Linear Programming

A beginner’s guide to linear optimization

Linear programming

- Recall that a function $f: \mathbb{R}^n \to \mathbb{R}^m$ is **linear** if $f(\lambda x + \mu y) = \lambda f(x) + \mu f(y)$ for all $\lambda, \mu \in \mathbb{R}$ and $x, y \in \mathbb{R}^n$
- The one-dimensional case is very easy: all linear functions are $f(x) = \lambda x$ for some $\lambda \in \mathbb{R}$
- Vectors and matrices give us the power to neatly express the high-dimensional versions: every linear function $f: \mathbb{R}^n \to \mathbb{R}^m$ can be described by an $n \times m$ matrix A where for a (column) vector $x \in \mathbb{R}^n$ we have $f(x) = Ax$

- A **linear program** has
  - **variables**, conventionally called $x_1, \ldots, x_n$, and usually we insist on $x_j \geq 0$
  - **constraints** of the form $\sum_j a_{i,j} x_j \leq b_i$, $\sum_j a_{i,j} x_j = b_i$, or $\sum_j a_{i,j} x_j \geq b_i$,
  - an **objective:** maximize or minimize $\sum_j c_j x_j$

The college student diet problem

- There are n foods available at the campus cafeteria
- There are m macronutrients with minimum daily requirements
- For obvious reasons, you want to meet your nutritional needs for **minimum cost** and this cafeteria prices food linearly by e.g. weight

- Let $a_{i,j}$ be the amount of nutrient i present in food j
- If you eat x_j units of food j then you get $a_{i,j}x_j$ units of nutrient i from food j
- The total amount of nutrient i is therefore $\sum_j a_{i,j}x_j$
- We want this to exceed the daily minimum b_j
- If food j costs c_j dollars then we spend $\sum_j c_j x_j$
- Goal: minimize $\sum_j c_j x_j$ such that $\forall j.\ x_j \geq 0$ and $\forall i.\ \sum_j a_{i,j}x_j \geq b_i$

Linear algebra recap

- $\mathbb{R}^n$ is the set of vectors of length n whose entries are real numbers
- $\mathbb{R}^{n \times m}$ is the set of matrices with n rows and m columns whose entries are real numbers
- You can think of a column vector in $\mathbb{R}^n$ as a $n \times 1$ matrix
- You can think of a row vector in $\mathbb{R}^n$ as a $1 \times n$ matrix
- Indices i, j represent the ith row and the jth column
- If you have an $n \times r$ and a $r \times m$ matrix then you can multiply them and the answer is an $n \times m$ matrix
- Entry-wise we have $(AB)_{i,j} = \sum_k A_{i,k} B_{k,j}$
- Transposing: $(A^T)_{i,j} = A_{j,i}$

Linear algebra

- If we interpret $A = (a_{i,j})_{1 \le i \le m, 1 \le j \le n}$ as a matrix and $c = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}, b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$ as (column) vectors then a linear programming problem becomes:
- $\max c^T x$ such that $x \geq 0$ and $Ax \leq b$
- Each row of the system of equations represented by $Ax \leq b$ is a constraint

- Without loss of generality we can focus on maximizing as minimizing $c^T x$ is the same as maximizing $-c^T x$
- We can restrict to upper bound constraints as $\sum_j a_{i,j} x_j \leq b_j \Leftrightarrow -\sum_j a_{i,j} x_j \geq -b_j$ and $a = b \Leftrightarrow (a \leq b) \wedge (-a \leq -b)$
- We only need nonnegative variables as $x_j < 0 \Leftrightarrow -x_j > 0$ and if $x_j \in \mathbb{R}$ we can represent it as $x'_j - x''_j$ for $x'_j, x''_j \geq 0$
- In practice we use whatever is convenient

