

Integer Programming

Aayush Bajaj

July 10, 2026

Contents

1	forms	1
1.1	why NP-hard	2
2	LP relaxation and the integrality gap	2
3	branch and bound	2
3.1	worked example	3
4	cutting planes	3
5	total unimodularity	4
6	modelling tricks	4
7	applications	5
7.1	knapsack	5
7.2	travelling salesman, two ways	5
7.3	sudoku	5
8	code	5
9	see also	6

take a **linear program** and add one word — *integer* — and the complexity class jumps from polynomial to NP-hard. that word buys expressive power nothing continuous can match: yes/no decisions, either/or logic, fixed costs, sequencing, assignment. integer programming is the lingua franca of operations research precisely because “decide” is not a convex verb.

commercial solvers — gurobi, cplex, scip — routinely crack instances with millions of variables, np-hardness notwithstanding. hard in the worst case is not hard on your case

1 forms

$$\begin{array}{ll}\text{maximise} & c^\top x \\ \text{subject to} & Ax \leq b \\ & x \geq 0, \quad x \in \mathbb{Z}^n,\end{array}$$

- **ILP** (pure): all variables integer.
- **MILP** (mixed): some integer, some continuous — the industrial workhorse.
- **BIP** (binary): $x \in \{0, 1\}^n$ — decisions in their purest form.

1.1 why NP-hard

binary ILP feasibility is one of karp's original 21 NP-complete problems. the reduction is almost insulting in its brevity: given a 3-sat formula, make a binary variable x_v per boolean variable and translate each clause directly —

$$(v_1 \vee \neg v_2 \vee v_3) \quad \longmapsto \quad x_1 + (1 - x_2) + x_3 \geq 1. \quad (1)$$

the formula is satisfiable iff the 0-1 system is feasible. so a polynomial ILP algorithm would collapse P vs NP; see the NP-completeness chapter of clrs for the surrounding theory (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009). the practical reading: no algorithm is safe on all instances, so the field's energy goes into bounds, cuts and heuristics that make *your* instance fast.

2 LP relaxation and the integrality gap

drop the integrality constraint and you get the **LP relaxation** — the same problem over the convex feasible polyhedron. two facts do all the work:

- the relaxation's optimum is an **upper bound** (maximising) on the ILP optimum: fewer constraints, no worse objective.
- if the relaxation's optimum happens to be integral, it solves the ILP outright.

the *integrality gap* is the ratio (or difference) between the two optima. it measures how much lying the relaxation does — and therefore how hard branch-and-bound will have to work, since all its pruning power comes from relaxation bounds. rounding the relaxation, by contrast, is not a strategy: the nearest integer point to the LP optimum can be infeasible or arbitrarily far from optimal in high dimension.

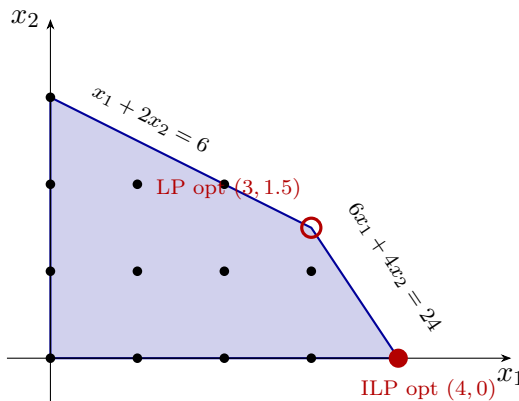


Figure 1: LP relaxation vs the integer lattice for the worked ILP. the relaxation optimises over the shaded polytope and lands at the fractional vertex (3, 1.5) (open circle); the integer optimum (4, 0) (filled red) is a different vertex entirely — rounding (3, 1.5) to (3, 1) or (3, 2) finds $z = 19$ or infeasibility, never $z = 20$.

3 branch and bound

divide and conquer with certificates. maintain an *incumbent* (best integer solution found) and a tree of subproblems:

- **solve** the node's LP relaxation.

- **fathom by infeasibility** if the relaxation is infeasible.
- **fathom by bound** if the relaxation optimum is no better than the incumbent — nothing in this subtree can win.
- **fathom by integrality** if the relaxation is integral — update the incumbent.
- **branch** otherwise: pick a fractional $x_j = f$, split into children $x_j \leq \lfloor f \rfloor$ and $x_j \geq \lceil f \rceil$. no integer point is lost, the fractional one is.

