i, links, alpha):
2    dist = {i: 0}; queue = [i]
3    while queue:
▶      u = queue.pop(0)
▶      for v in nbrs(links, u):
▶        if v not in dist:
▶          dist[v] = dist[u] + 1
▶          queue.append(v)
9    bought = sum(1 for (a, b) in links if a == i)
10   return alpha * bought + sum(dist.values())
```

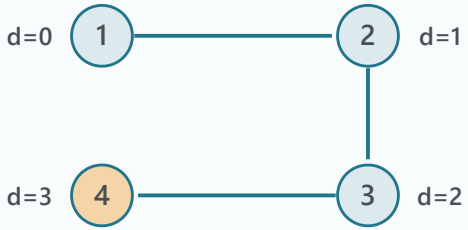
VARIABLES

`dist = {1: 0, 2: 1, 3: 2, 4: 3}`

`queue = [4]` `u = 3`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: []

dist: 1:0 2:1 3:2 4:3

Σ dist = 6

arrow: buyer → other end · red: new link

u = 4 leaves the queue; all its neighbors are already visited.

Step 5 / 12

```
1  def cost(i, links, alpha):
2    dist = {i: 0}; queue = [i]
3    while queue:
▶      u = queue.pop(0)
▶      for v in nbrs(links, u):
▶        if v not in dist:
▶          dist[v] = dist[u] + 1
▶          queue.append(v)
9    bought = sum(1 for (a, b) in links if a == i)
10   return alpha * bought + sum(dist.values())
```

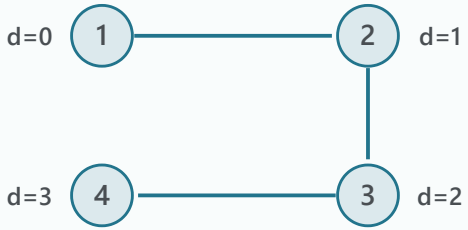
VARIABLES

`dist = {1: 0, 2: 1, 3: 2, 4: 3}`

`queue = []` `u = 4`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: []

dist: 1:0 2:1 3:2 4:3

Σ dist = 6

arrow: buyer → other end · red: new link

Everyone is reachable. bought = 1 → cost 1.5·1 + 6 = **7.5**. Now let's try buying one more link.

Step 6 / 12

```
1 def cost(i, links, alpha):
2   dist = {i: 0}; queue = [i]
3   while queue:
4     u = queue.pop(0)
5     for v in nbrs(links, u):
6       if v not in dist:
7         dist[v] = dist[u] + 1
8         queue.append(v)
▶   bought = sum(1 for (a, b) in links if a == i)
▶   return alpha * bought + sum(dist.values())
```

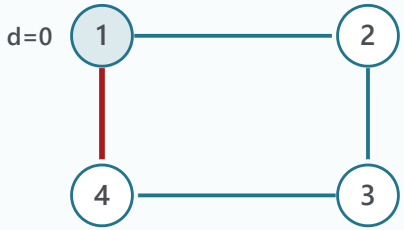
VARIABLES

`dist = {1: 0, 2: 1, 3: 2, 4: 3}`

`queue = []` `bought = 1`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: [1]

dist: 1:0

Σ dist = 0

arrow: buyer → other end · red: new link

Deviation: 1 also buys link (1, 4) (orange). Same computation again. BFS starts at i = 1: dist = {1: 0}.

Step 7 / 12

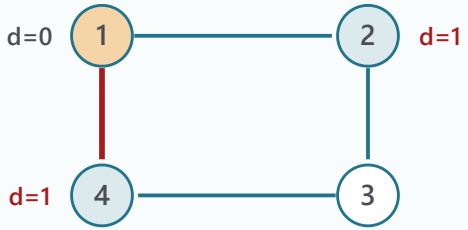
```
1  def cost(i, links, alpha):
▶     dist = {i: 0}; queue = [i]
3     while queue:
4       u = queue.pop(0)
5       for v in nbrs(links, u):
6         if v not in dist:
7           dist[v] = dist[u] + 1
8           queue.append(v)
9     bought = sum(1 for (a, b) in links if a == i)
10    return alpha * bought + sum(dist.values())
```

VARIABLES

`dist = {1: 0}` `queue = [1]`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: [2, 4]

dist: 1:0 2:1 4:1

Σ dist = 2

arrow: buyer → other end · red: new link

u = 1 leaves the queue; new neighbors 2, 4 → distance 1.

Step 8 / 12