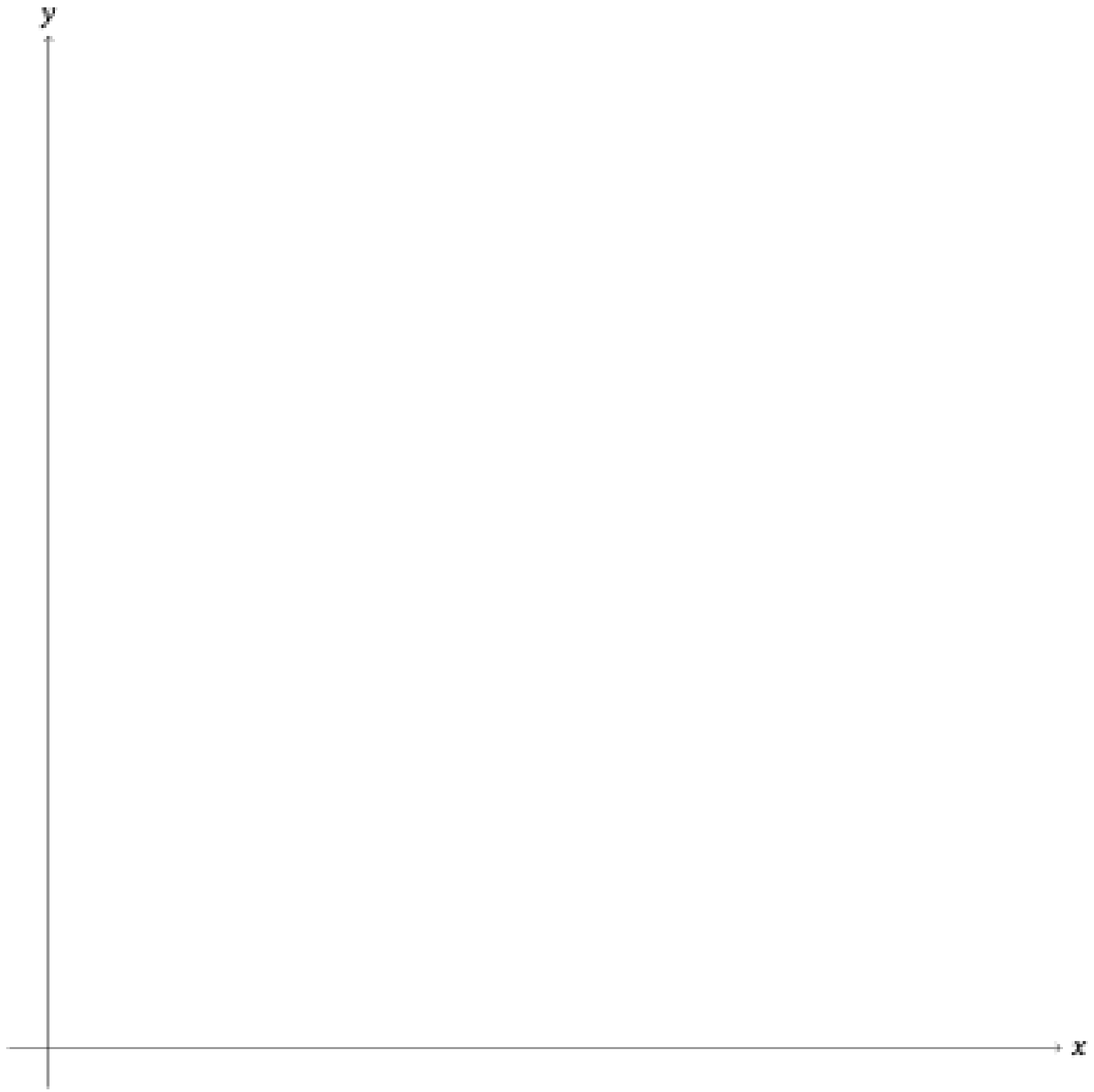
A geometric example

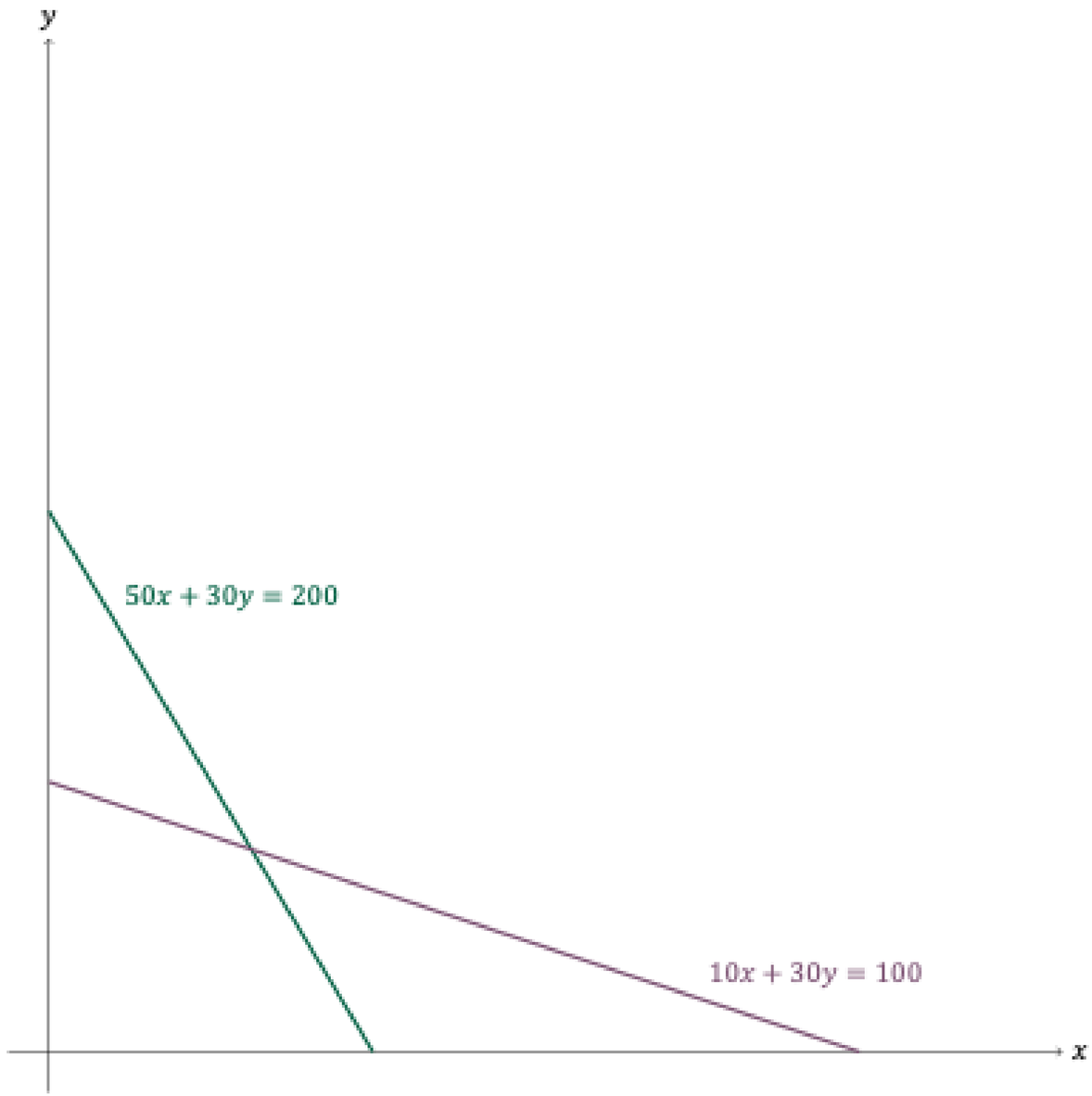
- Suppose that we have two foods available: **pasta** and **stew**
- Suppose that we track **carbs** and **protein** and want to eat at least 200g carbs and 100g protein

