Optimization

A gentle introduction

Vittorio Maniezzo
Univ. Bologna

1

Integer programming

2

Integer linear Programming

Given rational numbers a_{ij}, b_i, c_j, find integers x_j that solve:

$$
\begin{array}{llcll}
\min & \sum_{j=1}^{n} c_j x_j & & & \\
\text{s. t.} & \sum_{j=1}^{n} a_{ij} x_j & \geq & b_i & 1 \leq i \leq m \\
& x_j & \geq & 0 & 1 \leq j \leq n \\
& x_j & & \text{integral} & 1 \leq j \leq n
\end{array}
$$

INTEGER-PROGRAMMING is NP-hard.

Integer linear programs

Integer linear programs are almost identical to linear programs with the very important exception that some of the decision variables need to have only integer values.

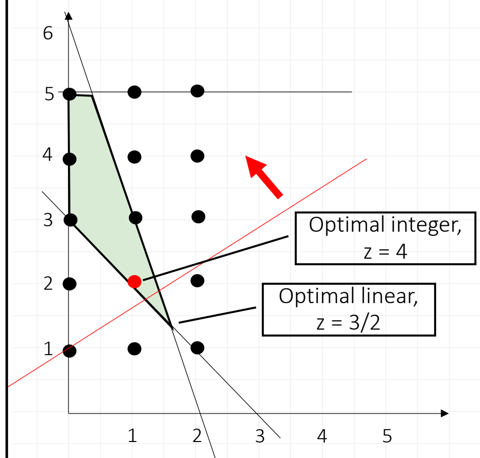
Since most integer programs contain a mix of real and integer variables they are often known as Mixed Integer Programs (MIP).

While the change from linear programming is a minor one, the effect on the solution process is enormous. Integer programs can be very difficult problems to solve, just rounding solutions is not an option.

Rounding variables

Taking a rational value to its nearest integer might work.

$$\begin{aligned} \min\ z = &-2x_1 + 3x_2 \\ s.t.\quad & x_1 + x_2 \geq 3 \\ & 3x_1 + x_2 \leq 6 \\ & x_2 \leq 5 \\ & x_1, x_2 \geq 0 \text{ and integer} \end{aligned}$$



Linear sol: $z = 3/2, x_1 = 3/2, x_2 = 3/2$
Integer sol: $z = 4, x_1 = 1, x_2 = 2$

In this case, rounding works

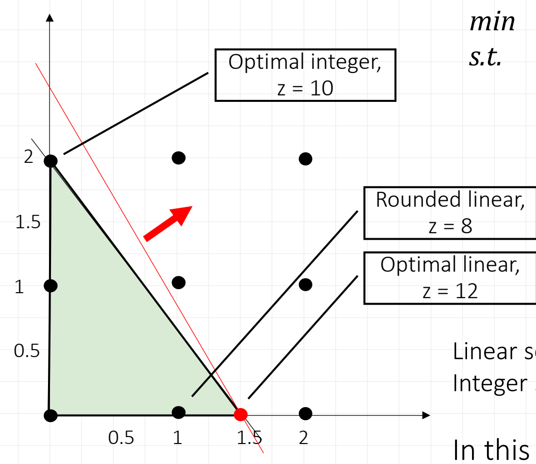
Vittorio Maniezzo - University of Bologna 5

5

Rounding variables

Taking a rational value to its nearest integer might be totally wrong.

$$\begin{aligned} \min\ z = &\ 8x_1 + 5x_2 \\ s.t.\quad & 4x_1 + 3x_2 \leq 6 \\ & x_1, x_2 \geq 0 \text{ and integer} \end{aligned}$$



Linear sol: $z = 12, x_1 = 1/2, x_2 = 0$
Integer sol: $z = 10, x_1 = 0, x_2 = 2$

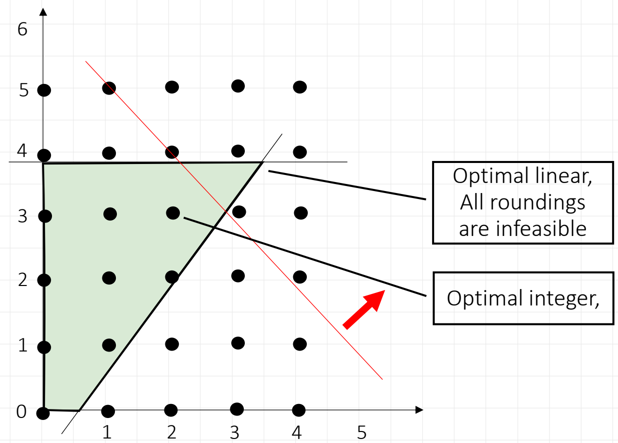
In this case, rounding does not work

Vittorio Maniezzo - University of Bologna 6

6

Rounding variables

In this case, rounding is infeasible.



