

# ISyE/Math/CS 728

## Integer Optimization

### Getting Started

Prof. Jim Luedtke  
University of Wisconsin-Madison



Significant use from the book *Integer Programming* by M. Conforti, G. Cornuéjols, and G. Zambelli

# Outline

- Sec. 1.1 We introduce **integer programming** and some special cases.
- Sec. 1.2 We introduce two **algorithmic principles** that have proven successful for solving integer programs.
- Sec. 1.3 We review the conventions used in describing the **computational requirements** (operation count) of algorithms.
- Sec. 1.4 We recall the concept of **convex hull** and we introduce **perfect formulations**.

## 1.1 Integer Programming

# So what is an Integer Program?

A pure integer linear program (ILP):

$$\begin{array}{ll}\max & cx \\ \text{s. t.} & Ax \leq b \\ & x \geq 0 \quad \text{integral (i.e., } x \in \mathbb{Z}^n\text{)}.\end{array}$$

Rational data:

► Row vector  $c = (c_1, \dots, c_n)$ .

►  $m \times n$  matrix  $A = (a_{ij})$ .

► Column vector  $b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$ .

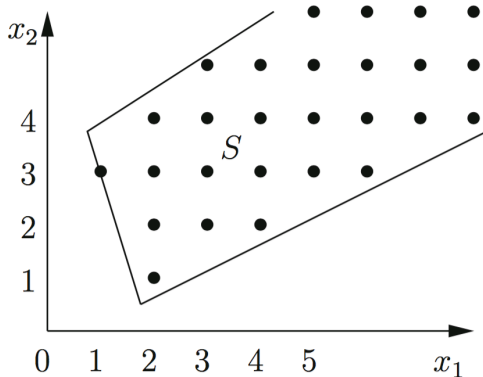
Variables:

► Column vector  $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$ .

# So what is an Integer Program?

A pure integer linear set:

$$S = \{x \in \mathbb{Z}_+^n : Ax \leq b\}.$$



# So what is an Integer Program?

A mixed integer linear program (MILP) or integer program:

$$\begin{array}{ll}\max & cx + hy \\ \text{s. t.} & Ax + Gy \leq b \\ & x \geq 0 \quad \text{integral} \\ & y \geq 0.\end{array}$$

Rational data:

- ▶  $c = (c_1, \dots, c_n)$ .
- ▶  $h = (h_1, \dots, h_p)$ .
- ▶  $A = (a_{ij})$  is an  $m \times n$  matrix.
- ▶  $G = (g_{ij})$  is an  $m \times p$  matrix.
- ▶  $b \in \mathbb{R}^m$ .

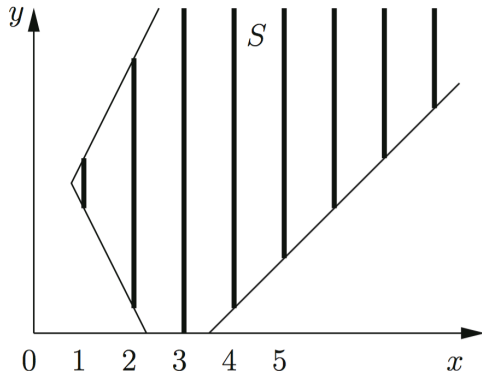
Variables:

- ▶  $x \in \mathbb{Z}^n, n \geq 1$ .
- ▶  $y \in \mathbb{R}^p$ .

## So what is an Integer Program?

A mixed integer linear set:

$$S = \{(x, y) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p : Ax + Gy \leq b\}.$$



# Other classes of Integer Programs

All variables are binary:

- ▶ 0, 1 linear program:  $\max\{cx : Ax \leq b, x \in \{0, 1\}^n\}$ .
- ▶ 0, 1 linear set:  $\{x \in \{0, 1\}^n : Ax \leq b\}$ .

Binary and continuous variables:

- ▶ Mixed 0, 1 linear program:  
 $\max\{cx + hy : Ax + Gy \leq b, x \in \{0, 1\}^n, y \in \mathbb{R}_+^p\}$ .
- ▶ Mixed 0, 1 linear set:  $\{(x, y) \in \{0, 1\}^n \times \mathbb{R}_+^p : Ax + Gy \leq b\}$ .

All variables are continuous (not an integer program):

- ▶ Linear program (LP):  $\max\{cx : Ax \leq b, x \geq 0\}$ .
- ▶ Linear set or polyhedron:  $\{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$ .



## What if we ignore the integer restrictions?

- For the mixed integer linear set

$$S = \{(x, y) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p : Ax + Gy \leq b\},$$

the natural linear relaxation is

$$P_0 = \{(x, y) \in \mathbb{R}_+^n \times \mathbb{R}_+^p : Ax + Gy \leq b\}.$$

- The natural linear programming relaxation of the mixed integer linear program

$$\max\{cx + hy : (x, y) \in S\},$$

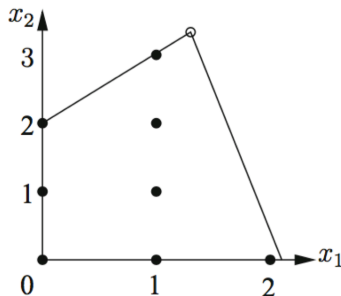
is the **linear program**

$$\max\{cx + hy : (x, y) \in P_0\}.$$

- What happens if we solve this instead?

## Example of natural LP relaxation

$$\begin{array}{ll}\max & 5.5x_1 + 2.1x_2 \\ \text{s. t.} & -x_1 + x_2 \leq 2 \\ & 8x_1 + 2x_2 \leq 17 \\ & x_1, x_2 \geq 0 \\ & x_1, x_2 \text{ integer}\end{array}$$



- ▶ The optimal solution is (1, 3) with objective value 11.8.
- ▶ The solution of the **natural LP relaxation** is (1.3, 3.3), with objective value 14.08.
- ▶ It looks like we could **round** (1.3, 3.3) to the optimum solution (1, 3)...

## Can we just round the fractional solution?

- ▶ Can we always **round** the solution of the natural LP relaxation to the optimal solution of the original integer program?
- ▶ No! There are two main problems:
  1. **Rounding** to a feasible integer solution may be difficult or impossible.
  2. The optimal solution to the natural LP relaxation can be far from the optimal solution of the original integer program.
- ▶ Can you give me examples?

## More general Linear Programming relaxations

- ▶ A linear relaxation of a mixed integer (linear) set  $S \subseteq \mathbb{Z}^n \times \mathbb{R}^p$  is a set of the form

$$P' = \{(x, y) \in \mathbb{R}^n \times \mathbb{R}^p : A'x + G'y \leq b'\}$$

that **contains**  $S$ .

- ▶ A linear programming relaxation of a mixed integer linear program

$$\max\{cx + hy : (x, y) \in S\}$$

is a linear program

$$\max\{cx + hy : (x, y) \in P'\}.$$

### Why LP relaxations?

1. We can solve these relaxations efficiently in theory and practice.
2. We can generate a sequence of linear relaxations of  $S$  that provide increasingly tighter approximations of the set  $S$ .

## 1.2 Methods for Solving Integer Programs

# Outline

Two algorithmic principles:

- ▶ The Branch-and-Bound Method.
- ▶ The Cutting Plane Method.
- ▶ Based on simple ideas.
- ▶ At the heart of state-of-the-art software.

# Methods for solving Integer Programs

Let  $S$  be a **mixed integer linear set**

$$S = \{(x, y) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p : Ax + Gy \leq b\}.$$

**Integer program (MILP):**

$$\max \{cx + hy : (x, y) \in S\}$$

**Natural LP relaxation:**

$$\max \{cx + hy : (x, y) \in P_0\}$$

We **want**:

- ▶ Optimal solution  $(x^*, y^*)$ .
- ▶ Optimal value  $z^*$ .