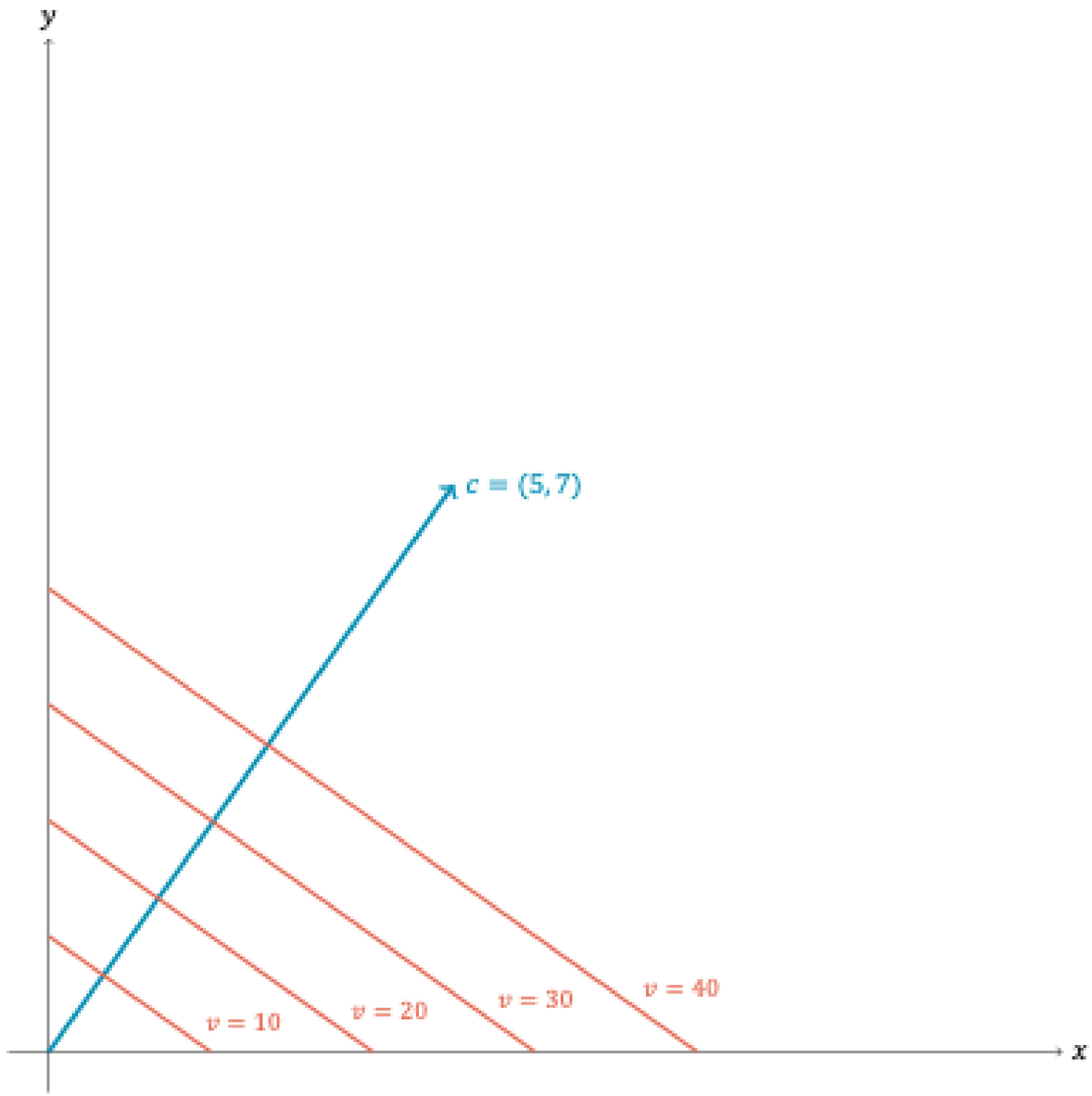
- Pasta: 50g carbs, 10g protein per 100g
- Stew: 30g carbs, 30g protein per 100g

