Exact Methods of Optimization

Nuno Antunes Ribeiro
Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN
Engineering Systems and Design

Optimization Problem

- **Optimize:** $\max(f(x_1, x_2) = 5x_1 + 8x_2)$

- **Constraints:** $x_1 + x_2 \leq 6$

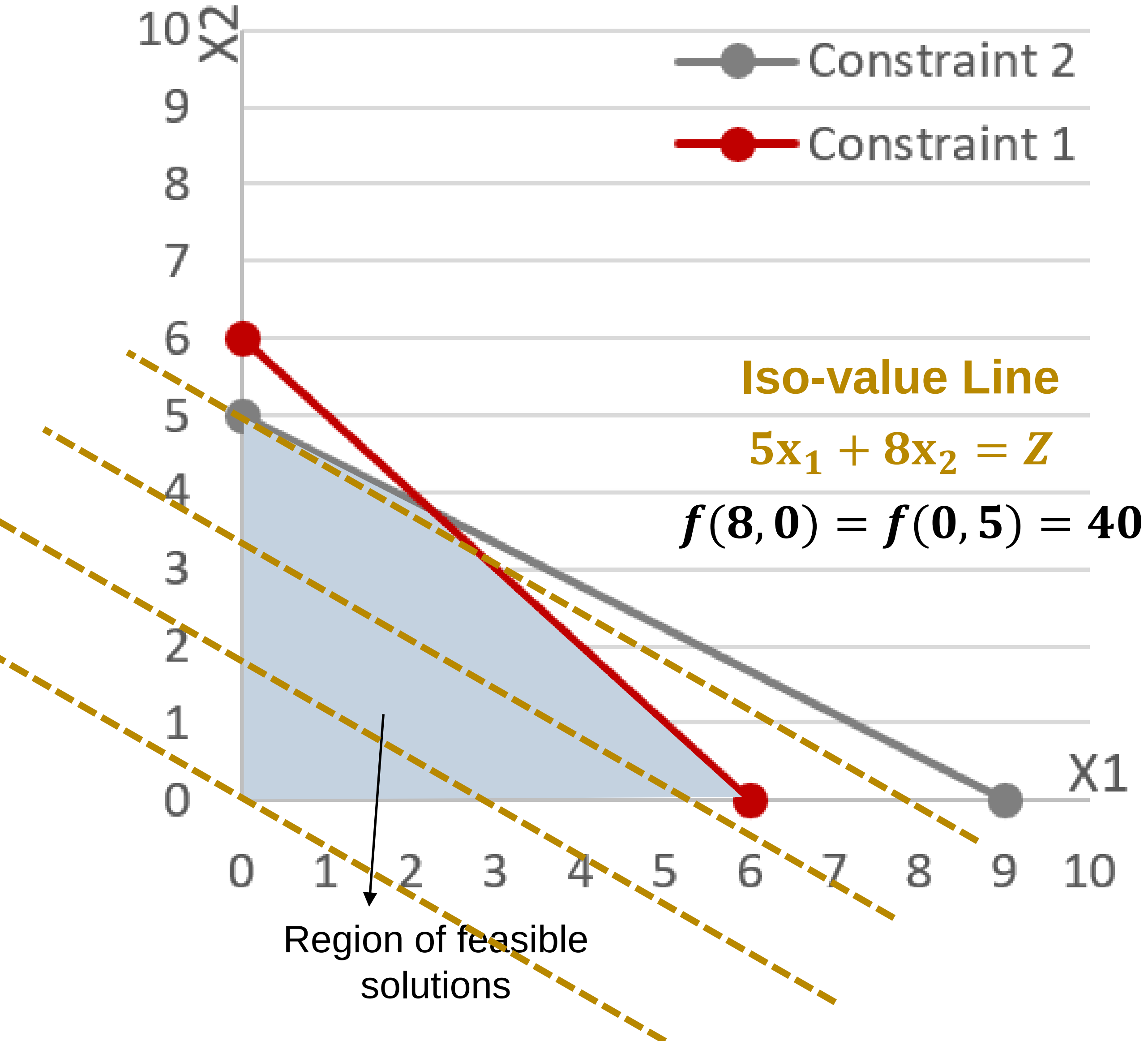
  $5x_1 + 9x_2 \leq 45$

  $x_1, x_2 \geq 0$

- **The Carpenter Problem**
  - A carpenter can either make chairs or tables
  - Chairs take 5 units of lumber, 1 day of labour, and the carpenter makes $500
  - Tables take 9 units of lumber, 1 day of labour, and the carpenter makes $800
  - 45 units of lumber available
  - 6 days of labour available per week