3.1 worked example

maximise $5x_1 + 4x_2$ subject to $6x_1 + 4x_2 \leq 24$, $x_1 + 2x_2 \leq 6$, $x \in \mathbb{Z}_{\geq 0}^2$. running the exact algorithm (code at the end of this page ran it; every number below is from that run):

- **root**: relaxation gives $(3, 1.5)$, $z = 21$. fractional x_2 — branch.
- **node** $x_2 \leq 1$: relaxation $(10/3, 1)$, $z = 20.67$. fractional x_1 — branch.
 - **node** $x_2 \leq 1, x_1 \leq 3$: $(3, 1)$, $z = 19$, integral — incumbent 19.
 - **node** $x_2 \leq 1, x_1 \geq 4$: $(4, 0)$, $z = 20$, integral — incumbent 20.
- **node** $x_2 \geq 2$: relaxation $(2, 2)$, $z = 18 \leq 20$ — fathomed by bound.

optimum $z^* = 20$ at $(4, 0)$: five LP solves instead of enumerating all thirteen lattice points, and the bound-fathoming at the last node is where the savings scale — on real instances whole exponential subtrees die to a single comparison.

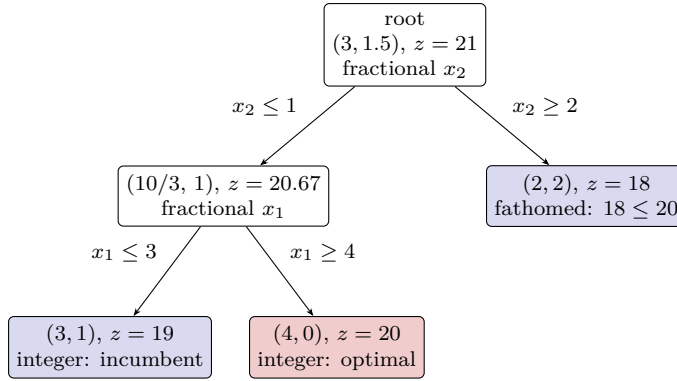


Figure 2: the branch-and-bound tree for the worked ILP. each node shows its LP relaxation; leaves are fathomed by integrality or by bound.

4 cutting planes

instead of splitting the problem, tighten it: add a linear inequality that every integer-feasible point satisfies but the current fractional LP optimum violates, then re-solve. the **gomory cut** manufactures such an inequality from any simplex tableau row with fractional right-hand side. if the optimal tableau contains

$$x_i + \sum_{j \in N} \bar{a}_{ij} x_j = \bar{b}_i, \quad \bar{b}_i \notin \mathbb{Z}, \quad (2)$$

then, writing $\{t\} = t - \lfloor t \rfloor$ for the fractional part, every non-negative *integer* solution

satisfies

$$\sum_{j \in N} \{\bar{a}_{ij}\} x_j \geq \{\bar{b}_i\}, \quad (3)$$

because rounding the coefficients down changes the left side by an integer amount, forcing the fractional parts to cover $\{\bar{b}_i\}$. the current LP vertex has all non-basic $x_j = 0$, so it violates the cut by exactly $\{\bar{b}_i\} > 0$ — sliced off, along with none of the lattice. gomory proved pure cutting planes converge finitely (1958); in practice they are slow alone but devastating in combination — modern solvers are **branch-and-cut**: a branch-and-bound tree whose node relaxations are continually tightened by cut families (gomory, knapsack covers, clique cuts, ...).

5 total unimodularity

when is the relaxation *exactly right*? a matrix A is **totally unimodular** (TU) if every square submatrix has determinant in $\{-1, 0, 1\}$. the hoffman-kruskal theorem: if A is TU and b is integral, every vertex of $\{x : Ax \leq b, x \geq 0\}$ is integral — so the LP relaxation *is* the integer program, solvable in polynomial time.

*intuition via cramer's rule:
vertex coordinates are ratios of
subdeterminants, and the
denominators are all
plus-minus one*

the two celebrity examples:

- **bipartite matching**: the incidence matrix of a bipartite graph is TU. the assignment LP therefore has integral optima, and LP duality hands you könig's theorem (max matching = min vertex cover in bipartite graphs) for free.
- **network flows**: the node-arc incidence matrix of any directed graph is TU. max-flow, min-cost flow and shortest-path LPs all inherit integer optima — the deep reason the combinatorial algorithms of **classical algorithms** return integer flows without ever being told to.

TU is the frontier of niceness: step outside it (general matching needs blossom inequalities, TSP needs exponentially many cuts) and integrality must be fought for.

6 modelling tricks

the craft of ILP is encoding logic in linear inequalities over binaries:

trick	encoding
fixed cost / big-M	$x \leq My$: pay for $y \in \{0, 1\}$ only if $x > 0$; keep M as small as validly possible or the relaxation goes mushy
indicator	$y = 1 \Rightarrow a^\top x \leq b$ via $a^\top x \leq b + M(1 - y)$
disjunction (either/or)	$a_1^\top x \leq b_1 + My$ and $a_2^\top x \leq b_2 + M(1 - y)$
logical and	$y = y_1 \wedge y_2$: $y \leq y_1, y \leq y_2, y \geq y_1 + y_2 - 1$
logical or	$y = y_1 \vee y_2$: $y \geq y_1, y \geq y_2, y \leq y_1 + y_2$
at most k of n	$\sum_i y_i \leq k$
piecewise linear	one binary (or sos2 pair) per segment selects the active piece

the recurring warning: every big-M loosens the LP relaxation, and the relaxation is your bound. tight formulations — smallest valid M , strongest valid inequalities — are worth more than faster hardware.