7

Dealing with IP

There are three main categories of algorithms for integer programming problems:

- Exact algorithms: they *guarantee to find an optimal solution*, but may take an exponential time. They include branch-and-bound, cutting-planes and dynamic programming.
- Approximation algorithms: they provide in polynomial time a suboptimal solution together with *a bound on the quality* of the solution proposed.
- Heuristic algorithms: they provide a suboptimal solution, with *no guarantee on its quality*. Even the running time is not always guaranteed to be polynomial, but empirical evidence suggests that these algorithms find a good solution fast.

8

Branch and bound

It is a *divide et impera* (divide and conquer) method.

The problem to solve is decomposed into a number of simpler subproblems.

Decomposition proceeds recursively until simple sub-sub-problems can be solved.

The overall solution is derived from the solutions of the subproblems.

The decomposition of the problem into subproblems is the *branching* phase. It can be done in different ways, we will see only the one based on LP.

Branch and bound

From LP: the branching part eliminates non-integer values for integer variables in the following way:

- Initially, all variables are left as real variables. The problem is solved using linear programming;
- If one of the integer variables in the linear programming solution has a fractional value, e.g., $x_1 = 0.5$ then the linear program is split in two and the fractional region eliminated. This is done by branching on the variable value, e.g., adding the constraint $x_1 <= 0$ to form one linear program and $x_1 >= 1$ to form the other.
- By finding the optimal solution in each of these new linear programs and comparing them, we can find the optimal solution for the original problem.
- If either of the new linear programs has a fractional value for an integer variable then a new branch is needed.

Branching

Example: IP^0

$max\ z = 13x_1 + 8x_2$

s.t. $x_1 + 2x_2 \le 10$

$5x_1 + 2x_2 \le 20$

$x_1, x_2 \ge 0$ and integer

LP solution: $x_1^0 = 2.5, x_2^0 = 2.5, z^0 = 59.5$

Branching on x_1:

$max\ z = 13x_1 + 8x_2$

s.t. $x_1 + 2x_2 \le 10$

$5x_1 + 2x_2 \le 20$

$x_1 \le 2$

$x_1, x_2 \ge 0$ and integer

IP^1

$max\ z = 13x_1 + 8x_2$

s.t. $x_1 + 2x_2 \le 10$

$5x_1 + 2x_2 \le 20$

$x_1 \ge 3$

$x_1, x_2 \ge 0$ and integer

IP^2

Branching

Solving IP^1: $x_1^1 = 2, x_2^1 = 4, z^1 = 58$

Solution is feasible, branch concluded.

Solving IP^2: $x_1^2 = 3, x_2^2 = 2.5, z^2 = 59$

Branching on x_2:

$max\ z = 13x_1 + 8x_2$

s.t. $x_1 + 2x_2 \le 10$

$5x_1 + 2x_2 \le 20$

$x_1 \ge 3$

$x_2 \le 2$

$x_1, x_2 \ge 0$ and integer

IP^3

$max\ z = 13x_1 + 8x_2$

s.t. $x_1 + 2x_2 \le 10$

$5x_1 + 2x_2 \le 20$

$x_1 \ge 3$

$x_2 \ge 3$

$x_1, x_2 \ge 0$ and integer

IP^4

Branching

Solving IP^3: $x_1^3 = 3.2, x_2^3 = 2, z^3 = 57.6$

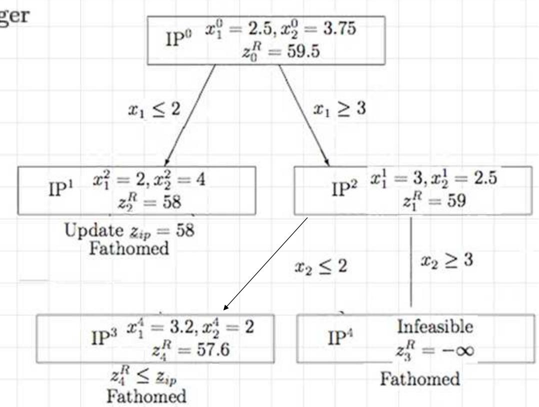
Solution is fractional, but we already know a better feasible solution, branch concluded.