How many chairs and tables to produce per week

Linear Programming Model



$$\max\left(f(x_1, x_2) = 5x_1 + 8x_2\right)$$

$$\text{s.t.} \quad x_1 + x_2 \leq 6$$

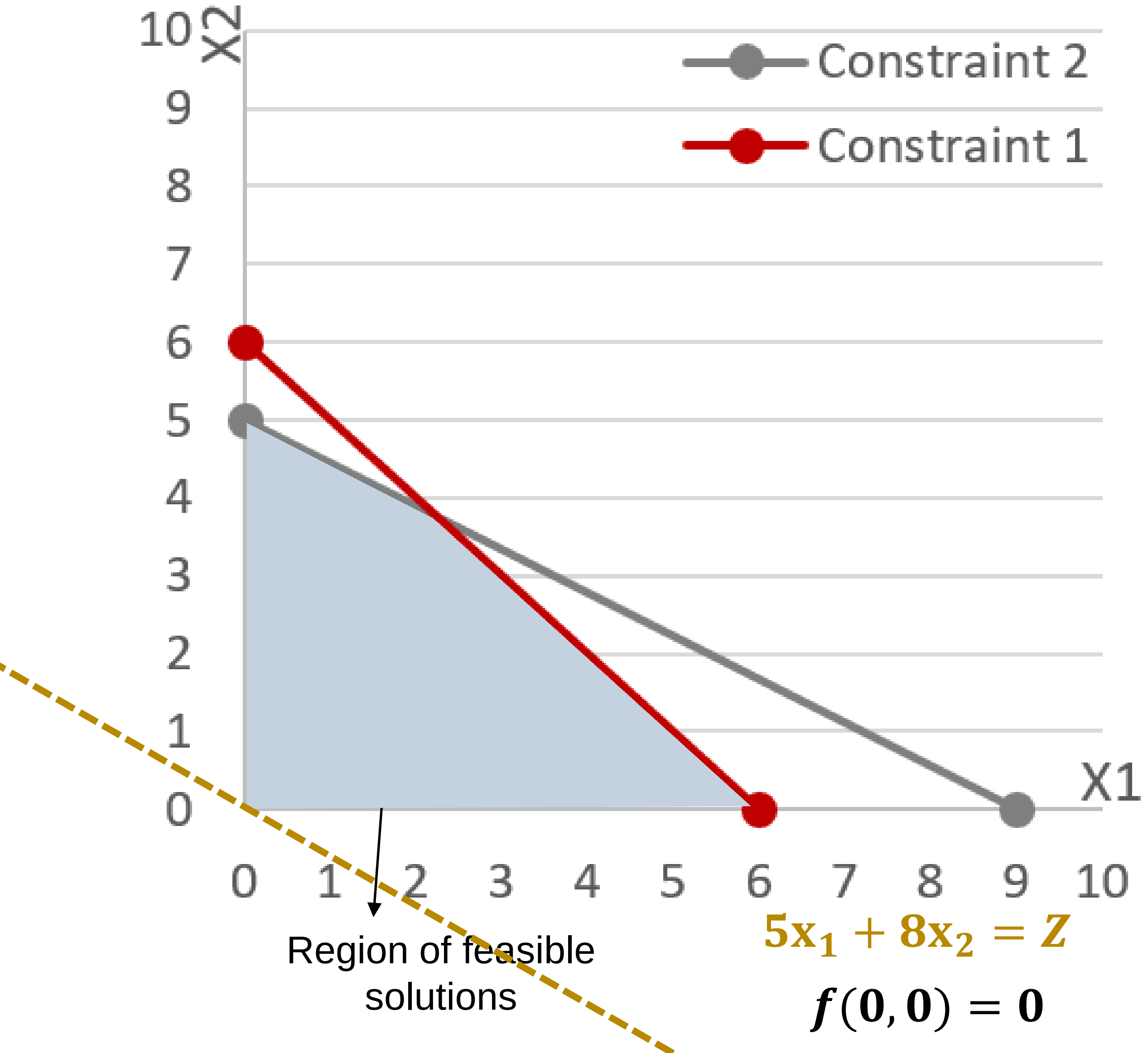
$$5x_1 + 9x_2 \leq 45$$

$$x_1, x_2 \geq 0$$

Constraint 1	
x1	x2
0	6
6	0

Constraint 2	
x1	X2
9	0
0	5

Linear Programming Model



$$\max(f(x_1, x_2) = 5x_1 + 8x_2)$$

s.t.