```
1  def cost(i, links, alpha):
2    dist = {i: 0}; queue = [i]
3    while queue:
▶      u = queue.pop(0)
▶      for v in nbrs(links, u):
▶        if v not in dist:
▶          dist[v] = dist[u] + 1
▶          queue.append(v)
9    bought = sum(1 for (a, b) in links if a == i)
10   return alpha * bought + sum(dist.values())
```

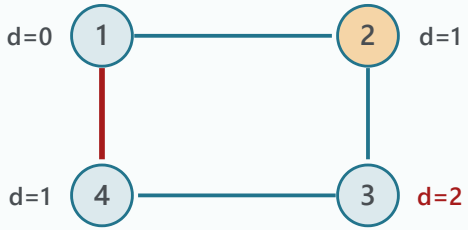
VARIABLES

`dist = {1: 0, 2: 1, 4: 1}`

`queue = [2, 4]` `u = 1`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: [4, 3]

dist: 1:0 2:1 3:2 4:1

Σ dist = 4

arrow: buyer → other end · red: new link

u = 2 leaves the queue; new neighbor 3 → distance 2.

Step 9 / 12

```
1  def cost(i, links, alpha):
2    dist = {i: 0}; queue = [i]
3    while queue:
▶      u = queue.pop(0)
▶      for v in nbrs(links, u):
▶        if v not in dist:
▶          dist[v] = dist[u] + 1
▶          queue.append(v)
9    bought = sum(1 for (a, b) in links if a == i)
10   return alpha * bought + sum(dist.values())
```

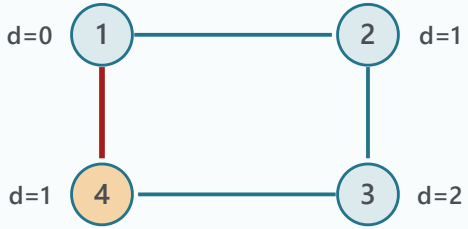
VARIABLES

`dist = {1: 0, 2: 1, 3: 2, 4: 1}`

`queue = [4, 3]` `u = 2`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: [3]

dist: 1:0 2:1 3:2 4:1

Σ dist = 4

arrow: buyer → other end · red: new link

u = 4 leaves the queue; all its neighbors are already visited.

Step 10 / 12

```
1  def cost(i, links, alpha):
2    dist = {i: 0}; queue = [i]
3    while queue:
▶      u = queue.pop(0)
▶      for v in nbrs(links, u):
▶        if v not in dist:
▶          dist[v] = dist[u] + 1
▶          queue.append(v)
9    bought = sum(1 for (a, b) in links if a == i)
10   return alpha * bought + sum(dist.values())
```

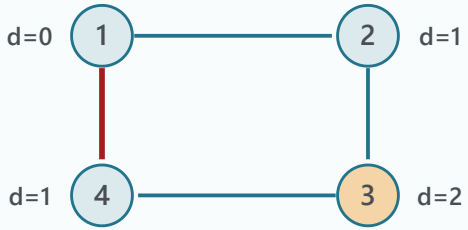
VARIABLES

`dist = {1: 0, 2: 1, 3: 2, 4: 1}`

`queue = [3]` `u = 4`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: []

dist: 1:0 2:1 3:2 4:1

Σ dist = 4

arrow: buyer → other end · red: new link

u = 3 leaves the queue; all its neighbors are already visited.

Step 11 / 12

```
1  def cost(i, links, alpha):
2    dist = {i: 0}; queue = [i]
3    while queue:
▶      u = queue.pop(0)
▶      for v in nbrs(links, u):
▶        if v not in dist:
▶          dist[v] = dist[u] + 1
▶          queue.append(v)
9    bought = sum(1 for (a, b) in links if a == i)
10   return alpha * bought + sum(dist.values())
```

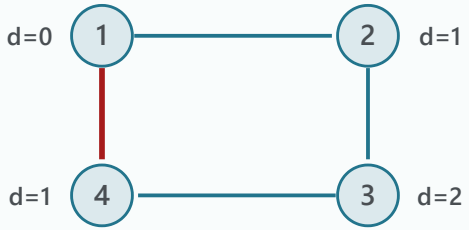
VARIABLES

`dist = {1: 0, 2: 1, 3: 2, 4: 1}`

`queue = []` `u = 3`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Code Trace: `cost(i, links, alpha)`



player i = 1

queue: []

dist: 1:0 2:1 3:2 4:1

Σ dist = 4

arrow: buyer → other end · red: new link

Everyone is reachable. bought = 2 → cost 1.5·2 + 4 = **7**. 7 < 7.5 → **a profitable deviation**: the path is not a Nash equilibrium.

Step 12 / 12