We **have**:

- ▶ Optimal solution  $(x^0, y^0)$ .
- ▶ Optimal value  $z_0$ .

- ▶ How does  $z_0$  compare to  $z^*$ ?  $z^* \leq z_0$
- ▶ What do we know if  $(x^0, y^0)$  has  **$x^0$  integral**?  $z^* = z_0$
- ▶ We now deal with the case  **$x^0$  not integral**.

## 1.2.1 The Branch-and-Bound method



# Branching: The “divide” in “divide-and-conquer”

**Idea of branching:** Dividing the original problem into subproblems.

- ▶ Choose an index  $j$  such that  $x_j^0 = f$  is fractional.
- ▶ Define the sets:

$$S_1 := S \cap \{(x, y) : x_j \leq \lfloor f \rfloor\}, \quad S_2 := S \cap \{(x, y) : x_j \geq \lceil f \rceil\}.$$

$x_j$  integer for every  $(x, y) \in S \Rightarrow (S_1, S_2)$  is a partition of  $S$ .

- ▶ Consider:

$$\text{MILP}_1 : \max\{cx + hy : (x, y) \in S_1\}, \quad \text{MILP}_2 : \max\{cx + hy : (x, y) \in S_2\}.$$

- ▶ Optimal solution of MILP is the best among optimal solution of  $\text{MILP}_1$  and  $\text{MILP}_2$ .
- ▶ We just need to solve the two new subproblems.

## Bounding: The “conquer” in “divide-and-conquer”

Idea of bounding: When can we discard a subproblem?

Consider the **natural linear relaxations** of  $S_1, S_2$ :

$$P_1 := P_0 \cap \{(x, y) : x_j \leq \lfloor f \rfloor\}, \quad P_2 := P_0 \cap \{(x, y) : x_j \geq \lceil f \rceil\}.$$

$$LP_1 : \max\{cx + hy : (x, y) \in P_1\}, \quad LP_2 : \max\{cx + hy : (x, y) \in P_2\}.$$

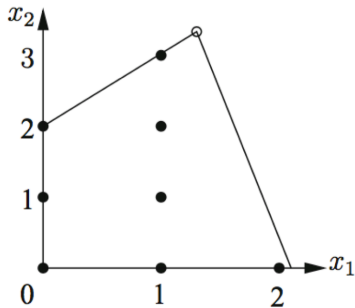
- (i) If  $LP_i$  is infeasible (i.e.,  $P_i = \emptyset$ )  $\Rightarrow S_i = \emptyset \Rightarrow \text{MILP}_i$  is infeasible. Pruned by infeasibility.
- (ii) Let  $(x^j, y^j)$  be an optimal solution of  $LP_i$ , and  $z^j$  its value.
  - (iia) If  $x^j$  is integral, then  $(x^j, y^j)$  is an optimal solution of  $\text{MILP}_i$ .  
Pruned by integrality. Moreover  $z^j \leq z^*$ .
  - (iib) If  $x^j$  is not integral, and  $z^j$  at most best known lower bound on  $z^*$ , then  $S_i$  cannot contain a better solution. Pruned by bound.
  - (iic) If  $x^j$  is not integral, and  $z^j$  is greater than best known lower bound on  $z^*$ . Let  $x_{j'}^j = f'$  be a fractional component of  $x^j$ ,

$$S_{i_1} := S_i \cap \{(x, y) : x_{j'} \leq \lfloor f' \rfloor\}, \quad S_{i_2} := S_i \cap \{(x, y) : x_{j'} \geq \lceil f' \rceil\}.$$

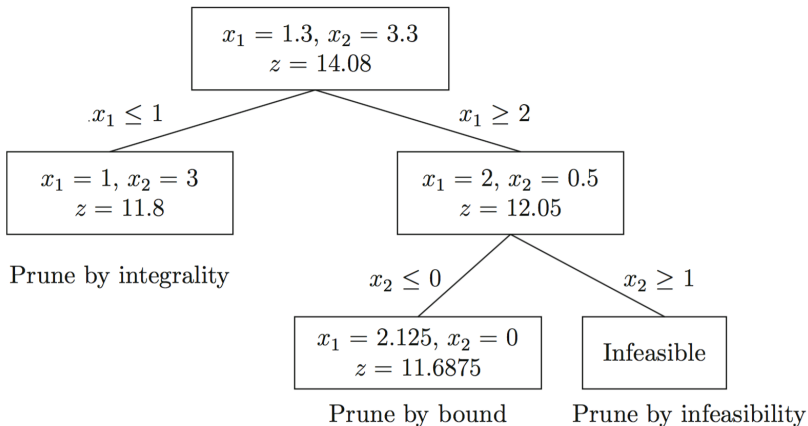
Repeat.

## Let's see an example

$$\begin{array}{ll}\max & 5.5x_1 + 2.1x_2 \\ \text{s. t.} & -x_1 + x_2 \leq 2 \\ & 8x_1 + 2x_2 \leq 17 \\ & x_1, x_2 \geq 0 \\ & x_1, x_2 \text{ integer}\end{array}$$



Let's see an example



# Formalization of branch-and-bound with LP bounding

- ▶ We keep a **list of LP problems** obtained by relaxing the integrality requirements and imposing linear constraints.
- ▶ Each LP corresponds to a **node of the enumeration tree**.

Notation:

- ▶ For a node  $N_i$ , let  $LP_i$  denote the corresponding linear program, and let  $z_i$  denote its value.
- ▶ Node  $N_0$  is associated with the LP relaxation of the original MILP problem.
- ▶ Let  $\mathcal{L}$  denote the list of nodes that must still be solved.
- ▶ Let  $\underline{z}$  denote a lower bound on the optimum value  $z^*$ .

# Formalization of branch-and-bound with LP bounding

## Branch-and-Bound Method

### 0. Initialize

$\mathcal{L} := \{N_0\}$ ,  $\underline{z} := -\infty$ ,  $(x^*, y^*) := \emptyset$ .

### 1. Terminate?

If  $\mathcal{L} = \emptyset$ , the solution  $(x^*, y^*)$  is optimal.

### 2. Select node

Choose a node  $N_i$  in  $\mathcal{L}$  and delete it from  $\mathcal{L}$ .

### 3. Bound

Solve  $LP_i$ . If it is **infeasible**, go to **Step 1**.

Else, let  $(x^i, y^i)$  be an optimal solution of  $LP_i$  and  $z_i$  its objective value.

### 4. Prune

If  $z_i \leq \underline{z}$ , go to **Step 1**.

If  $(x^i, y^i)$  is **feasible** to MILP, set  $\underline{z} := z_i$ ,  $(x^*, y^*) := (x^i, y^i)$  and go to **Step 1**.

Otherwise:

### 5. Branch

From  $LP_i$ , construct  $k \geq 2$  linear programs  $LP_{i_1}, \dots, LP_{i_k}$  with smaller feasible regions whose union does not contain  $(x^i, y^i)$ , but contains all the solutions of  $LP_i$  with  $x \in \mathbb{Z}^n$ . Add the corresponding new nodes  $N_{i_1}, \dots, N_{i_k}$  to  $\mathcal{L}$  and go to **Step 1**.

# Choices in branch-and-bound

Each of the steps in a branch-and-bound algorithm can be done in many different ways

- ▶ Heuristics to find feasible solutions – yields lower bounds
- ▶ Solving a relaxation – yields upper bounds
- ▶ Node selection – which subproblem to look at next
- ▶ Branching – dividing the feasible region

You can “help” an integer programming solver by telling it how it should do these steps

- ▶ You can even implement your own **better** way to do one or more of these steps
- ▶ You can do better because you know more about your problem

## How long does branch-and-bound take?

Simplistic (but useful) approximation:

$$\text{Total time} = (\text{Time to process a node}) \times (\text{Number of nodes})$$

When making choices in branch-and-bound, think about effect on these separately

### Question

Which of these is likely to be *most important*?