$$x_1 + x_2 \leq 6$$

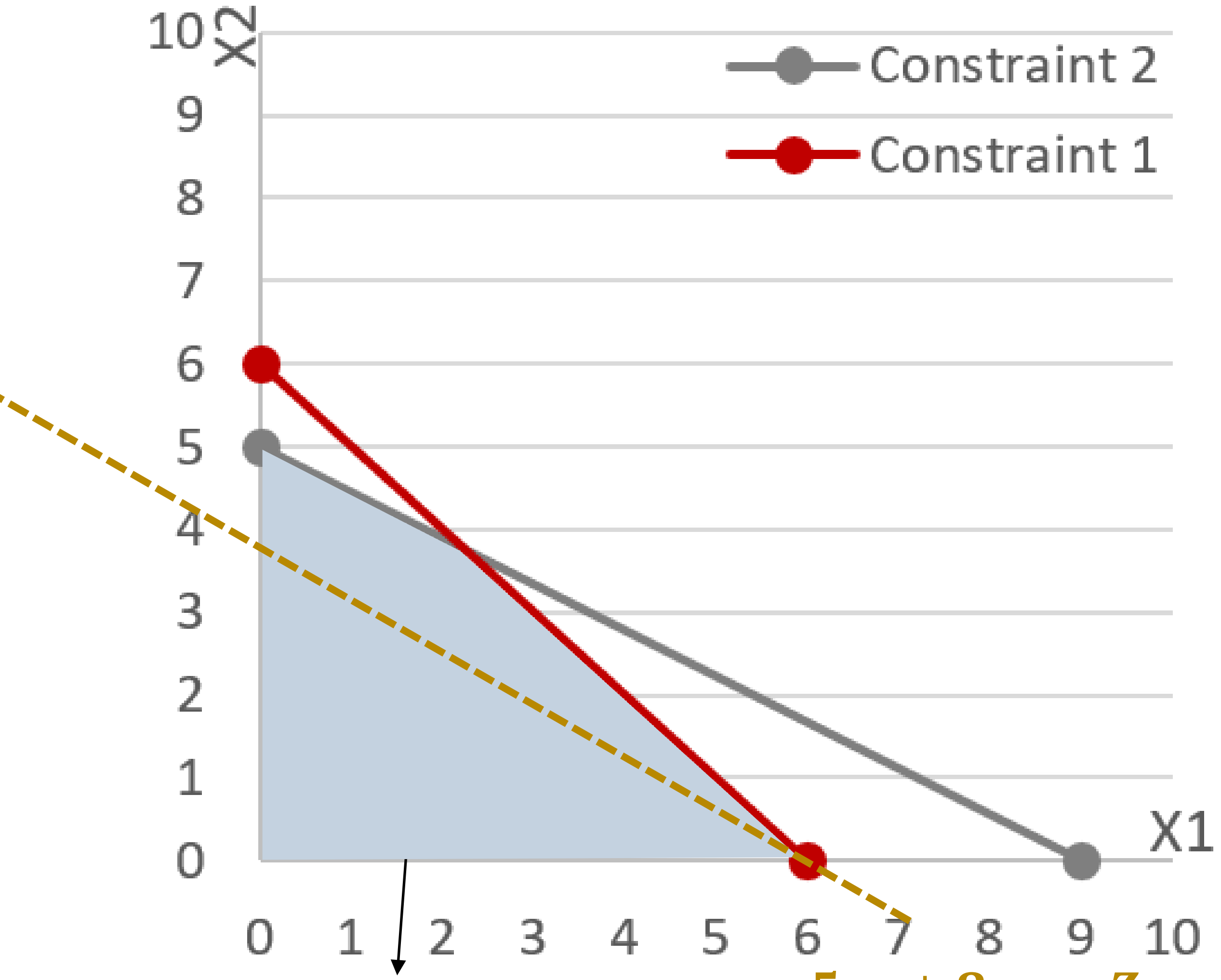
$$5x_1 + 9x_2 \leq 45$$

$$x_1, x_2 \geq 0$$

Constraint 1	
x1	x2
0	6
6	0

Constraint 2	
x1	X2
9	0
0	5

Linear Programming Model



Region of feasible solutions

$\mathbf{5x_1 + 8x_2 = Z}$