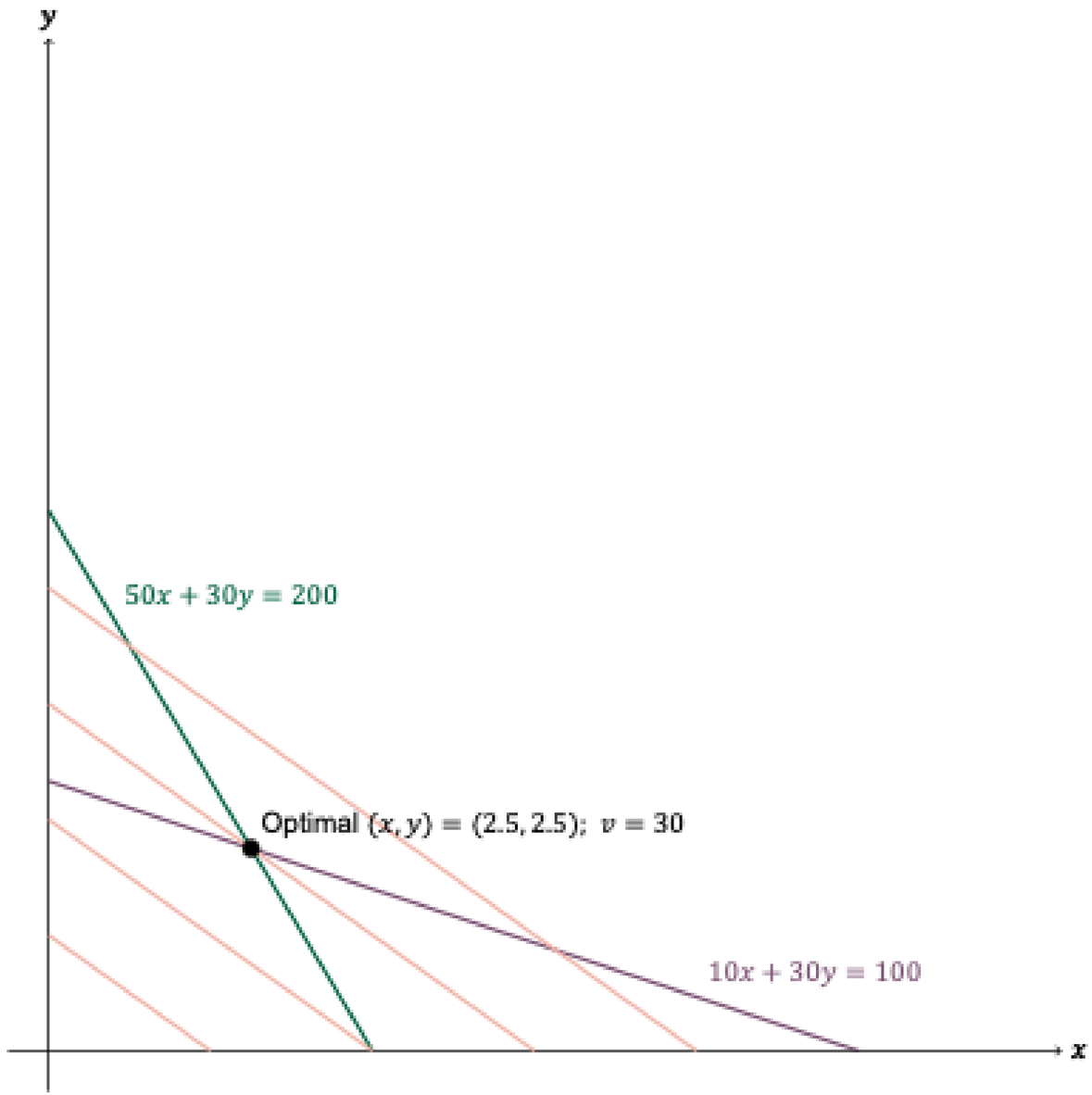
- Suppose that x is the amount of pasta and y is the amount of stew that we eat, in 100g units. Then a decision about what to eat corresponds to a point in (x, y)-space. We also have $x \geq 0$ and $y \geq 0$.
- Our nutrient needs form constraints: for carbs we want $50x + 30y \geq 200$ and for protein $10x + 30y \geq 100$
- Suppose pasta costs \$5 per 100g and stew \$7 per 100g. Then we must minimize $5x + 7y$

Examples




y
50x + 30y = 200
10x + 30y = 100
x


y
c = (5, 7)
v = 10
v = 20
v = 30
v = 40
x


y
50x + 30y = 200
Optimal (x, y) = (2.5, 2.5); v = 30
10x + 30y = 100
x

Linear program definitions

- A **linear program** is an optimization problem of the form $\max c^T x$ such that $x \geq 0$, $Ax \leq b$
- There is some flexibility in the presentation
- The **objective** is $c^T x$ (and the **sense** is min or max)
- A vector x is **feasible** if the constraints $x \geq 0$ and $Ax \geq b$ hold
- A vector x is **optimal** if it is feasible and it optimizes the objective in the correct sense
- If we think of $A = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}$ where a_i is a (row) vector representing the ith constraint $a_i x \leq b_i$, the constraint is **tight** if it holds with **equality**

Linear program problems

- Consider $\max x$ such that $x \geq 0$
- What's the solution?

This LP is **unbounded**

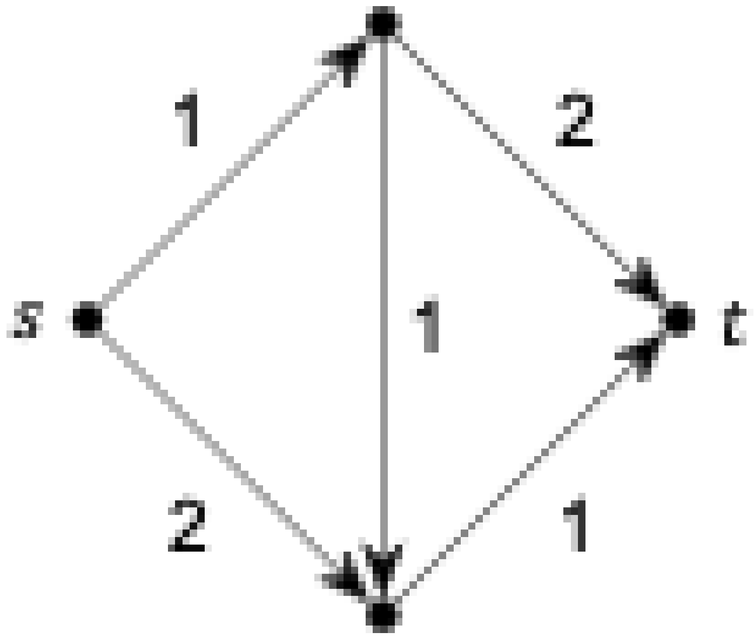
- Consider $\max x$ such that $x \geq 0$ and $x \leq -1$
- What's the solution?