## Choices in branch-and-bound: **Heuristics**

Practical perspective: finding good **feasible solutions** is most important

- ▶ Manager won't be happy if you tell her you have no solution, but you know the optimal solution is at most  $\bar{z}$

A **heuristic** is an algorithm that *tries* to find a good feasible solution

- ▶ No guarantees – maybe fails to find a solution, maybe finds a poor one
- ▶ But, typically runs fast
- ▶ Sometimes called “primal heuristics”

Good heuristics help find an *optimal solution* in branch-and-bound

- ▶ Key to success: **Prune early and often**
- ▶ We prune when  $z_i \leq \underline{z}$ , where  $\underline{z}$  is the best lower bound
- ▶ Good heuristics  $\Rightarrow$  larger  $\underline{z} \Rightarrow$  prune more

## Choices in branch-and-bound: **Choosing/solving the relaxation**

- ▶ The relaxation is the most important factor for **proving** a solution is optimal
- ▶ Optimal value of the relaxation yields the **upper bound**
  - ▶ Recall: we prune when  $z_i \leq \underline{z}$
  - ▶ Smaller (tighter) upper bounds  $\Rightarrow$  prune more
  - ▶ So the **formulation** is very important
  - ▶ Much of this course will be devoted to understanding good formulations and automatically improving formulations
- ▶ Time spent solving the relaxation at each node usually dominates the total solution time
  - ▶ Want to solve it fast, but also want to solve fewer
  - ▶ Potential trade-off: a formulation that yields a better upper bound may be larger (more time to solve relaxation)
    - ▶ Often, the formulation with a better upper bound wins

## Solving the LP relaxation efficiently

- ▶ Branching is usually done by changing bounds on a variable which is fractional in the current solution ( $x_j \leq 0$  or  $x_j \geq 1$ )
- ▶ Only difference in the LP relaxation in the new subproblem is this bound change
  - ▶ LP dual solution remains feasible
  - ▶ Reoptimize with dual simplex
  - ▶ If choose to process the new subproblem next, can even avoid refactoring the basis
- ▶ Another advantage of dual simplex: it works by improving an upper bound on optimal value of the relaxation
  - ▶ Let  $z_{ik}$  be the upper bound at iteration  $k$  of dual simplex:  
$$z_{ik} \geq z_i$$
  - ▶ If  $z_{ik} \leq \underline{z}$ , then  $z_i \leq z_{ik} \leq \underline{z}$ , so we can prune the node
  - ▶ We didn't even have to solve the LP relaxation completely!

## Choices in branch-and-bound: **Node selection**

Node selection: Strategy for selecting the next subproblem (node) to be processed.

- ▶ Important, but not as important as heuristics, relaxations, or branching (to be discussed next)
- ▶ Often called **search strategy**

Two **different** goals:

- ▶ Minimize overall solution time.
- ▶ Find a good feasible solution quickly.

# The Best First Approach

- ▶ One way to minimize overall solution time is to try to minimize the size of the search tree.
- ▶ We can achieve this if we choose the subproblem with the **best bound** (highest upper bound if we are maximizing).
- ▶ Let's prove this
  - ▶ A candidate node is said to be *critical* if its bound exceeds the value of an optimal solution to the IP.
  - ▶ Every critical node will be processed **no matter what the search order**
  - ▶ Best first is guaranteed to examine only critical nodes, thereby minimizing the size of the search tree (for a given fixed choice of branching decisions).

# Drawbacks of Best First

1. Doesn't necessarily find feasible solutions quickly
  - ▶ Feasible solutions are “more likely” to be found deep in the tree
2. Node setup costs are high
  - ▶ The linear program being solved may change quite a bit from one node LP solve to the next
3. Memory usage is high
  - ▶ It can require a lot of memory to store the candidate list, since the tree can grow “broad”

# The Depth First Approach

Depth first: always choose the deepest node to process next

- ▶ Dive until you prune, then back up and go the other way

Avoids most of the problems with best first:

- ▶ Number of candidate nodes is minimized (saving memory)
- ▶ Node set-up costs are minimized since LPs change very little from one iteration to the next
- ▶ Feasible solutions are usually found quickly

Unfortunately, if the initial lower bound is not very good, then we may end up processing lots of **non-critical nodes**.

- ▶ We want to avoid this extra expense if possible.

# Hybrid Strategies

## A Key Insight

If you *knew* the optimal solution value, the best thing to do would be to go depth first

- ▶ Idea: Go depth-first until  $z_i$  goes below optimal value  $z^*$ , then make a best-first move.
- ▶ But we don't know the optimal value!
  - ▶ Make an estimate  $z_E$  of the optimal solution value
  - ▶ Go depth-first until  $z_i \leq z_E$
  - ▶ Then jump to a better node



## Choices in branch-and-bound: **Branching**

- ▶ If our “relaxed” solution  $\hat{x}$  is not integer feasible, we must decide how to partition the search space into smaller subproblems
- ▶ The strategy for doing this is called a **Branching Rule**
- ▶ Branching wisely is very important
  - ▶ Significantly impacts bounds in subproblems
  - ▶ It is most important at the top of the branch-and-bound tree

# Branching in integer programming

Most common approach: Changing variable bounds

- ▶ If  $\hat{x}$  is not integer feasible, choose  
 $j \in N$  such that  $f_j := \hat{x}_j - \lfloor \hat{x}_j \rfloor > 0$
- ▶ Create two problems with additional constraints
  1.  $x_j \leq \lfloor \hat{x}_j \rfloor$  on one branch
  2.  $x_j \geq \lceil \hat{x}_j \rceil$  on other branch
- ▶ In the case of 0-1 IP, this dichotomy reduces to
  1.  $x_j = 0$  on one branch
  2.  $x_j = 1$  on other branch

Why is branching by changing variable bounds convenient when using LP relaxations?

## Key question

Which variable to branch on?

# The goal of branching

Branching divides one problem into two or more subproblems

- ▶ We would like to choose the branching that minimizes the sum of the solution times of all the created subproblems.
- ▶ This is the solution of the *entire subtree* rooted at the node.

How do we know how long it will take to solve each subproblem?

- ▶ **Answer:** We don't.
- ▶ **Idea:** Try to branch on variables that will cause the upper bounds to decrease the most
- ▶ This will lead to more pruning, and smaller subtrees

## Finding a good branching variable

I want to branch on a variable that causes the upper bound to decrease *a lot* in the subproblems!

- ▶ Then I can prune those nodes, or should be able to prune them quickly
- ▶ So a branching variable that changes these bounds “the most” is likely to be a good choice.

### Ideas?

What are some ideas **you** have for deciding on a branching variable?

# Predicting the bound change in a subproblem

How can I (quickly?) estimate the upper bounds that would result from branching on a variable?

- ▶ **Strong branching**

- ▶ Actually solve the LP relaxation of each subproblem for each potential branching variable

- ▶ **Pseudo-costs**

- ▶ Approximate the bound change based on previous information collected in the branch-and-bound tree

- ▶ Hybrid: “Reliability branching”

- ▶ **Tentative branching**

- ▶ Like strong branching, but also add valid inequalities to the subproblems, and possibly branch a few times

## Strong branching: Practicalities

Don't fully solve the subproblem LPs – just do a few dual simplex pivots

- ▶ This gives an upper bound on the subproblem bound
- ▶ How many is “a few”? — empirical study suggests 25 or so

Don't check subproblem for *every* candidate branching variable

- ▶ Which to evaluate?
- ▶ Look at an estimate of their effectiveness that is very cheap to evaluate
  - ▶ E.g., “most fractional” variables, or pseudocost
- ▶ Perhaps evaluate more candidates near the top of the tree

Fully solving the LPs or evaluating more candidates will probably reduce search tree size, but likely increases total time

# Combining multiple subproblem bounds

- ▶ For each candidate branching variable, we calculate an estimate of the upper bound change for each subproblem
  - ▶ Either via strong branching or pseudo-costs
- ▶ How do we combine the two numbers together to form one measure of goodness for choosing a branching variable?
- ▶ Idea: branch on variable  $x_{j^*}$  with:

$$j^* = \arg \max \{ D_j^+ D_j^- \}.$$

- ▶ Older alternative: A weighted sum of (min/max)...