$\boldsymbol{f(6,0) = f(0,3.75) = 30}$

$$\max(f(x_1, x_2) = 5\mathrm{x}_1 + 8\mathrm{x}_2)$$

$$\text{s.t.} \qquad \mathrm{x}_1 + \mathrm{x}_2 \leq 6$$

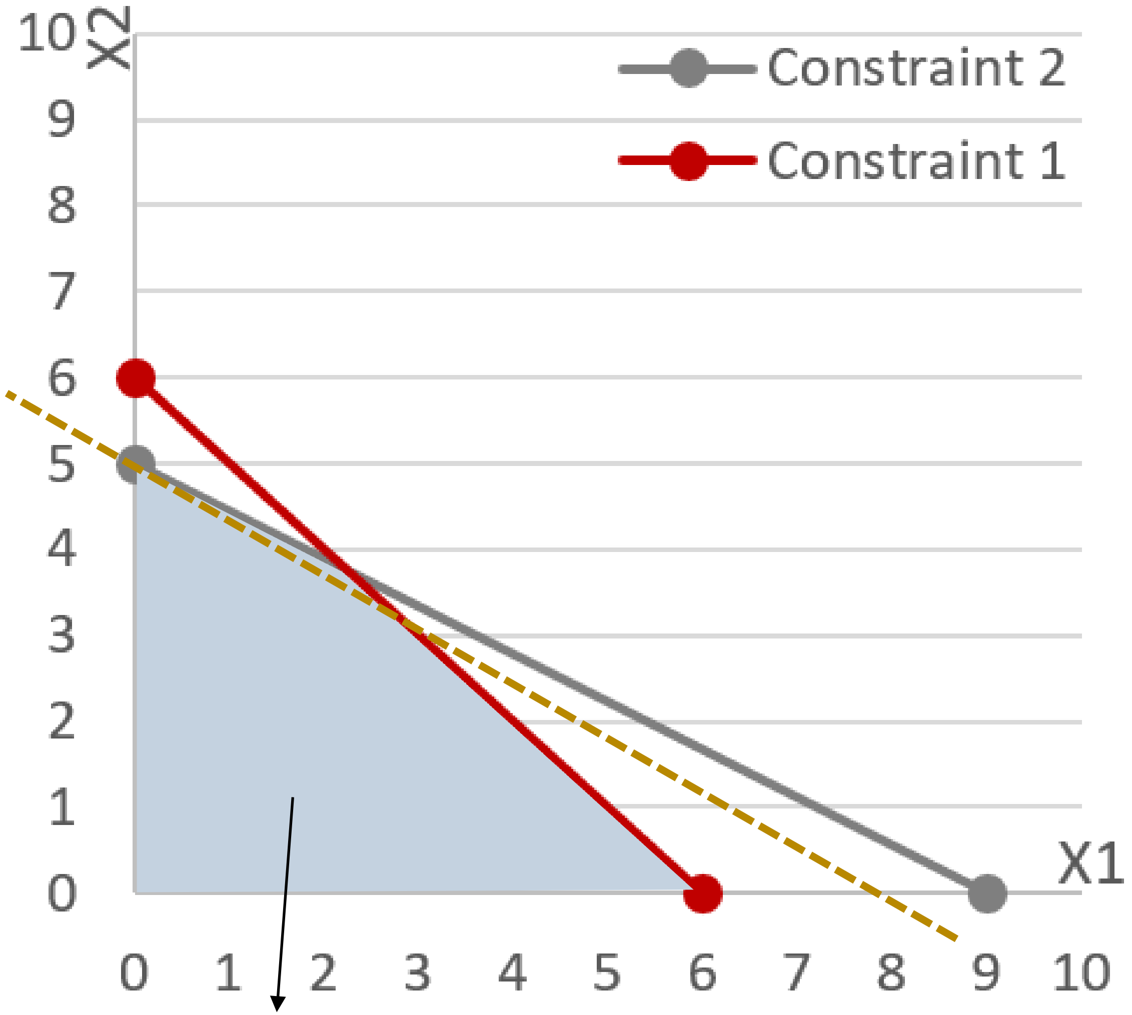
$$5\mathrm{x}_1 + 9\mathrm{x}_2 \leq 45$$