This LP is **infeasible**

Important special cases

- Linear programming originated in the 20th century and history provides some important use cases
- Two of the most important are **max-flow** and **min-cut**
- Another is **maximum matching in a bipartite graph**

- Real-world problems are not always linear, but sometimes one can approximately solve hard problems with a **linear relaxation** and still get somewhere
- This leads us to **rounding LPs**



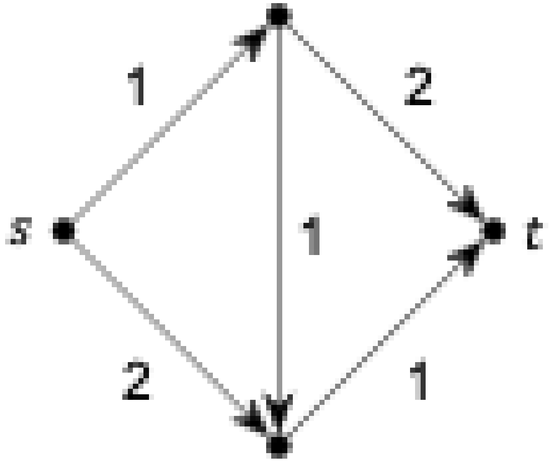
Vertices:	cities
Directed edges:	railroads
Capacity on the edges:	amount of goods the specific railroad can transport each day

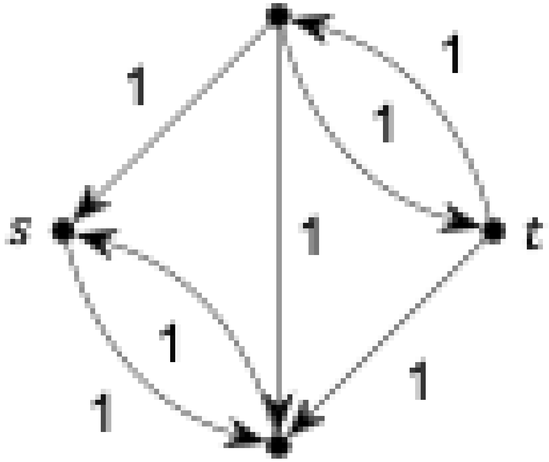
The bottleneck is transport capacity, not production or consumption. There is no available storage at any city.

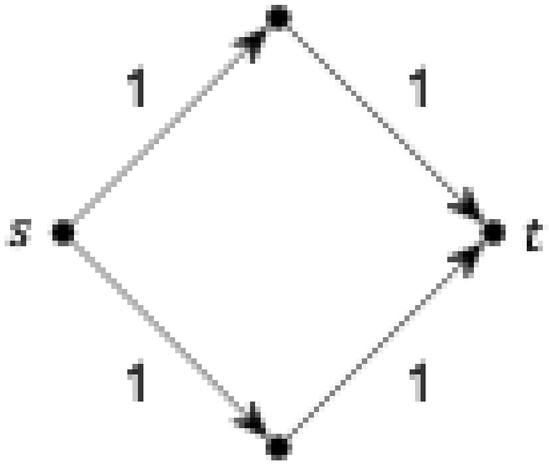
Flows in networks

- A set of vertices V
- A capacity function $c: V \times V \to [0, \infty)$ with $c(v, v) = 0$ for all v
- A **source** s and a **target** t (sometimes called a sink)
- A **flow** is a function $f: V \times V \to [0, \infty)$ such that for all vertices $v \in V \setminus \{s, t\}$ we have $\sum_u f(u, v) = \sum_w f(v, w)$
- A flow is **feasible** if $f(u, v) \leq c(u, v)$ for all u, v
- The **value** of a flow is $|f| = \sum_{v \in V} \big(f(s, v) - f(v, s)\big)$

- **Goal:** find a feasible flow of maximum value


1
2
s
1
t
2
1


1
1
1
s
1
t
1
1
1


1
1
s
t
1
1

Residual flow networks

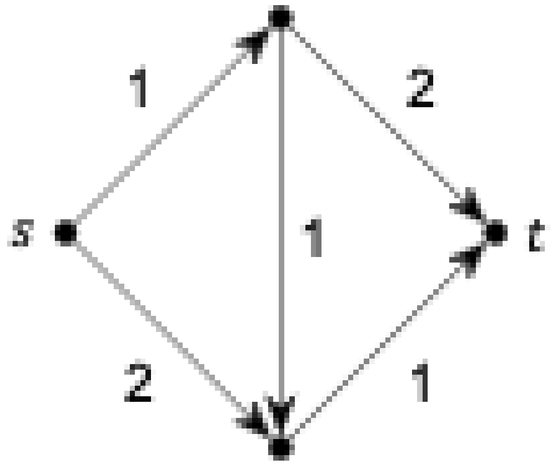
Given a flow network $G = (V, c, s, t)$ and a feasible flow f, the **residual flow network** G_f is constructed as follows

- Draw each original edge (u, v) with its **remaining capacity** $c(u, v) - f(u, v)$
- Draw an edge (v, u) with capacity $f(u, v)$ representing **reversible flow**
- To make things simpler, do not draw edges of zero capacity

The importance of residual flow

- Imagine a greedy algorithm for making flows
- Start with $f = 0$ everywhere
- While there is a path P from s to t (found e.g. by BFS) in (V, c):
  - Let f' be the flow that sends flow equal to the minimum capacity of an edge in P along P
  - Let $f = f + f'$
  - Reduce capacities by f'