# Putting it all together

Choices we've discussed in branching:

- ▶ Strong branching or pseudo-costs?
- ▶ Pseudo-costs
  - ▶ How should we initialize?
  - ▶ How should we update?
- ▶ Strong branching
  - ▶ How do we choose the list of branching candidates?
  - ▶ How many pivots to do on each?
- ▶ Once we have the bound estimates, how do we finally choose the branching variable?

Ultimately, we must use **empirical evidence** and intuition to answer these questions.



## 1.2.2 The Cutting Plane Method

# The Cutting Plane Method

Integer program (MILP):

$$\max \{cx + hy : (x, y) \in S\}$$

We **want**:

- ▶ Optimal solution  $(x^*, y^*)$ .
- ▶ Optimal value  $z^*$ .

Natural LP relaxation:

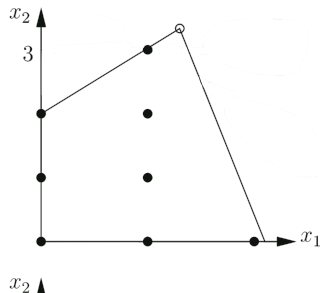
$$\max\{cx + hy : (x, y) \in P_0\}$$

We **have**:

- ▶ **Basic** optimal solution  $(x^0, y^0)$ .
- ▶ Optimal value  $z_0$ .

**Different idea** for dealing with the case  $(x^0, y^0) \notin S$ .

# The Cutting Plane Method



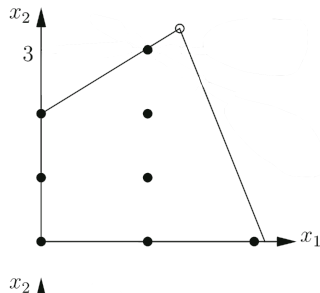
# The Cutting Plane Method

Find an inequality

$$\alpha x + \gamma y \leq \beta$$

that is:

- ▶ valid for  $S$ :  $\alpha x + \gamma y \leq \beta \quad \forall (x, y) \in S$ ,
- ▶ and violated by  $(x^0, y^0)$ :  $\alpha x^0 + \gamma y^0 > \beta$ .



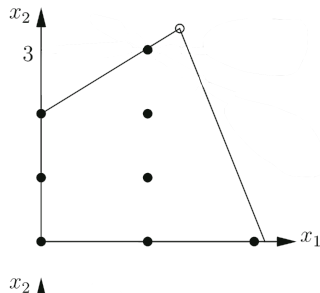
# The Cutting Plane Method

Find an inequality

$$\alpha x + \gamma y \leq \beta$$

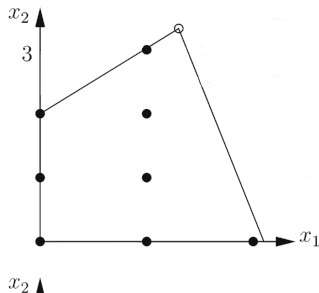
that is:

- ▶ valid for  $S$ :  $\alpha x + \gamma y \leq \beta \quad \forall (x, y) \in S$ ,
- ▶ and violated by  $(x^0, y^0)$ :  $\alpha x^0 + \gamma y^0 > \beta$ .



# The Cutting Plane Method

- ▶ We call such an inequality a valid inequality, or cutting plane separating  $(x^0, y^0)$  from  $S$ .
- ▶



# The Cutting Plane Method

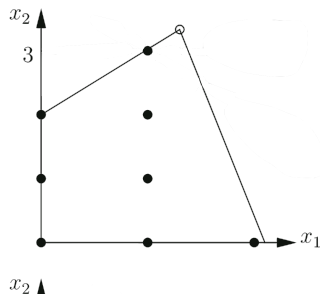
- Define

$$P_1 := P_0 \cap \{(x, y) : \alpha x + \gamma y \leq \beta\}.$$

- Since

$$S \subseteq P_1 \subset P_0,$$

the LP relaxation of MILP based on  $P_1$  is **stronger** than the natural LP relaxation.

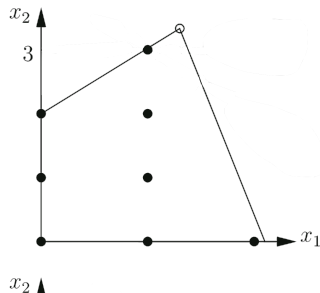


# The Cutting Plane Method

More precisely:

- ▶ The optimal solution  $(x^0, y^0)$  of the natural LP relaxation does not belong to  $P_1$ .
- ▶ The optimal value of the LP  $\max\{cx + hy : (x, y) \in P_1\}$  is **at least as good** an upper-bound on the value  $z^*$  as  $z_0$ :

$$\max\{cx + hy : (x, y) \in P_1\} \leq \max\{cx + hy : (x, y) \in P_0\}.$$



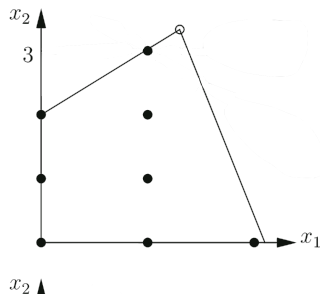


# The Cutting Plane Method

More precisely:

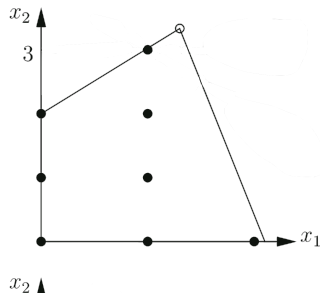
- ▶ The optimal solution  $(x^0, y^0)$  of the natural LP relaxation does not belong to  $P_1$ .
- ▶ The optimal value of the LP  $\max\{cx + hy : (x, y) \in P_1\}$  is **at least as good** an upper-bound on the value  $z^*$  as  $z_0$ :

$$\max\{cx + hy : (x, y) \in P_1\} \leq \max\{cx + hy : (x, y) \in P_0\}.$$



# The Cutting Plane Method

- The recursive application of this idea leads to the **cutting plane approach**.



# The Cutting Plane Method

## Cutting Plane Method

Starting with  $i = 0$ , repeat:

**Recursive Step.** Solve the LP  $\max\{cx + hy : (x, y) \in P_i\}$ .

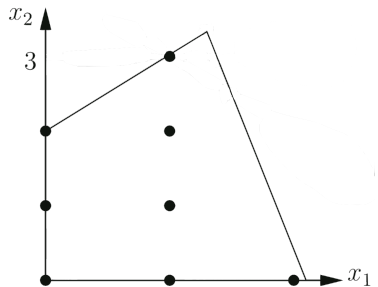
- ▶ If the associated optimal basic solution  $(x^i, y^i)$  belongs to  $S$ , stop.
- ▶ Otherwise solve the separation problem:

Find a cutting plane  $\alpha x + \gamma y \leq \beta$  separating  $(x^i, y^i)$  from  $S$ .

Set  $P_{i+1} := P_i \cap \{(x, y) : \alpha x + \gamma y \leq \beta\}$ .