$$\mathrm{x}_1, \mathrm{x}_2 \geq 0$$

Constraint 1	
x1	x2
0	6
6	0

Constraint 2	
x1	X2
9	0
0	5

Linear Programming Model



Region of feasible solutions

$\mathbf{5x_1 + 8x_2 = Z}$

$\boldsymbol{f(8,0) = f(0,5) = 40}$

$$\max(f(x_1, x_2) = 5x_1 + 8x_2)$$

$$\text{s.t.} \qquad x_1 + x_2 \leq 6$$

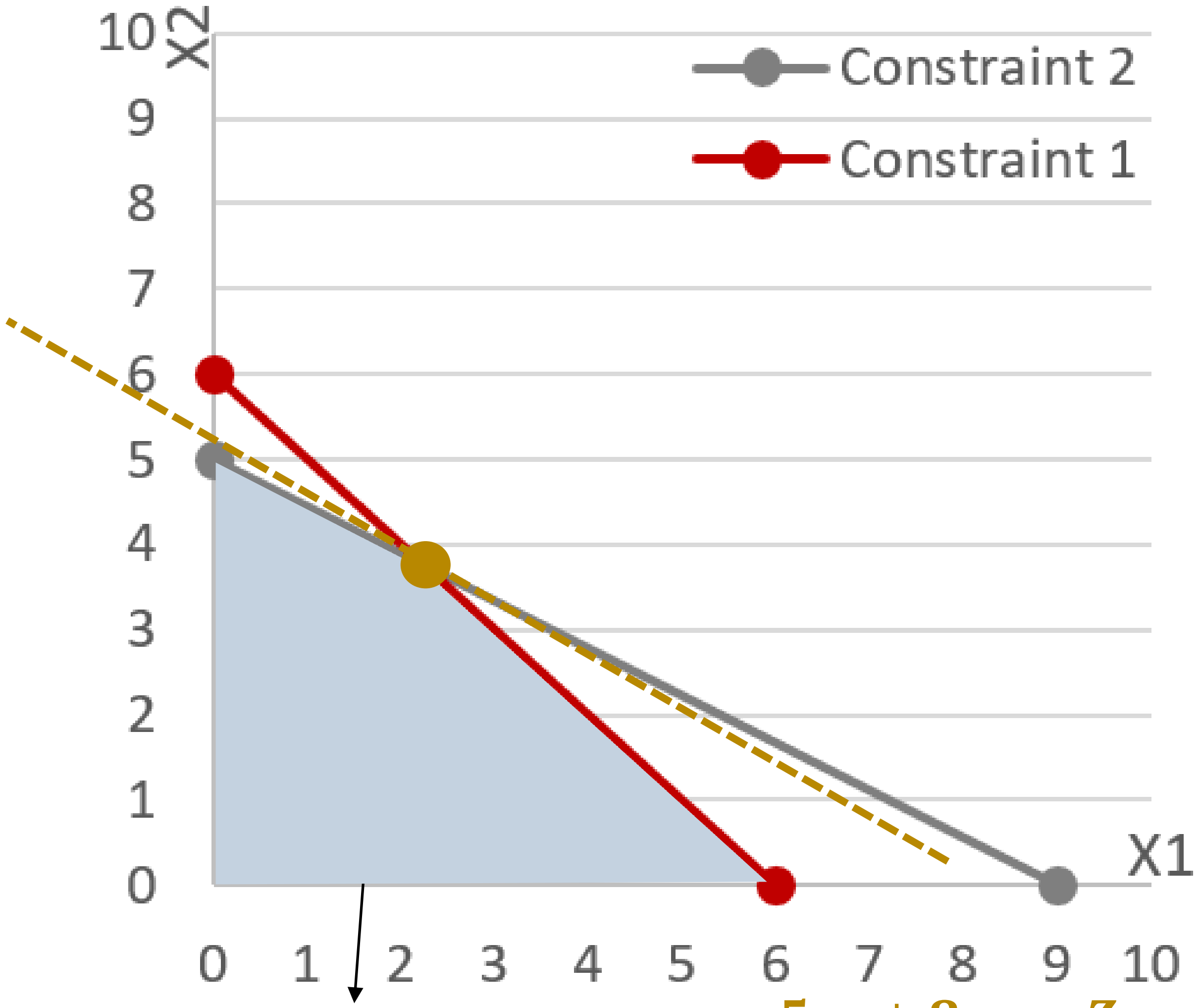
$$5x_1 + 9x_2 \leq 45$$

$$x_1, x_2 \geq 0$$

Constraint 1	
x1	x2
0	6
6	0

Constraint 2	
x1	X2
9	0
0	5

Linear Programming Model



Region of feasible solutions

$\mathbf{5x_1 + 8x_2 = \mathit{Z}}$

$\boldsymbol{f(2.25, 3.75) = 41.25}$

$$\max(f(x_1, x_2) = 5\mathrm{x}_1 + 8\mathrm{x}_2)$$