7 applications

7.1 knapsack

maximise $\sum_i v_i y_i$ subject to $\sum_i w_i y_i \leq W$, $y \in \{0, 1\}^n$. the LP relaxation is solved by greedy density sorting with at most one fractional item (dantzig's bound) — the code below shows exactly that structure. small-capacity instances also yield to [dynamic programming](#) in $\mathcal{O}(nW)$ — pseudo-polynomial, so both methods coexist: DP when W is small, branch-and-bound when W is astronomical but the LP bound is tight.

7.2 travelling salesman, two ways

binary x_{ij} = “tour uses arc (i, j) ”, degree constraints force one arc in and out of each city. the degree constraints alone permit *subtours* — disjoint little cycles — and the two classic fixes trade constraint count against relaxation strength:

- **DFJ (subtour elimination)**: $\sum_{i,j \in S} x_{ij} \leq |S| - 1$ for every proper subset S — exponentially many, so generate them lazily inside branch-and-cut (solve, find a violated subtour, add it, repeat). strong relaxation; this is how TSPs with tens of thousands of cities are solved to optimality.
- **MTZ (miller-tucker-zemlin)**: order variables u_i with $u_i - u_j + n x_{ij} \leq n - 1$ for $i \neq j \geq 2$ — only $\mathcal{O}(n^2)$ constraints, but big-M-flavoured and weak; fine for small instances, hopeless at scale.

same feasible set, wildly different bounds: formulation *is* the algorithm.

7.3 sudoku

pure feasibility BIP: binaries $x_{r,c,d}$ = “cell (r, c) holds digit d ”, with exactly-one constraints over cells, rows, columns and boxes, givens pinned to 1, objective zero. the constraint matrix is not TU — LP relaxations of hard puzzles go fractional — so solvers branch; the [sudoku](#) page attacks the same model with constraint propagation instead, and the comparison is instructive: same constraints, different inference engines.

8 code

`scipy.optimize.milp` (highs branch-and-cut) on a 6-item knapsack, with the LP relaxation run alongside to expose the gap:

```
1 import numpy as np
2 from scipy.optimize import milp, LinearConstraint, Bounds
3
4 # 0-1 knapsack: 6 items, capacity 15
5 values = np.array([10, 13, 18, 31, 7, 15])
6 weights = np.array([2, 3, 4, 7, 1, 3])
7 cap = 15
8
9 con = LinearConstraint(weights, -np.inf, cap)
10 bnd = Bounds(0, 1)
11
12 # integer solve
13 res = milp(-values, constraints=con, bounds=bnd,
14           integrality=np.ones(6))
15 picked = np.round(res.x).astype(int)
16 print("ILP: pick =", picked, " value =", int(-res.fun),
```

```

17     "weight =", int(weights @ picked))
18
19 # LP relaxation (same call, integrality off)
20 rel = milp(-values, constraints=con, bounds=bnd,
21           integrality=np.zeros(6))
22 print("LP : x =", np.round(rel.x, 3), " bound =", round(-rel.fun, 3))
23 print("integrality gap: LP bound - ILP value =", round(-rel.fun - (-res.fun),
    ↪ 3))

```

```

ILP: pick = [0 0 1 1 1 1] value = 71 weight = 15
LP : x = [1.    0.    1.    0.714 1.    1.    ] bound = 72.143
integrality gap: LP bound - ILP value = 1.143

```

the relaxation is textbook dantzig: items enter in value-density order until the sack is full, then item 4 gets in *fractionally* ($x_4 = 5/7 \approx 0.714$). the integer solve must instead drop item 1 entirely to fit item 4 whole — a swap no rounding of the LP would find, and the whole reason branching exists.

the branch-and-bound walkthrough earlier was produced by a 30-line pure-python solver (recursive, LP bound via `linprog` at each node); its full trace:

```

root: x=(3,1.5) z=21 fractional x2 -> branch
  root.L(x2<=1): x=(3.33,1) z=20.67 fractional x1 -> branch
    root.L(x2<=1).L(x1<=3): x=(3,1) z=19 INTEGER -> new incumbent
    root.L(x2<=1).R(x1>=4): x=(4,-0) z=20 INTEGER -> new incumbent
  root.R(x2>=2): x=(2,2) z=18 <= incumbent 20 -> fathom by bound

```

```
optimal integer solution: [ 4. -0.] z* = 20.0
```

9 see also

- [linear programming](#) — the relaxation that powers every bound above
- [quadratic programming](#) — MIQP: the same tree over curved relaxations
- [dynamic programming](#) — knapsack's other conqueror
- [sudoku as a csp](#) — the same binaries, propagated instead of relaxed
- [classical algorithms](#) — where total unimodularity quietly does the work

references

Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford (2009). *Introduction to Algorithms*, MIT Press.