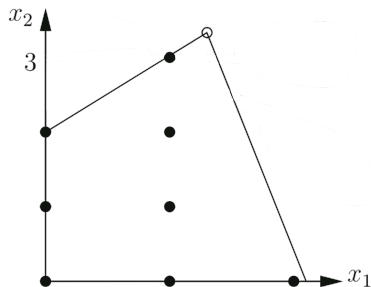
Repeat the recursive step.

Let's see an example



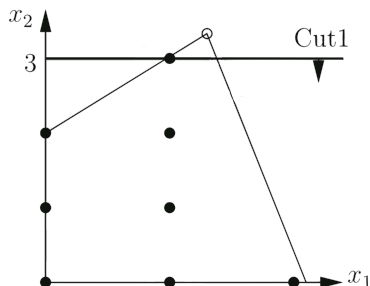
## Let's see an example

- The solution of the natural LP relaxation is  $(1.3, 3.3)$  with value 14.08.



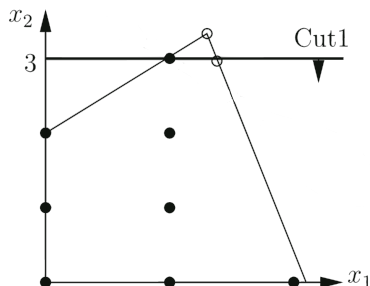
## Let's see an example

- ▶ The solution of the natural LP relaxation is  $(1.3, 3.3)$  with value 14.08.
- ▶ **Cut 1:**  $x_2 \leq 3$ .



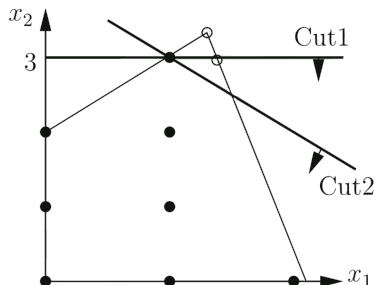
## Let's see an example

- ▶ The solution of the natural LP relaxation is  $(1.3, 3.3)$  with value 14.08.
- ▶ **Cut 1:**  $x_2 \leq 3$ .
- ▶ The solution of the LP relaxation is now  $(1.375, 3)$  with value 13.8625.



## Let's see an example

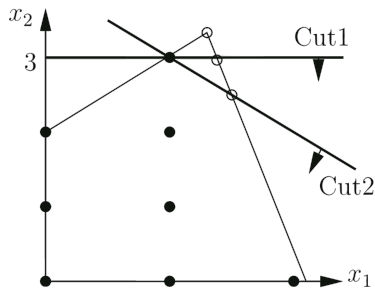
- ▶ The solution of the natural LP relaxation is  $(1.3, 3.3)$  with value 14.08.
- ▶ **Cut 1:**  $x_2 \leq 3$ .
- ▶ The solution of the LP relaxation is now  $(1.375, 3)$  with value 13.8625.
- ▶ **Cut 2:**  $x_1 + x_2 \leq 4$ .





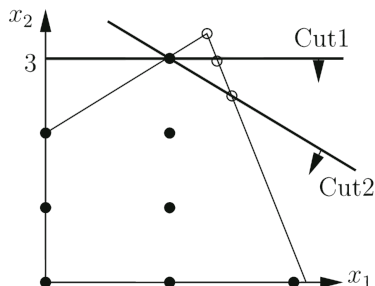
## Let's see an example

- ▶ The solution of the natural LP relaxation is  $(1.3, 3.3)$  with value 14.08.
- ▶ **Cut 1:**  $x_2 \leq 3$ .
- ▶ The solution of the LP relaxation is now  $(1.375, 3)$  with value 13.8625.
- ▶ **Cut 2:**  $x_1 + x_2 \leq 4$ .
- ▶ The solution of the LP relaxation is now  $(1.5, 2.5)$  with value 13.5.



## Let's see an example

- ▶ The solution of the natural LP relaxation is  $(1.3, 3.3)$  with value 14.08.
- ▶ **Cut 1:**  $x_2 \leq 3$ .
- ▶ The solution of the LP relaxation is now  $(1.375, 3)$  with value 13.8625.
- ▶ **Cut 2:**  $x_1 + x_2 \leq 4$ .
- ▶ The solution of the LP relaxation is now  $(1.5, 2.5)$  with value 13.5.
- ▶ ...



See the book to understand how these cuts can be generated.

# Choices in the Cutting Plane Method

- ▶ The **separation problem** is a central issue.
- ▶ If the basic solution  $(x^i, y^i)$  is not in  $S$ , there are **infinitely many** cutting planes separating  $(x^i, y^i)$  from  $S$ .
- ▶ How does one produce effective cuts?
- ▶ There is a **tradeoff** between the running time of a separation procedure and the quality of the cutting planes it produces.
- ▶ In practice, it may also be preferable to generate **several** cutting planes separating  $(x^i, y^i)$  from  $S$ , instead of a **single** cut, and to add them all to  $P_i$  to create  $P_{i+1}$ .

### 1.2.3 The Branch-and-Cut Method

# The Branch-and-Cut Method

- ▶ In the branch-and-bound approach, the **tightness of the upper bound** is crucial for pruning the enumeration tree (**Step 4**).
- ▶ Tighter upper bounds can be calculated by applying the **cutting plane approach** to the subproblems.
- ▶ This leads to the **branch-and-cut approach**.
- ▶ Currently the **most successful method** for solving integer programs.
- ▶ It is obtained by adding a cutting-plane step before the branching step in the branch-and-bound method.

# The Branch-and-Cut Method

## Branch-and-Cut Method

### 0. Initialize

$\mathcal{L} := \{N_0\}$ ,  $\underline{z} := -\infty$ ,  $(x^*, y^*) := \emptyset$ .

### 1. Terminate?

If  $\mathcal{L} = \emptyset$ , the solution  $(x^*, y^*)$  is optimal.

### 2. Select node

Choose a node  $N_i$  in  $\mathcal{L}$  and delete it from  $\mathcal{L}$ .

### 3. Bound

Solve  $LP_i$ . If it is infeasible, go to **Step 1**.

Else, let  $(x^i, y^i)$  be an optimal solution of  $LP_i$  and  $z_i$  its objective value.

### 4. Prune

If  $z_i \leq \underline{z}$ , go to **Step 1**.

If  $(x^i, y^i)$  is feasible to MILP, set  $\underline{z} := z_i$ ,  $(x^*, y^*) := (x^i, y^i)$  and go to **Step 1**.

Otherwise:

### 5. Add cuts?

Decide whether to strengthen the formulation  $LP_i$  or to branch. In the first case, strengthen  $LP_i$  by adding cutting planes and go back to **Step 3**. In the second case, go to **Step 6**.

### 6. Branch

[...]

# Choices in Branch-and-Cut

The decision of whether to add cuts in Step 5 is made empirically based on:

- ▶ The success of previously added cuts.
- ▶ Characteristics of the new cuts such as their density (the fraction of nonzero coefficients in the cut).
- ▶ Etc.

Typically:

- ▶ Several rounds of cuts are added at the root node  $N_0$ .
- ▶ Fewer or no cuts might be generated deeper in the enumeration tree.

## 1.3 Complexity



# Computational complexity

## Why do we care?

Computational complexity answers the question:

Which problems are **hard** and which problems are **easy**?

Your unreasonable boss (or advisor):

“Solve problem X to optimality – and fast!”

1. You're having no luck!
2. Idea: **prove** the problem is **hard**.

You can tell your boss that you need to give something up:

- ▶ Seek approximate solutions.
- ▶ Allow an algorithm which **sometimes** may take **very** long.

### 1.3.1 Problems, Instances, Encoding Size

# What is a problem? What is an instance?

## Problem

A problem is a question to be answered for any given set of data.

- ▶ Linear program

$$\max\{cx : Ax \leq b, x \geq 0\}.$$

- ▶ Mixed integer linear program.
- ▶ Assignment problem.
- ▶ Knapsack problem.
- ▶ Travelling salesman problem.