$$\text{s.t.} \qquad \mathrm{x}_1 + \mathrm{x}_2 \leq 6$$

$$5\mathrm{x}_1 + 9\mathrm{x}_2 \leq 45$$

$$\mathrm{x}_1, \mathrm{x}_2 \geq 0$$

Constraint 1	
x1	x2
0	6
6	0

Constraint 2	
x1	X2
9	0
0	5

$from\ constraint\ 1;\ x_2 = 6 - x_1$

$from\ constraint\ 2;\ x_2 = \frac{45 - 9x_2}{5}$

$6 - x_1 = \frac{45 - 9x_2}{5} \Leftrightarrow x_2 = 3.75\,;\, x_1 = 2{,}25$

Standard Linear Programming Model

profit

$$max\ \boxed{z} = p_1x_1 + p_2x_2 + \cdots + p_ix_i$$

budget

$$s.t. \quad \boxed{c_{11}}x_1 + c_{21}x_2 + \cdots + c_{i1}x_n \leq \boxed{b_1}$$

cost

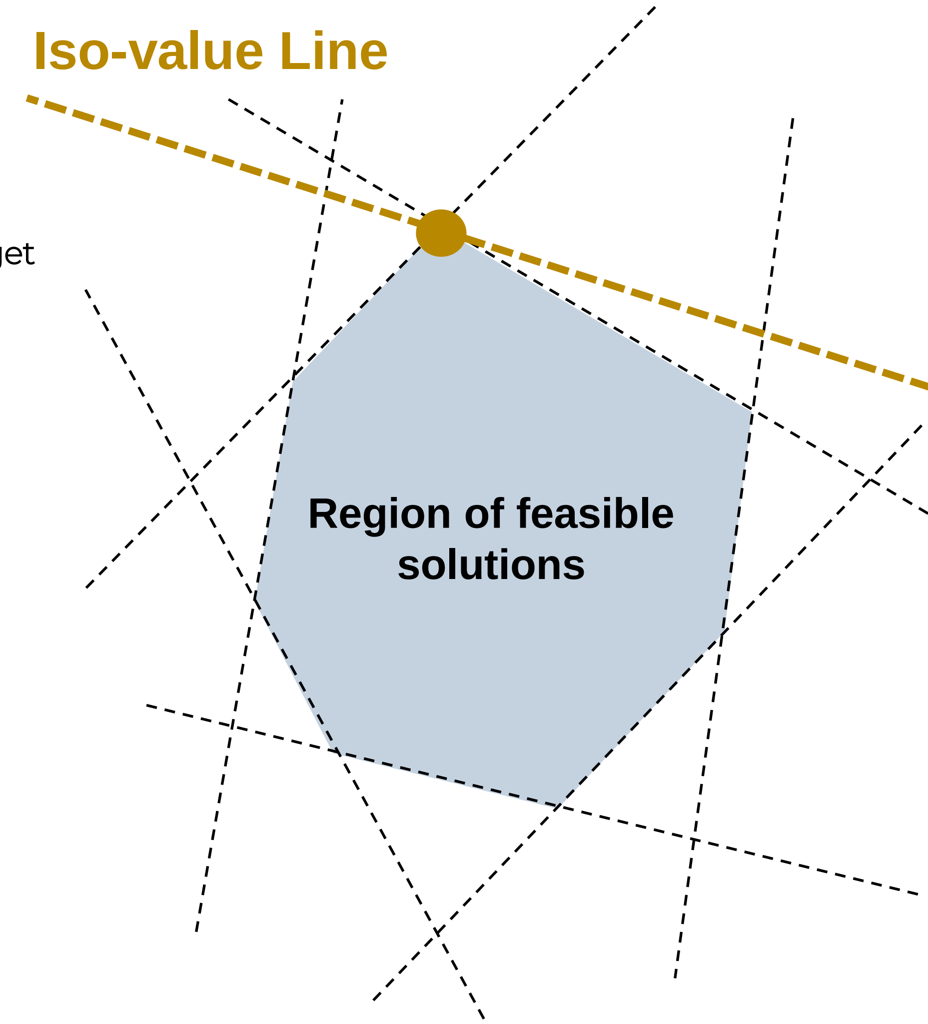
$$c_{12}x_1 + c_{22}x_2 + \cdots + c_{i2}x_n \leq b_2$$