```
1 def cost(i, links, alpha):
2   dist = {i: 0}; queue = [i]
3   while queue:
4     u = queue.pop(0)
5     for v in nbrs(links, u):
6       if v not in dist:
7         dist[v] = dist[u] + 1
8         queue.append(v)
▶   bought = sum(1 for (a, b) in links if a == i)
▶   return alpha * bought + sum(dist.values())
```

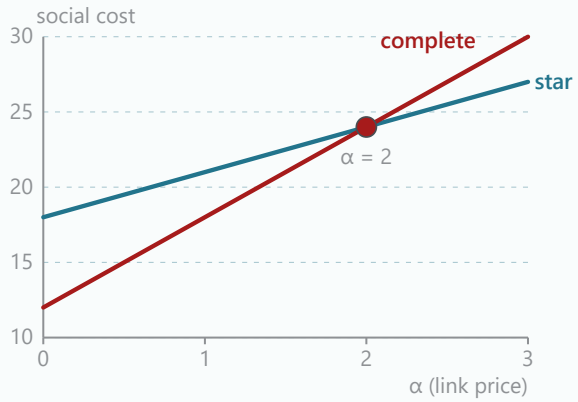
VARIABLES

`dist = {1: 0, 2: 1, 3: 2, 4: 1}`

`queue = []` `bought = 2`

nbrs(links, u): neighbors regardless of who bought the link (everyone uses links); the network is assumed connected. A player pays only for the links *they* bought.

Equilibrium and Optimum as the Link Price Changes



α	Optimum	Equilibria (script)
0.5	compl. (15)	compl. (15)
1.5	compl. (21)	cycle 22, star 22.5
2.5	star (25.5)	star 25.5 · path 27.5

For n = 4 the star costs 3α + 18 and the complete graph 6α + 12: for α < 2 the complete graph is cheaper, for α > 2 the star. For general n the optimum is likewise the complete graph for α ≤ 2 and the star for α ≥ 2 (Fabrikant et al., 2003).

α < 1: links are cheap, everyone connects → the complete graph is both equilibrium and optimum. α ≥ 1: the star is always an equilibrium, but costlier equilibria can appear too (at α = 2.5, the path, with the middle players buying the links).

Routing in Networks: Centralized or Distributed?

	Centralized routing	Distributed (selfish) routing
Who decides?	A traffic control center, an SDN (software-defined networking) controller, ISP traffic engineering	Each driver / each router (navigation apps, BGP: the inter-domain routing protocol)
Goal	Minimum total delay C(f) (Wardrop's 2nd principle)	Minimum own delay (Wardrop's 1st principle)
Two-route example	1,500 / 1,500 → 71,250 car·min	2,000 / 1,000 → 75,000 car·min
Requires	Full demand and delay information; users must *comply*	Only local information; no compliance problem
Weakness	At the optimum some are on the slower road → urge to deviate; a single point of failure	Inefficient equilibria such as Braess's

Hybrids are common in networks: the center controls only **prices** (congestion tolls), **information** (route suggestions) or part of the traffic (Stackelberg routing: the center routes that part itself); users choose the rest.

Incentives, Load Balancing and Real Life

Right incentives

+ Steer users toward balanced use of network resources → efficiency improves.
+ Traffic: tolls or warnings reduce congestion.
+ Smart grid: time-of-use pricing balances load at peak hours.

Wrong incentives

– Strategic, selfish behavior grows → congestion and low efficiency.
– An app sending everyone the same “fastest” route can flood side streets.
– A badly chosen toll shifts traffic to another bottleneck.

Internet

ISPs route data packets; BGP picks routes between networks.

Traffic management

Route optimization, smart signals, navigation apps.

Energy

Energy distribution and load shifting in smart grids.

Efficiency vs. Fairness: Sharing Bandwidth

Two links have capacity 1. Flow A uses both, B only link 1, C only link 2: $x_A + x_B \le 1$, $x_A + x_C \le 1$. Which allocation should we pick?

Criterion	What is optimized?	(x_A, x_B, x_C)	Total
Maximum total flow	Σx_i	(0, 1, 1)	2 (A starves)
Max-min fairness (egalitarian)	Raise the smallest share as far as possible, then the next	(1/2, 1/2, 1/2)	1.5
Proportional fairness (Nash social welfare)	Maximize the product of shares (Kelly, 1997)	(1/3, 2/3, 2/3)	≈ 1.67

Under proportional fairness A, which uses both links, gets less, but nobody drops to zero: a zero share zeroes the product. Maximizing $x_A(1 - x_A)^2$ is the same as maximizing $\log x_A + 2\log(1 - x_A)$; since the derivative of $\log x$ is $1/x$, $1/x_A = 2/(1 - x_A) \Rightarrow x_A = 1/3$. Kelly et al. (1998) interpret TCP-like congestion control as a distributed algorithm that converges to this allocation via per-link congestion prices.