Solving IP^4: subproblem is infeasible , branch concluded.

Search completed, solution was found exploring IP^2

Branching tree

$$
\begin{array}{llcll}
\text{maximize} & 13x_1 & + & 8x_2 & \\
\text{subject to} & x_1 & + & 2x_2 \le 10 & \\
 & 5x_1 & + & 2x_2 \le 20 & (\mathrm{IP}^0) \\
 & x_1 \ge 0, & x_2 \ge 0 & & \\
 & x_1, & x_2 \text{ integer} & &
\end{array}
$$



Branching

This branching process results in the formation of a branch-and-bound tree.

Each node in the tree represents a linear program consisting of the original linear program and additional added constraints.

Adding constraints to a mathematical programming will result in a deterioration of the objective value, descendent nodes have worse objective values.

The leaf nodes correspond to problems either infeasible or where all the integer variables have integer values, no further branching is needed.

All these values can be compared and the best one is the solution to the original integer program.

Note: for MIPs of any reasonable the size this tree could be huge, it grows exponentially with the number of integer variables.

The bounding process allows sections of the branch-and-bound tree to be removed from consideration before all the leaf nodes have integer solutions.

Branching

Adding the branching constraints to the linear programs at the nodes implies that the resulting nodes will have an optimal objective function value that is equal to or worse than the optimal objective function value of the original linear program.

The objective function values get worse the deeper we get into the tree.

Since we are finding the integer solution in the branch-and-bound tree with the best objective value, we can use any integer solution to bound the tree.

The current best integer solution is called the incumbent.

Branching

After solving a linear program at a node of the branch-and-bound tree one of the following conditions holds:

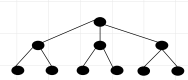
- The linear program is infeasible (no more branching is possible);
- The linear program gives an integer solution with a better objective value than the incumbent. The incumbent is replaced with this new solution;
- The linear program solution has a worse objective than the incumbent. Any nodes created from this node will also have a worse objective than the incumbent.
- The linear program solution is fractional and has a better objective value than the incumbent. Further branching from this node is necessary to ensure an optimal solution is found.

Only the last condition requires more branching, all the other conditions result in the node becoming fathomed.

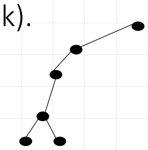
Branching strategy

As any tree, the search tree can be explored breadth-first, depth-first or with combinations thereof.

Breadth-first: you generate all sons of the incumbent node before changing the incumbent. Possibly get the best solution sooner, but large memory consumption.



Depth-first: you move the incumbent to the first generated son and backtrack when no further generation is possible. Can be slower to optimality but less memory consumption (just a stack).



Bounding

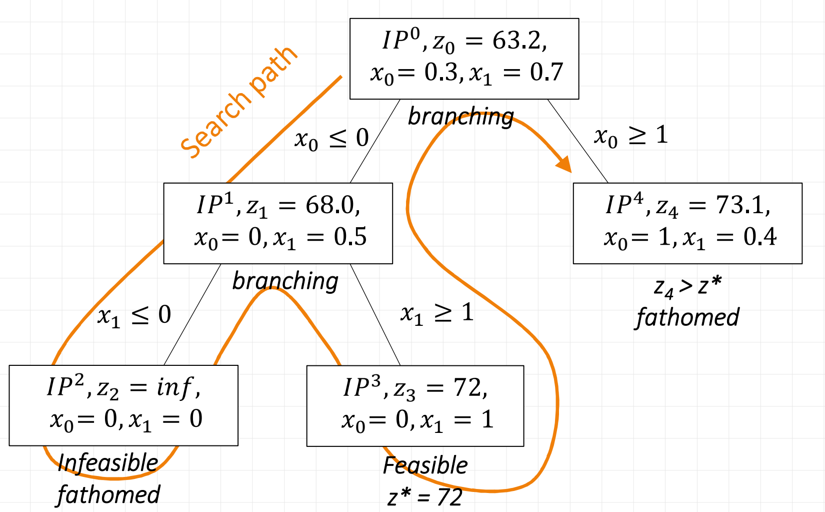
The Linear Programming (LP) relaxation has the same form as the integer program, except that integrality constrains are “relaxed” to allow variables to take fractional values.