$$\ldots$$

$$c_{1j}x_1 + c_{2j}x_2 + \cdots + c_{ij}x_i \leq b_j$$

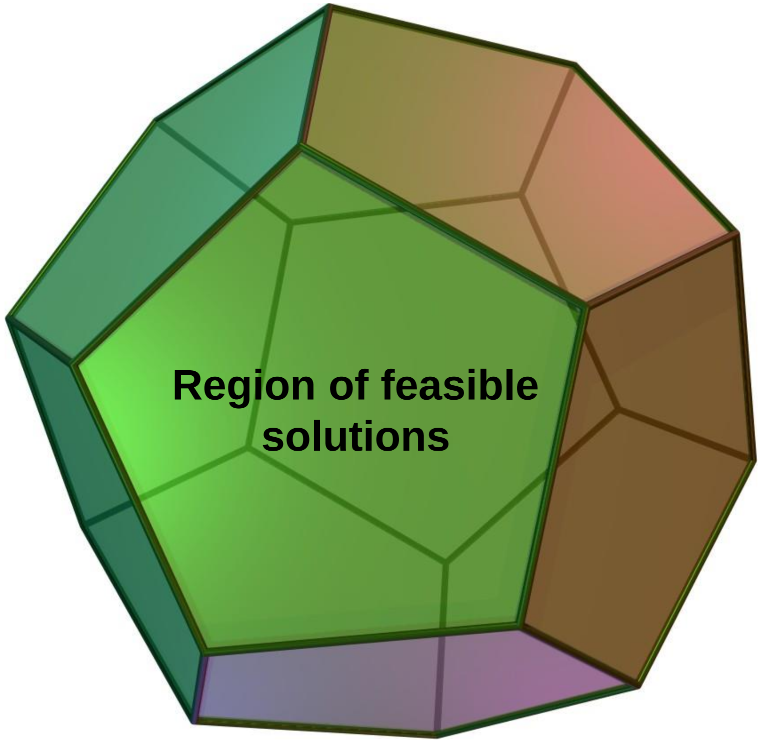
$$\boxed{x_1, x_2, \ldots, x_i} \geq 0$$

Jobs



Idea of Simplex Algorithm

- The *simplex algorithm*, created by the American mathematician George Dantzig in 1947, is a very popular algorithm for solving linear programs.
- The *Simplex method* uses row operations on matrices in Linear Algebra to find the optimal solution of an LP
  - Start at a corner of the feasible region
  - While there is an adjacent corner that is a better solution, move to that corner.
  - For “most” instances, the algorithm terminates (in a finite number of steps) at an optimal solution.



https://sites.google.com/view/40-510/home

- Other more sophisticated methods have also been proposed to solve LP problems, such as the *ellipsoid method* or the *barrier method*

Standard LP Model Formulation

- **Sets**

$Set\ of\ Jobs: i = 1,2, \dots, i$

$Set\ of\ Constraints: j = 1,2, \dots, j$

- **Parameters**

$p_i = unit\ \ of\ profit\ of\ working\ on\ job\ i\ (profit\ per\ unit\ of\ time)$

$c_{ij} = cost\ of\ job\ i\ under\ constraint\ j\ (e.g. manpower, resources, inventory, etc.)$

$b_j = \ budget\ available\ under\ constraint\ j (e.g. no. labours, amount\ of\ resources, inv. capacity, etc.)$