Discussion Questions

- When navigation apps suggest the “fastest” route to everyone, do we approach the Wardrop equilibrium or the system optimum? Why?
- A city plans a new bridge. To foresee a Braess effect, how do we compute the equilibrium drivers will reach once it opens?
- Are Internet routers (BGP) selfish agents? Whose cost do they minimize?

REVIEW

Summary

Concept	Core idea	Example
Selfish routing	Everyone minimizes their own delay	3,000 cars, two routes
Wardrop equilibrium	Used routes are equal and fastest	2,000 / 1,000, 25 min
System optimum	Marginal costs are equal	1,500 / 1,500, 71,250 car·min
Braess's paradox	New road → everyone slower	65 → 80 min
Rosenthal's potential	$\Delta\Phi$ = the mover's cost change	Φ: 26 → 23 → 21 → 20
Network formation	Link cost ↔ reach	α = 1.5: star / cycle equilibria, complete graph optimal

Equilibrium and optimum often diverge: a driver looks at their own delay, a planner at marginal cost.

REVIEW

Sample Problems

1. 6,000 cars; Route 1: $\ell_1 = 10 + x_1/200$, Route 2: $\ell_2 = 25 + x_2/100$. Wardrop equilibrium?
2. System optimum of the same network?
3. Braess network (x/100 and 45 min), 6,000 cars: equilibrium time before and after the new road?
4. Φ at the atomic example's equilibrium (bottom, zig, zig, top)? Change if driver 4 switches to zig?
5. $n = 4$, $\alpha = 0.5$: is the complete graph an equilibrium? The star?
6. Problem 1: which x does `find_eq` try in two rounds, and the interval?

▶ **Show answers**

References

- T. Roughgarden, *Selfish Routing and the Price of Anarchy*, MIT Press, 2005.
- N. Nisan, T. Roughgarden, É. Tardos, V. Vazirani (eds.), *Algorithmic Game Theory*, Cambridge University Press, 2007 (Chapters 18–19).
- J. G. Wardrop, “Some Theoretical Aspects of Road Traffic Research”, *Proc. Institution of Civil Engineers*, 1952.
- D. Braess, “Über ein Paradoxon aus der Verkehrsplanung”, *Unternehmensforschung*, 1968.
- R. W. Rosenthal, “A Class of Games Possessing Pure-Strategy Nash Equilibria”, *Int. J. Game Theory*, 1973.
- D. Monderer, L. Shapley, “Potential Games”, *Games and Economic Behavior*, 1996.
- A. Fabrikant et al., “On a Network Creation Game”, *PODC*, 2003. · M. Jackson, A. Wolinsky, “A Strategic Model of Social and Economic Networks”, *J. Economic Theory*, 1996.

REVIEW

References: Papers, Books and Courses

- A. Fabrikant, C. Papadimitriou, K. Talwar, “The Complexity of Pure Nash Equilibria”, *STOC*, 2004. · M. Beckmann, C. B. McGuire, C. B. Winsten, *Studies in the Economics of Transportation*, Yale University Press, 1956.
- A. Rapoport, T. Kugler, S. Dugar, E. J. Gisches, “Choice of Routes in Congested Traffic Networks: Experimental Tests of the Braess Paradox”, *Games and Economic Behavior* 65(2), 2009. · W. Knödel, *Graphentheoretische Methoden und ihre Anwendungen*, Springer, 1969. · G. Kolata, “What if They Closed 42d Street and Nobody Noticed?”, *The New York Times*, December 25, 1990.
- I. Milchtaich, “Network Topology and the Efficiency of Equilibrium”, *Games and Economic Behavior* 57, 2006. · T. Roughgarden, “On the Severity of Braess's Paradox”, *J. Computer and System Sciences* 72, 2006.
- F. Kelly, “Charging and Rate Control for Elastic Traffic”, *European Trans. Telecommunications* 8, 1997. · F. Kelly, A. Maulloo, D. Tan, *J. Operational Research Society* 49, 1998.
- D. Easley, J. Kleinberg, *Networks, Crowds, and Markets*, Cambridge University Press, 2010 (Chapter 8: traffic networks and Braess; Chapter 17: network effects).
- A. R. Karlin, Y. Peres, *Game Theory, Alive*, American Mathematical Society, 2017 (Price of Anarchy and traffic routing).
- Course pages: **T. Roughgarden, CS364A (Stanford)** · **M. Hoefer, AGT (Frankfurt)** · **M. Dinitz, AGT (Johns Hopkins)**

The End

Chapter 8: Network Design and Routing

Previous: Chapter 7: Social Choice and Voting · Next: Chapter 9: The Price of Anarchy