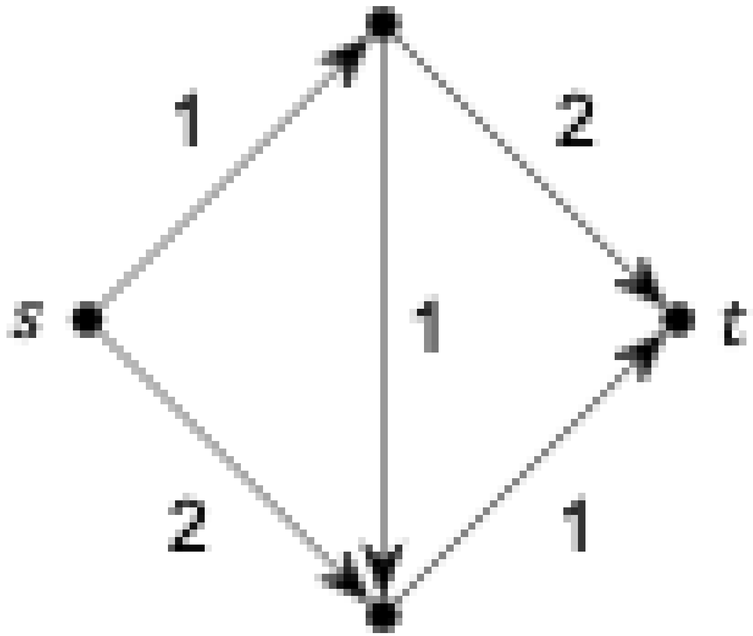
- What's wrong with this algorithm?


1
2
s
1
t
2
1

Ford–Fulkerson

- Start with $f = 0$ everywhere
- While there is a path P from s to t (found e.g. by BFS) in the **residual flow network** G_f
  - Let f' be the flow that sends flow equal to the minimum capacity of an edge in P along P
  - Let $f = f + f'$

- It turns out that this “fixes” the greedy algorithm
- Considering the residual flow network gives us the ability to reverse bad decisions
- If the while loop terminates then we have a maximum flow, and we can prove this



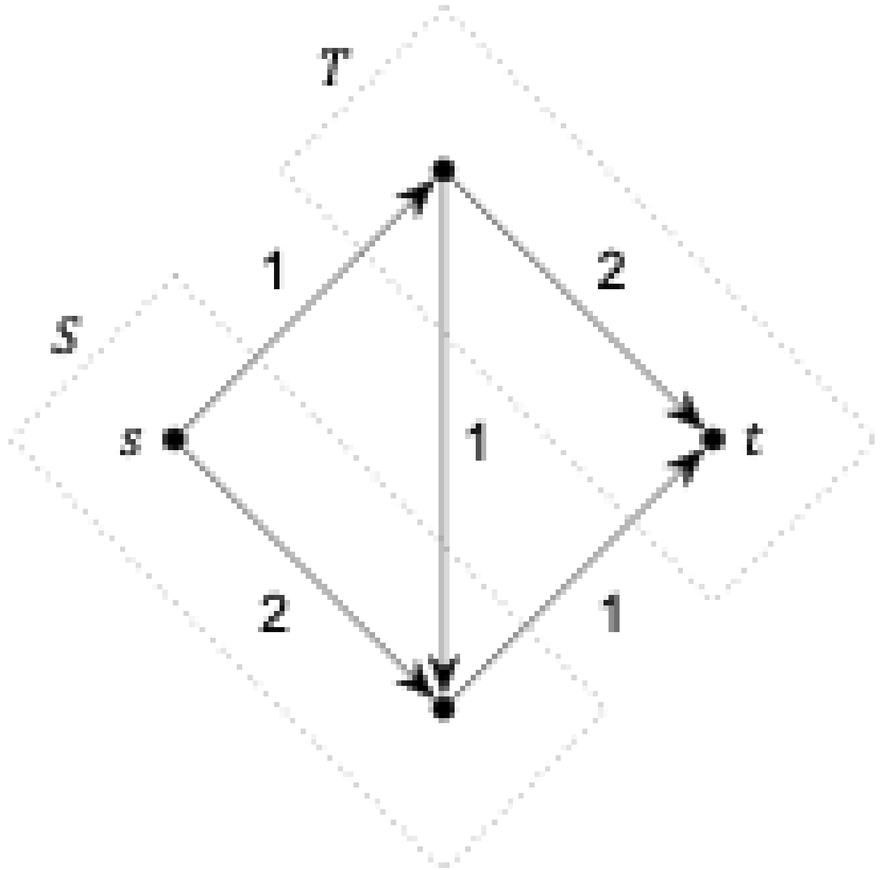
Vertices: cities
Directed edges: railroads
Capacity on the edges: amount of goods the specific railroad can transport each day

Suppose that you want to destroy some railroads so that no goods can move from s to t, and the cost of destruction corresponds to the capacity of the railroads involved

Cuts in networks

- A set of vertices V
- A capacity function $c: V \times V \to [0, \infty)$ with $c(v, v) = 0$ for all v
- A **source** s and a **target** t (sometimes called a sink)
- A **cut** is a partition $V = S \cup T$ such that $s \in S$, $t \in T$, and $S \cap T = \emptyset$
- The **cost** of a cut is $\|S, T\| = \sum_{u \in S} \sum_{v \in T} c(u, v)$

- Goal: find a cut of minimum cost



The max-flow min-cut theorem

Theorem (Max-flow min-cut).

Let $G = (V, c, s, t)$ be a flow network with capacities, source and target and let f be a feasible flow in G. Then the following are equivalent:

a) The flow f is maximum

b) The “residual flow network” G_f has no (s, t)-path

c) The value of f is equal to the cost of some cut in G

Proof.

Note that a) ⇒ b) is trivial as we can increase $|f|$ as in FF if there is an (s, t)-path in G_f

Flow value computed across cuts I

Lemma 1. For any flow f and cut (S, T) we have $|f| = \sum_{u \in S} \sum_{v \notin S} f(u, v) - \sum_{u \notin S} \sum_{v \in S} f(u, v)$.