The integer program's feasible region lies within the feasible region of the LP relaxation (at points where the integer variables have integer values). Therefore the integer restrictions cause the optimal objective function value to be worse in the integer program when compared to the LP relaxation.

If a solution $\mathbf{x}$ of the LP relaxation has integer values for the integer variables, then $\mathbf{x}$ is also a solution for the integer program.

Bounding, search tree

Depth first exploration, min problem



Branch and bound, pseudocode

```
LB = -Inf
UB = Inf
store root node in unexpanded node list
while unexpanded node list is not empty do
    Extract a node from the unexpanded node list {Node selection}
    Solve subproblem
    if infeasible then node is fathomed
    else
        if integer solution then
            if obj > LB then {Better integer solution found}
            LB:=obj
            Remove nodes j from list with UBj < LB
        else
            Find variable xk with fractional value v {Variable selection}
            Create node jnew with bound xk ≤ ⌊v⌋
            Ubjnew = obj
            Store node jnew in unexpanded node list
            Create node jnew with bound xk ≥ ⌈v⌉
            UB_jnew = obj
            Store node jnew in unexpanded node list
        end if
    end if
    UB = maxj UBj
end while
```

Can be a project

Branch and cut

For branch and cut, the lower bound is again provided by the linear relaxation of the integer program.

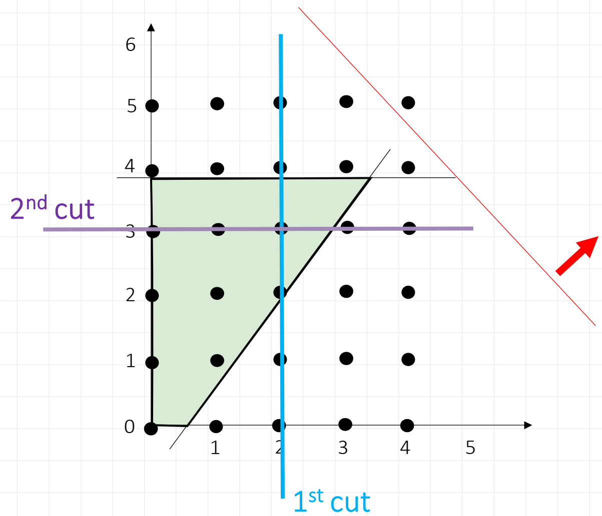
If the optimal solution to the problem is not integral, this algorithm searches for a constraint which is violated by this solution, but is not violated by any optimal integer solutions. This constraint is called a cutting plane. Well-known cuts are the Gomory cuts (which act on fractional parts of a variable).

When this constraint is added to the model, the old optimal solution is no longer valid, and so the new optimal will be different, potentially providing a tighter bound.

Cutting planes are iteratively derived until either an integral solution is found or it becomes impossible or too expensive to find another cutting plane.

In the latter case, a branch operation is performed and the search for cutting planes continues on the subproblems

Branch and cut



23

Dynamic programming

Dynamic programming is based on Bellman's Optimality Principle: "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must [illegible]nstitute an optimal policy."

1. Obtaining feasible solutions comes af[illegible]ce of decisions occurring in stages, leading to sta[illegible]otal cost is the sum of the costs of the individual[illegible]
2. A state can be seen a[illegible]y of all relevant past decisions.
3. Need to dete[illegible]n state transitions are possible. Let the cost of e[illegible]ansition be the cost of the corresponding decis[illegible]
4. Given t[illegible]e objective function, one must derive a recursion on the optimal cost from the origin state to a final, optimal state

Very interesting, but I cannot fit this in this course

24

Heuristics

A heuristic algorithm is an algorithm that finds a hopefully good feasible solution. Typically, heuristics are fast (polynomial) algorithms.

For some problems, even feasibility is NP-complete, in such cases a heuristic cannot even guarantee to find a feasible solution.