## Instance

An instance of a problem is a specific data set.

- ▶ An instance of a LP specifies  $A, b, c$ :

$$\begin{array}{ll}\max & 5.5x_1 + 2.1x_2 \\ \text{s. t.} & -x_1 + x_2 \leq 2 \\ & 8x_1 + 2x_2 \leq 17 \\ & x_1, x_2 \geq 0\end{array}$$

- ▶ An instance of a knapsack problem specifies  $b, n, c_i, a_i, \forall i = 1, \dots, n$ .

# What is the encoding size of an instance?

The encoding size of an instance is the **number of bits** required to write down the data of the instance.

We assume that all **integers** are written in binary encoding.

- ▶ An integer  $0 \leq n < 2^{t+1}$  can be represented by a **binary vector**  $(\delta_0, \delta_1, \dots, \delta_t)$ , where  $n = \sum_{i=0}^t \delta_i 2^i$ .
- ▶ The encoding size of an integer  $n$  is  $1 + \lceil \log_2(|n| + 1) \rceil$ .

A **rational** number can be written as  $\frac{p}{q}$ , where  $p$  is an integer and  $q$  is a positive integer.

- ▶ The encoding size of  $\frac{p}{q}$  is  $1 + \lceil \log_2(|p| + 1) \rceil + \lceil \log_2(|q| + 1) \rceil$ .

The encoding size of a **rational vector or matrix** is the sum of the encoding sizes of its entries.

# Polynomially bounded?

## Definition: Polynomially bounded

Function  $f: S \rightarrow \mathbb{R}_+$  is polynomially bounded by  $g: S \rightarrow \mathbb{R}_+$  if there exists a polynomial  $\phi: \mathbb{R} \rightarrow \mathbb{R}$  such that

$$f(s) \leq \phi(g(s)) \quad \text{for every } s \in S.$$

- **Example:**  $s^2$  is **polynomially bounded** by  $s$  since  $\phi(x) = x^2$  is a polynomial that satisfies the definition.
- of  $v$ .

# Big O Notation

- ▶ Convenient way to compare relative sizes of functions.
- ▶ Highlights large scale differences.
- ▶ Let  $f, g: S \rightarrow \mathbb{R}_+$ . Then  $f(x) = O(g(x))$  if there exists a positive **real number**  $M$  and an  $x_0 \in S$  such that

$$f(x) \leq M \cdot g(x) \quad \text{for every } x > x_0.$$

Examples:

- ▶ Is  $f(x) = 100x^2 + 3x = O(x^2)$ ?
- ▶ Is  $f(x) = 6x^3 = O(x^2)$ ?
- ▶ Is  $\log(x) = O(x)$ ?

How to work with big O:

- ▶ Any polynomial function is “big O” of the monomial that is its highest term.
- ▶ Polynomials are “big O” of exponentials.
- ▶ Logarithms are “big O” of polynomials.

### 1.3.2 Polynomial Algorithm

# What is an algorithm?

## Algorithm

An algorithm for solving a problem is a procedure that, given any possible instance, produces the correct answer in **finite time**.

Some algorithms for solving LP:

- ▶ Fourier-Motzkin elimination.
- ▶ Dantzig's **simplex method** with an anti-cycling rule.
- ▶ Khachiyan's **ellipsoid algorithm**.
- ▶ Karmarkar's **interior point algorithm**.
- ▶ Chubanov's **relaxation algorithm**.

**Question:** What about the branch-and-bound method for integer programming? And the cutting plane method?



# What is a polynomial-time algorithm?

## Polynomial-time algorithm

An algorithm is said to solve a problem in polynomial time if its **running time** is **polynomially bounded** by the **encoding size** of the input.

- ▶ The **running time** is measured as the number of arithmetic operations carried out by the algorithm.  
(We use the bit model, see page 362 in **Chapter 8** in **525 book**.)
- ▶ The **running time** is a function  $f$  defined on the set of instances.
- ▶ The **encoding size of the input** is a function  $g$  defined on the set of instances.

Polynomial algorithm  $\equiv$  polynomial-time algorithm.

# Polynomial algorithms for LP

Not all algorithms for LP are polynomial!

- ▶ ~~Fourier-Motzkin elimination.~~
- ▶ ~~Dantzig's simplex method with an anti-cycling rule.~~
- ▶ Khachiyan's ellipsoid algorithm.
- ▶ Karmarkar's interior point algorithm.
- ▶ Chubanov's relaxation algorithm.

## At last: The class of “easy” problems

### Definition: Polynomially solvable

A problem is polynomially solvable if there exists a polynomial algorithm to solve it.

Examples: ( $L$  denotes the encoding size of an instance.)

- ▶ **Linear programming**: running time  $O(L^4)$ .
- ▶ **Solving systems of equations**: running time  $O(L^3)$ .
- ▶ **Assignment problem**: running time  $O(L^3)$ .
- ▶ **Shortest path problem with nonnegative weights**: running time  $O(L^2)$ .

In particular each polynomially solvable problem has an **optimal solution** of encoding size polynomial in the encoding size of the input.

# Decision problems

## Definition: Decision problem

A decision problem is a problem whose answer is either “yes” or “no”.

**Example:** Integer factorization.

- ▶ Given positive integers  $m$  and  $n$ , does  $m$  have a factor less than  $n$  and greater than one?

**Example:** Integer programming feasibility.

- ▶ **Optimization problem:**  $\max\{cx : x \in S\}$ .
- ▶ **Decision problem:** Does there exist  $x \in S$  such that  $cx \geq k$ ?

**Example:** TSP feasibility.

- ▶ **Optimization problem:** What is the **shortest possible route** that visits each city and returns to the origin city?
- ▶ **Decision problem:** Does there exist a route that visits each city and returns to the origin city, **whose length is at most  $k$** ?

# The complexity class $\mathcal{P}$

## Definition: $\mathcal{P}$

The complexity class  $\mathcal{P}$  is the class of all **decision problems** for which there exists a **polynomial algorithm** to solve it.

Examples of problems in  $\mathcal{P}$ :

- ▶ Linear programming feasibility.
- ▶ Assignment feasibility.
- ▶ Shortest path with nonnegative weights feasibility.
- ▶ ...

### 1.3.3 Complexity Class $\mathcal{NP}$

# Towards a definition of “hard” problems: the class $\mathcal{NP}$

First step: Define a class of “fair game” problems –  $\mathcal{NP}$ .

- ▶  $\mathcal{NP} \neq$  “Non-polynomial”.
- ▶  $\mathcal{NP} \equiv$  “Nondeterministic polynomial-time” – we won’t use this definition.

## Definition: $\mathcal{NP}$

$\mathcal{NP}$  is the class of all decision problems for which the “yes”-answer has a **certificate** that can be checked in polynomial time.

- ▶ The certificate is a **proof** that the “yes”-answer is correct.
- ▶ Such a certificate must also be of polynomial encoding size.

## Some problems in $\mathcal{NP}$

**Example:** Integer factorization.

- ▶ Given positive integers  $m$  and  $n$ , does  $m$  have a factor less than  $n$  and greater than one?
- ▶ If the answer is “yes,” what certificate can you give that can be checked in polynomial time?



## Some problems in $\mathcal{NP}$

**Example:** 0, 1 LP feasibility.

- ▶ Given a 0, 1 linear set

$$S = \{x \in \{0, 1\}^n : Ax \leq b\},$$

is it nonempty?

- ▶ If  $S \neq \emptyset$ , then it contains a solution whose encoding size is polynomial in the encoding size of the input  $(A, b)$ .

## Some problems in $\mathcal{NP}$

Some more problems in  $\mathcal{NP}$ :

- ▶ Assignment feasibility.
- ▶ Knapsack feasibility.
- ▶ TSP feasibility.
- ▶ ...
- ▶ Every problem in  $\mathcal{P}$  is also in  $\mathcal{NP}$ . Why?