Proof.

The flow conservation constraints mean that for any $u \in V \setminus \{s, t\}$ we have

$$0 = \sum_{v \in V} \big(f(u, v) - f(v, u)\big)$$

The definition of value gives

$$|f| = \sum_{v \in V} \big(f(s, v) - f(v, s)\big)$$

Flow value computed across cuts II

From before: for any $u \in V \setminus \{s,t\}$ we have $0 = \sum_{v \in V}\big(f(u,v) - f(v,u)\big)$ and $|f| = \sum_{v \in V}\big(f(s,v) - f(v,s)\big)$

If we add the first one for each $u \in S \setminus \{s\}$ we get

$$|f| = \sum_{v \in V}\sum_{u \in S}\big(f(u,v) - f(v,u)\big).$$

The cut satisfies $S \cap T = \emptyset$ and $S \cup T = V$ so we can split the sum over $v \in V$ into sums over S and T

$$|f| = \sum_{v \in S}\sum_{u \in S}\big(f(u,v) - f(v,u)\big) + \sum_{v \in T}\sum_{u \in S}\big(f(u,v) - f(v,u)\big)$$

But the first sum is zero and the second simplifies to exactly what we want. □

An upper bound on flow value

Lemma 2. For any feasible flow f and cut (S, T) we have $|f| \leq \|S, T\|$.

Proof. We just use Lemma 1 and the feasibility constraints.

$$|f| = \sum_{u \in S} \sum_{v \notin S} f(u, v) - \sum_{u \notin S} \sum_{v \in S} f(u, v) \leq \sum_{u \in S} \sum_{v \notin S} c(u, v) = \|S, T\|. \square$$

This means that c) ⇒ a) in the MFMCT. What's left is to prove that b) ⇒ c).

Finishing the proof

Lemma 3. Let G_f have no (s,t)-path. Then there is a cut (S,T) with $\|S,T\| = |f|$.

Proof.

Let S be the set of nodes reachable from s in G_f. By assumption, we have $t \notin S$, so let $T = V \setminus S$. This is a valid cut.

Let (u,v) be an edge out of S. Then $f(u,v) = c(u,v)$, otherwise v would be reachable from s in G_f.

Let (u,v) be an edge into S. Then $f(u,v) = 0$ otherwise v would be reachable from s in G_f.

But then all the inequalities in Lemma 2 are tight, so $|f| = \|S,T\|$. □

Consequences for FF

Does this mean Ford–Fulkerson is a good algorithm?

A modern perspective

- These problems are all just LPs
- There are many powerful and mature LP solvers available for use in software
- These days you might just solve max-flow or min-cut with a general-purpose LP solver…

- It’s reasonable to ask what an LP solver does
- In this class we focus on **applications** of LPs rather than **solving** the LPs
- Modern solvers implement some combination of **interior point** and (**dual-)simplex** algorithms

- The max-flow min-cut theorem has a broad generalization called the **LP duality theorem**

Flows as LPs

Given a flow network $G = (V, c, s, t)$, to see max-flow as an LP:

- For each $(u, v) \in V^2$ we have a variable $f(u, v)$, and the assumption that $f(u, v) \geq 0$
- For each $(u, v) \in V^2$ we have the linear capacity constraint $f(u, v) \leq c(u, v)$
- For each vertex $u \in V \setminus \{s, t\}$ we have the flow conservation constraint $\sum_v \big(f(u, v) - f(v, u)\big) = 0$
- We want to maximize $|f| = \sum_v \big(f(s, v) - f(v, s)\big)$ which is a linear function of the variables

Cuts as LPs

To see min-cut as an LP is quite a lot harder:

- We want variables $0 \leq p(u) \leq 1$ indicating which side of the cut u is on: $p(u) = 1$ for $u \in S$ and $p(u) = 0$ for $u \in T$
- Then we should have $p(s) = 1$ and $p(t) = 0$ (they're not really variables, but this helps the notation)
- To measure the cost of the cut we want variables $d(u, v) \geq 0$ for each edge $(u, v) \in V^2$
- Given some p variables that in $\{0,1\}$, minimizing $d(u, v)$ subject to the constraints $d(u, v) \geq p(u) - p(v)$ means that we will score $d(u, v) = 1$ for edges from S to T and $d(u, v) = 0$ otherwise.
- The objective is to minimize $\sum_{(u,v)\in V^2} d(u, v)c(u, v)$

- It's rather unclear that we can get away with allowing $p(u) \in [0,1]$ instead of $p(u) \in \{0,1\}$
- We will learn more general ways to understand this

Reductions

- A **decision** problem is one that has a yes/no answer
- When we have a solution to a computational problem, we might want to recycle it and solve other problems
- This leads to the concept of a **reduction:**
  1. Given an instance X of a **decision problem** A
  2. Perform some computation to convert it into an instance Y of **decision problem** B
  3. Solve instance Y using an algorithm for problem B
  4. Convert the solution to Y into the solution for problem X
- Reductions have some technical details such as the amount of computation allowed in step 2, and there are versions of reductions for optimization and counting problems

Maximum bipartite matching

- Suppose that we have a set $u_1, u_2, \ldots u_k$ of **input ports** and a set $v_1, v_2, \ldots, v_\ell$ of **output ports**
- Each input port has a queue of packets to send, each of one of which has a specific output port to go to
- Ports can only send/receive one packet in a clock cycle and packets must not collide
- How can we send the most information in one cycle?
- Model the problem as a graph: connect u_i to each output port v_j which is the destination of a packet in u_i's queue
- Now we want to find the **maximum matching**: the largest collection of edges M such that no vertex is incident to more than one edge in M

Maximum bipartite matching

- A graph $G = (V, E)$ is **bipartite** if the vertex set V has a partition into two parts $V = A \cup B$ such that all edges lie between A and B
- This is the same as being 2-colorable in the sense of proper vertex coloring
- A **matching** is a set of edges $M \subseteq E$ with the property that no vertex is incident to more than one edge in M
- A **maximum** matching is a matching M^* with the property that for all matchings M, $|M| \leq M^*$

- How can we find maximum matchings?