Heuristic algorithms can be classified among:

- Constructive: start from an empty solution and iteratively add new elements to the incumbent partial solution, until a complete solution is found. Usually greedy.
- Local Search: start from an initial feasible solution and iteratively try to improve it by "slightly" modifying it. Stop when no improving adjustment is possible (local optimum)
- Metaheuristics: try avoid local optima using specific techniques. Many approaches tested, in the 80's it was fashionable to draw inspiration from nature (annealing, genetics, ant colonies, …)

Example: K-means

k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster.

k-means minimizes within-cluster variances

$$\arg\min_S \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2$$

The Euclidean norm needs to be computable. Related combinatorial optimization problem is the p-median problem.

K-means

The k-means problem is a mixed nonlinear integer problem.

- All points need to be assigned to a centroid, integer variable.
- All points can have continuous coordinates, continuous variable.
- Objective function is the Euclidean norm, a quadratic function.

In this course, we will study a related linear problem.

PROJECT: optimize k-means other than with the heuristic hereafter

K-means heuristic

A pseudocode for a simple k-means clustering heuristic is as follow:

Step 1. Select k centroids randomly as initial centroids of k clusters (Initial solution).

Step 2. Allocate each customer to the nearest centroid, such that k new clusters are created.

Step 3. For each k new created clusters, recalculate new centroids.

Step 4. Repeat the steps 2 and 3, until new centroids for all clusters in each repetition are fixed.

K-means heuristic

```
# k-means heuristic
for i in range(max_iter):

    # distances between datapoints and centroids
    distances = np.array( [np.linalg.norm(data - c, axis=1) for c in centroids] )

    # new_labels, find centroid with minimal total distance
    new_labels = np.argmin(distances, axis=0)

    if (labels == new_labels).all():
        # labels unchanged
        labels = new_labels
        print('Labels unchanged. Terminating.')
        break
    else:
        # labels changed
        # difference : percentage of changed labels
        difference = np.mean(labels != new_labels)
        print('Iter {0}, {1}% labels changed'.format(i,(difference * 100)))
        labels = new_labels
        for c in range(k):
            # centroids are the means of their associated data points
            centroids[c] = np.mean(data[labels == c], axis=0)

return labels,centroids
```


p-median

Choose p vertices as median of their cluster. No new point is created, no need for Euclidean norms, loss function can be linear.

$$x_{ij} = \begin{cases} 1 & \text{if vertex } x_j \text{ is allocated to vertex } x_i \\ 0 & \text{otherwise} \end{cases}$$

The formulation becomes

$$\begin{array}{llll} min\ z = & \sum_{ij} d_{ij} x_{ij} & & (1) \\ s.t. & \sum_i x_{ij} = 1 & j = 1, \dots, n & (2) \\ & \sum_i x_{ii} = p & & (3) \\ & x_{ij} \leq x_{ii} & i,j = 1, \dots, n & (4) \\ & x_{ij} \in \{0,1\} & i,j = 1, \dots, n & (5) \end{array}$$

p-median in PULP

```
from pulp import *
import numpy as np
import pandas as pd

df = pd.read_csv("data/pmedval.csv", header=None)
c = df.values
p = 3  # number of clusters
n = len(c)
# decision variables
categ = 'Binary'; # 'Continuous''
X = LpVariable.dicts('X_%s_%s', (range(n), range(n)), cat = categ, lowBound = 0, upBound = 1)
# create the LP object, set up as a min problem
prob = LpProblem('PMedian', LpMinimize)
# cost function
prob += sum( sum(c[i][j] * X[i][j] for j in range(n)) for i in range(n))
# p clusters constraint  (x_ii = 1 is chosen as median)
prob += (sum(X[i][i] for i in range(n)) == p , "p medians")
# assignment constraint
for j in range(n):
     prob += (sum(X[i][j] for i in range(n)) == 1, "Assignment %d" % j)
for i in range(n):
     for j in range(n):
          prob += (X[i][j] - X[i][i] <= 0, "c{0}-{1}".format(i,j))
# save the model in a lp file
prob.writeLP("p-median.lp")
# view the model
print(prob)
# solve the model
prob.solve()
print("Status:",LpStatus[prob.status])
print("Objective: ",value(prob.objective))
for v in prob.variables():
     if(v.varValue > 0):
          print (v.name , "=", v.varValue)
```


p-median heuristics

Simple heuristics.

1) Simple greedy. Just as the one described for k-means

2) Lagrangian. Can relax either constraints (2) or constraint (3).

Either can be a project.

Insatnces can be retrieved from http://people.brunel.ac.uk/~mastjjb/jeb/orlib/pmedinfo.html