## Some problems in $\mathcal{NP}$

**Example:** Linear programming feasibility.

- ▶ Given a polyhedron

$$P = \{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\},$$

is it nonempty?

- ▶ If  $P \neq \emptyset$ , then it contains a **basic feasible solution**.
- ▶ A **basic feasible solution** has encoding size polynomial in the encoding size of the input  $(A, b)$ .

### **Proposition 1.2.**

Let  $A$  be a nonsingular  $n \times n$  rational matrix and  $b$  a rational  $n$ -vector. The encoding size of the unique solution of  $Ax = b$  is polynomially bounded by the encoding size of  $(A, b)$ .

## Some problems in $\mathcal{NP}$

**Example:** Integer programming feasibility.

- ▶ Given a mixed integer linear set

$$S = \{(x, y) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p : Ax + Gy \leq b\},$$

is it nonempty?

- ▶ If  $S \neq \emptyset$ , then it contains a solution whose encoding size is polynomial in the encoding size of the input  $(A, G, b)$ .
- ▶ This is not obvious... (for unbounded sets).

## Bring on the “hard” problems

How we would like to define a hard problem:

A problem is hard if there does not exist **any** polynomial time algorithm to solve it.

- ▶ Unfortunately, **no one** has been able to prove that a problem in  $\mathcal{NP}$  is hard by this definition.
- ▶ So, what can you tell your boss?

# Bring on the “hard” problems

Alternate (informal) definition of a hard problem:

A problem is hard if being able to solve it efficiently implies that I can efficiently solve **every** problem in  $\mathcal{NP}$ .

- ▶ So you can tell your boss:  
“I can’t solve it, but neither could Gomory, Edmonds, Lovasz, ...”
- ▶ Amazingly, such problems exist!

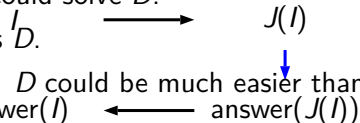
# Polynomial reductions

## Definition: Polynomially reducible

A problem  $D$  is polynomially reducible to a problem  $Q$  if there exist two polynomial algorithms such that:

- ▶ The first produces an instance  $J(I)$  of  $Q$  for any given instance  $I$  of  $D$ .
- ▶ The second produces the correct answer for  $I$  given a correct answer for  $J(I)$ .

If  $D$  is <sup>Problem</sup> polynomially reducible to  $Q$ , then:

- ▶ If I could solve  $Q$ , then I could solve  $D$ .
- ▶ So  $Q$  is at least as hard as  $D$ .  


```
graph LR; I["Instance I of D"] -- "polynomial algorithm" --> JI["J(I)"]; JI -- "polynomial algorithm" --> AI["answer(I)"]; Q["Problem Q"]
```
- ▶ The reverse may not hold!  $D$  could be much easier than  $Q$ .

## Formal definition of a hard problem: $\mathcal{NP}$ -complete

### Definition: $\mathcal{NP}$ -complete

A problem  $Q$  in  $\mathcal{NP}$  is  $\mathcal{NP}$ -complete if **all** other problems  $D$  in  $\mathcal{NP}$  are polynomially reducible to  $Q$ .

- ▶ Up to a polynomial factor, solving  $Q$  requires at least as much computing time as solving any problem in  $\mathcal{NP}$ .
- ▶ Every  $\mathcal{NP}$ -complete problem is in  $\mathcal{NP}$  by definition.



# The first $\mathcal{NP}$ -complete problem

The Satisfiability problem (SAT) is described by:

- ▶ A finite set  $N = \{1, \dots, n\}$  (the literals). and
- ▶  $m$  **pairs** of subsets of  $N$ ,  $(C_i, D_i), i = 1, \dots, m$  (the clauses).

Question: Does there exist  $x \in \{0, 1\}^n$  which satisfies

$$\sum_{j \in C_i} x_j + \sum_{j \in D_i} (1 - x_j) \geq 1 \quad \forall i = 1, \dots, m ?$$

- ▶ This problem is in  $\mathcal{NP}$ . **Why?**

**Theorem (Cook, 1971)**

SAT is  $\mathcal{NP}$ -complete.

- ▶ This was the first problem shown to be  $\mathcal{NP}$ -complete.
- ▶ We will not venture to understand the proof.

# How to prove $\mathcal{NP}$ -completeness

How to prove that a problem  $Q$  in  $\mathcal{NP}$  is  $\mathcal{NP}$ -complete:

- ▶ If a **special case** of  $Q$  is  $\mathcal{NP}$ -complete, then  $Q$  is  $\mathcal{NP}$ -complete.
- ▶ If **one**  $\mathcal{NP}$ -complete problem  $D$  is **polynomially reducible** to  $Q$ , then  $Q$  is  $\mathcal{NP}$ -complete.

Common mistake!!

- ▶ For a problem  $Q$  that you want to show is  $\mathcal{NP}$ -complete, you must show how to reduce an instance of an  $\mathcal{NP}$ -complete problem to an instance of  $Q$  (common mistake is to show vice versa!)

## Some $\mathcal{NP}$ -complete problems

- ▶ SAT.
- ▶ 0,1 LP feasibility. Why?
- ▶ Integer programming feasibility. Why?
- ▶ Knapsack feasibility.
- ▶ TSP feasibility.
- ▶ ...

## Formal definition of a hard problem: $\mathcal{NP}$ -hard

### Definition: $\mathcal{NP}$ -hard

A problem  $Q$  is said to be  $\mathcal{NP}$ -hard if all problems  $D$  in  $\mathcal{NP}$  are **polynomially reducible** to  $Q$ .

- $Q$  is not necessarily in  $\mathcal{NP}$  or a decision problem.

## Some $\mathcal{NP}$ -hard problems

- ▶ 0,1 LP.
- ▶ Integer programming.
- ▶ Knapsack problem.
- ▶ TSP.
- ▶ ...

# Who wants to be a millionaire?!

The million dollar question:

Does  $\mathcal{P} = \mathcal{NP}$ ? I.e. could **every** problem in  $\mathcal{NP}$  be solvable in polynomial time?

- One of the Millennium Problems posed by the **Clay Mathematics Institute**: <http://www.claymath.org/millennium-problems/p-vs-np-problem>



## The line between $\mathcal{P}$ and $\mathcal{NP}$ -complete is very thin

- ▶ Consider a  $0, 1$  matrix  $A$  and an integer  $k$  defining the decision problem

$$\exists x \in \{0, 1\}^n \text{ that satisfies } Ax \leq 1, \sum_{i=1}^n x_i \geq k ?$$

- ▶ If each column of  $A$  has at most **2 nonzero entries**, then this problem is in  $\mathcal{P}$ .
- ▶ If columns of  $A$  can have **3 nonzero entries**, then this problem is  $\mathcal{NP}$ -complete.
- ▶ Shortest Path (with non-negative edge weights) is in  $\mathcal{P}$ .
- ▶ Longest Path (with non-negative edge weights) is  $\mathcal{NP}$ -complete.
- ▶ Longest Path on an acyclic graph is in  $\mathcal{P}$ .

# Theory versus practice

In practice, most problems known to be in  $\mathcal{P}$  are “easy” to solve.

- ▶ Most known polynomial time algorithms are of relatively low order: e.g.,  $O(n^2)$  and not  $O(n^{100})$ .

Although all  $\mathcal{NP}$ -complete problems are “equivalent” in theory, they are not in practice.

- ▶ TSP – Solved instances of size  $\approx 80000$ .
- ▶ Quadratic assignment problem (QAP) – Solved instances of size  $\approx 30$ .



# What to do?

You now know whether your problem is easy or hard ...