Iteratively improving matchings

- The philosophy of Ford–Fulkerson is that you can find max-flows by carefully iteratively improving flows
- It turns out that this works for matchings
- Given a graph $G = (V, E)$ and a matching M in G, we say a vertex $v \in V$ is **matched** if its incident to an edge in M, otherwise it is **free**
- Given a matching M in G, an M-**augmenting** path is a path in G that
  - Starts and ends at free vertices
  - Edges alternate between being in M and not in M along the path (the path is M-**alternating**)
- If you find an M-augmenting path P (represented by its set of edges), then you can increase the size of your matching by moving to the symmetric difference $M \oplus P$

Examples

Maximum bipartite matching

How can we find maximum matchings?

We need to prove that this works…

Examples

Reducing bipartite matching to flows I

- Given a bipartite graph $G = (A \cup B, E)$
- Orient all edges from A to B and give them capacity 1
- Add a source vertex s and connect it to each $u \in A$ with an edge of capacity 1
- Add a target vertex t and connect each $v \in B$ to it with an edge of capacity 1
- Find the max-flow

Theorem. Let G be a bipartite graph and let G' be the flow network constructed as above. Then the size of a maximum matching in G is the same as the value of a maximum flow in G'.

Proof. Let m be the size of a maximum matching in G and let m' be the value of a maximum flow in G'. It is enough to show both $m \leq m'$ and $m' \leq m$.

Reducing bipartite matching to flows II

To see that $m \leq m'$, take any matching M in G. Construct a flow in G' by putting one unit of flow from s to each $u \in A$ that is matched, and one unit of flow from each $v \in B$ that is matched to t. Each edge of the matching also pushes 1 unit of flow. It is easy to check that this is a valid and feasible flow of value $|M|$.

To see that $m' \leq m$ we use FF. We can prove by induction on the value of the flow in a flow network that if the capacities are all integers, then FF can only construct flows with integer values. This means that in our G' we can find a maximum flow f with flows in $\{0,1\}$ along any edge.

At most one unit of flow can enter any $u \in A$ and at most one unit can leave any $v \in B$ by construction, so the number of saturated edges from A to B is the same as the flow value. And these saturated edges must form a matching in G by the flow conservation constraints. Then from f we can construct a matching in G of size m', and hence $m' \leq m$. □

The benefits of a reduction

We already figured out that we can essentially implement the FF idea directly for bipartite matchings via M-augmenting paths. But it wasn't clear that we'd never get stuck.

This was also tricky for FF where we had to consider residual flow to avoid getting stuck.

By the previous theorem, which gives a reduction from maximum bipartite matching to maximum flow, we have essentially proved that a matching M in a bipartite graph G is maximum if and only if there is no M-augmenting path.

But the reduction seemed easier than proving this from scratch! Try to prove it yourself…

Integer linear programs

- Many combinatorial optimization problems are not linear, but can be represented as **integer programs**
- We already thought about finding large independent sets:
  - Given a graph $G = (V, E)$, find $S \subseteq V$ of largest size such that $\forall uv \in E$ at most one of u, v is in S
- Encode S via $x_u = 1$ for $u \in S$ and $x_u = 0$ otherwise

Maximize $\sum_u x_u$ such that $0 \leq x \leq 1$, $\forall uv \in E.\ x_u + x_v \leq 1$, and $x \in \mathbb{Z}$

This looks just like an LP in the sense that the objective and constraint are linear

But the variables are not a real vector $x \geq 0$, the entries are forced to be in $\mathbb{Z}$ (in fact, in $\{0,1\}$)

Relaxation

- An **integer linear program (ILP)** is $\max x$ such that $x \geq 0$, $Ax \leq b$, and $x \in \mathbb{Z}^n$
- The **linear relaxation** of an ILP is the LP you get by dropping the condition $x \in \mathbb{Z}^n$
- This can **enlarge** the feasible set, so the solution to the LP is always at least the solution to the ILP
- Sometimes it is useful to solve the relaxation and then **round** the solution back to an integer
- Sometimes no rounding is necessary

ILPs for combinatorial optimization

Let $G = (V, E)$ be a graph

- Find $\max \sum_{uv \in E} x_{uv}$ such that $\forall u \in V$, $\sum_{v \in N(u)} x_{uv} \leq 1$ and $x_{uv} \in \{0,1\}$
- Find $\max \sum_{u \in V} x_u$ such that $\forall uv \in E$, $x_u + x_v \leq 1$ and $x_u \in \{0,1\}$
- Find $\min \sum_I x_I$ such that for all $u \in V$, $\sum_{I \ni v} x_I \geq 1$ and $x_I \in \{0,1\}$

- These problems have linear objectives, linear constraints and integer variables

Relaxing ILPs

Let $G = (V, E)$ be a graph

- Find $\max \sum_{uv \in E} x_{uv}$ such that $\forall u \in V$, $\sum_{v \in N(u)} x_{uv} \leq 1$ and $x_{uv} \in [0,1]$
- Find $\max \sum_{u \in V} x_u$ such that $\forall uv \in E$, $x_u + x_v \leq 1$ and $x_u \in [0,1]$
- Find $\min \sum_I x_I$ such that $\forall u \in V$, $\sum_{I \ni v} x_I \geq 1$ and $x_I \in [0,1]$

Problems?

Examples