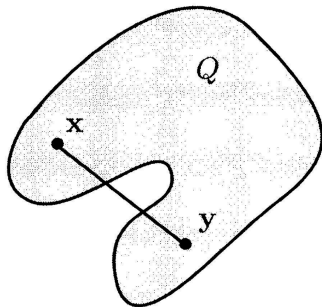
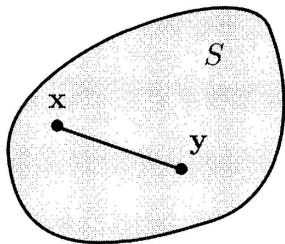
- ▶ If it is easy:
  - ▶ Find a **better** polynomial-time algorithm.
  - ▶ E.g., a combinatorial algorithm instead of a LP.
- ▶ If it is hard: give up???
  - ▶ No, but be realistic.
  - ▶ Approximation is still possible (sometimes).
  - ▶ We may be able to solve **large enough** instances.
  - ▶ **Look for special structure!!**

## 1.4 Convex Hulls and Perfect Formulations

# Convex set

## Definition: Convex set

A set  $S \subseteq \mathbb{R}^n$  is convex if, for any two distinct points in  $S$ , the line segment joining them is also in  $S$ ,  
i.e., if  $x, y \in S$  then  $\lambda x + (1 - \lambda)y \in S$  for all  $0 \leq \lambda \leq 1$ .

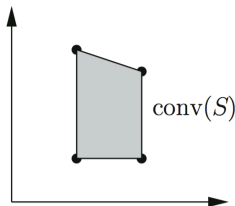
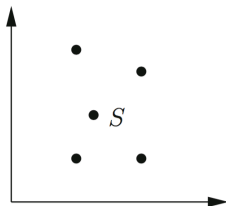


# Convex hull

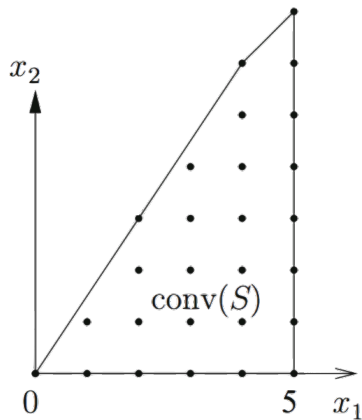
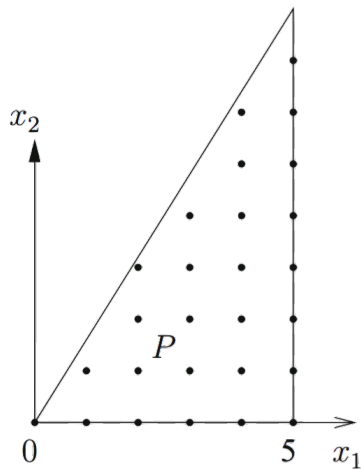
- ▶ Consider a set  $S \subseteq \mathbb{R}^n$ .
- ▶ The intersection of all convex sets containing  $S$  is itself a convex set containing  $S$ .
- ▶ It is therefore the **minimal (inclusionwise) convex set containing  $S$** .

## Definition: Convex hull

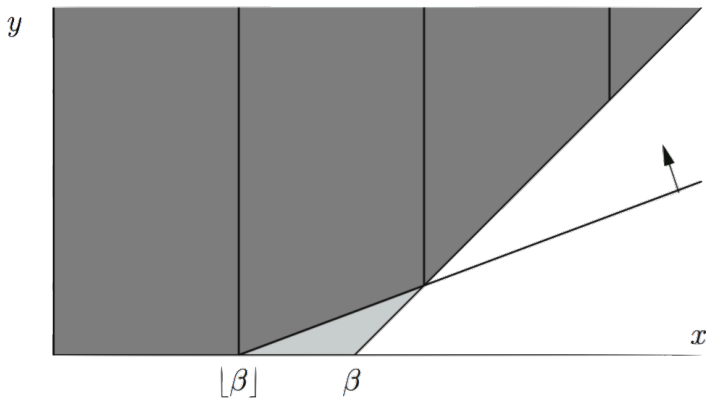
The convex hull of  $S \subseteq \mathbb{R}^n$  is the minimal convex set containing  $S$ , and it is denoted by  $\text{conv}(S)$ .



## Example: Convex hull of a pure integer linear set



## Example: Convex hull of a mixed integer linear set



# A characterization of $\text{conv}(S)$

## Definition: Convex combination

A point  $x \in \mathbb{R}^n$  is a convex combination of points in  $S$  if there exists a **finite** set of points  $x^1, \dots, x^p \in S$  and scalars  $\lambda_1, \dots, \lambda_p$  such that

$$x = \sum_{j=1}^p \lambda_j x^j, \quad \sum_{j=1}^p \lambda_j = 1, \quad \lambda_1, \dots, \lambda_p \geq 0.$$

**Exercise:** Prove that

$$\text{conv}(S) = \{x \in \mathbb{R}^n : x \text{ is a convex combination of points in } S\}.$$

## How to use $\text{conv}(S)$

Optimizing a linear function  $cx$  over  $S \subseteq \mathbb{R}^n$  is equivalent to optimizing  $cx$  over  $\text{conv}(S)$ .

**Lemma 1.3 (modified, assuming maximum exists)**

Let  $S \subset \mathbb{R}^n$  and  $c \in \mathbb{R}^n$ .

$$\max\{cx : x \in S\} = \max\{cx : x \in \text{conv}(S)\}.$$

Let's prove Lemma 1.3!



# Meyer's Theorem: Fundamental Theorem of IP

- Assume  $S$  is a mixed integer linear set

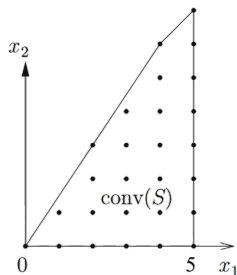
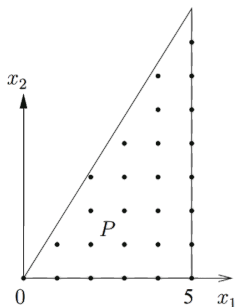
$$S := \{(x, y) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p : Ax + Gy \leq b\}$$

where  $A, G, b$  have **rational** entries.

- We will see in **Chapter 4** that in this case

$$\text{conv}(S) = \{(x, y) \in \mathbb{R}^n \times \mathbb{R}^p : A'x + G'y \leq b'\}$$

where  $A', G', b'$  have **rational** entries.



# From integer programming to LP

- ▶ For any linear objective function, we have

$$\max\{cx + hy : (x, y) \in S\} \stackrel{\text{Lemma 1.3}}{=} \max\{cx + hy : (x, y) \in \text{conv}(S)\}$$

- ▶ In order to solve the **integer program**

$$\max\{cx + hy : Ax + Gy \leq b, x \geq 0 \text{ integral}, y \geq 0\}$$

it is sufficient to solve the **LP**

$$\max\{cx + hy : A'x + G'y \leq b'\}.$$

- ▶ A central question is the **constructive** aspect of this LP:  
Given  $A, G, b$ , how does one compute  $A', G', b'$ ?

# From integer programming to LP

- ▶ The system  $A'x + G'y \leq b'$  also provides a **formulation** for the mixed integer set

$$\begin{aligned} S &:= \{(x, y) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p : Ax + Gy \leq b\} \\ &= \{(x, y) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p : A'x + G'y \leq b'\}. \end{aligned}$$

- ▶ This new formulation has the property that **for every objective function**, the **integer program** can be solved as a **LP**, disregarding the integrality requirement on the vector  $x$ .
- ▶ We call such a formulation a perfect formulation.
- ▶ When there are no continuous variables  $y$ , the set

$$\{x \in \mathbb{R}^n : A'x \leq b'\}$$

defined by a perfect formulation  $A'x \leq b'$  is called an integral polyhedron.