- **Decision Variables**

$x_i = amount\ of\ time\ working\ on\ job\ i$

Standard LP Model Formulation

$$max\, z = \sum_{i \in N} p_i x_i$$

Sum over all jobs

Maximize profits

$$s.t. \quad \sum_{i \in N} c_{ij} x_i \leq b_j\ , \qquad \forall j \in M$$

Sum over all jobs

For all constraints

Subject to budget constraints

$$\boxed{x_i \geq 0,} \qquad \forall i \in N$$

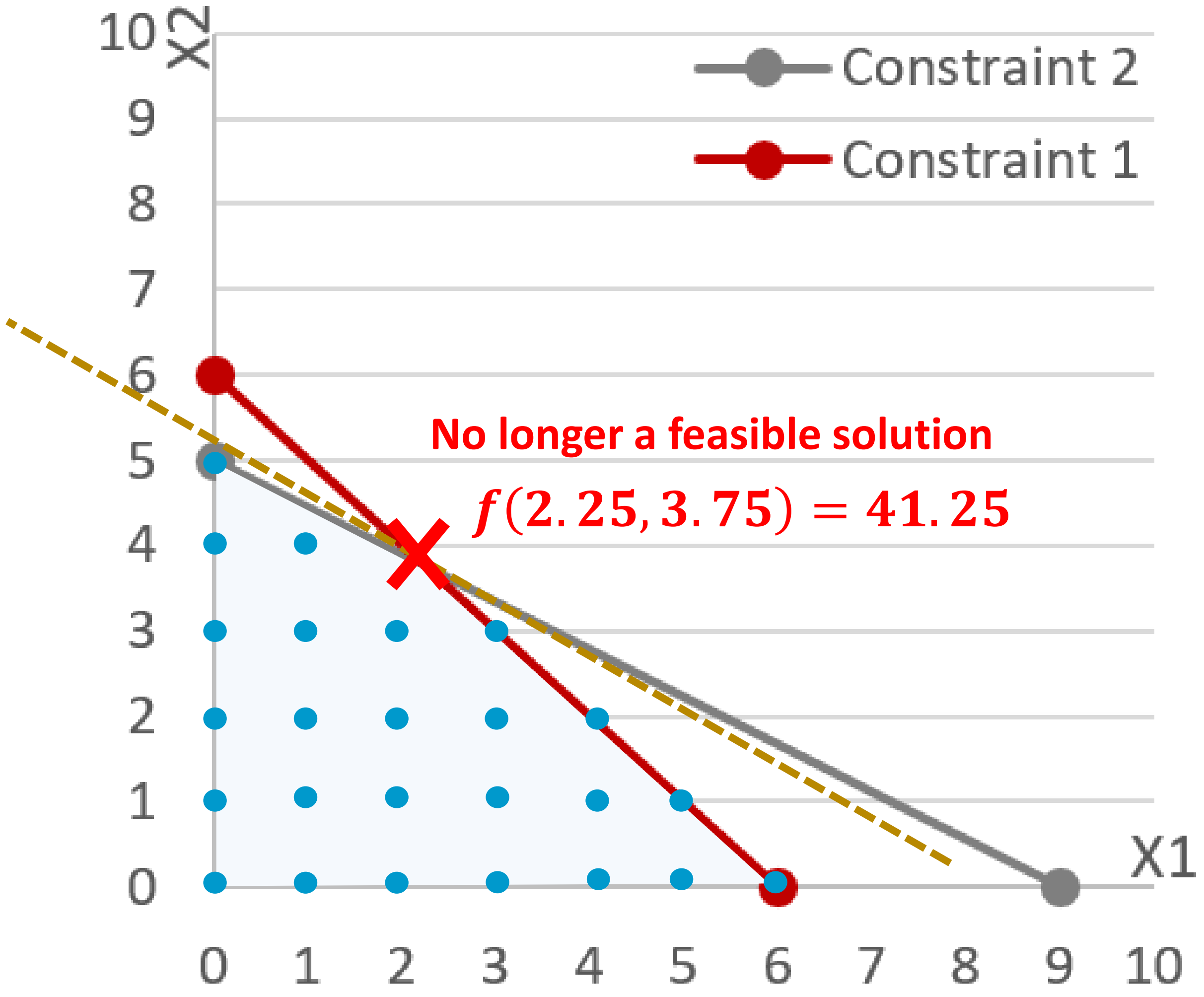
For all jobs

I cannot spend negative time working on job *i*

What if x_i needs to be integer

i.e. x_i is the number of job *i* completions (e.g. number of chairs or tables?

Integer Programming Model



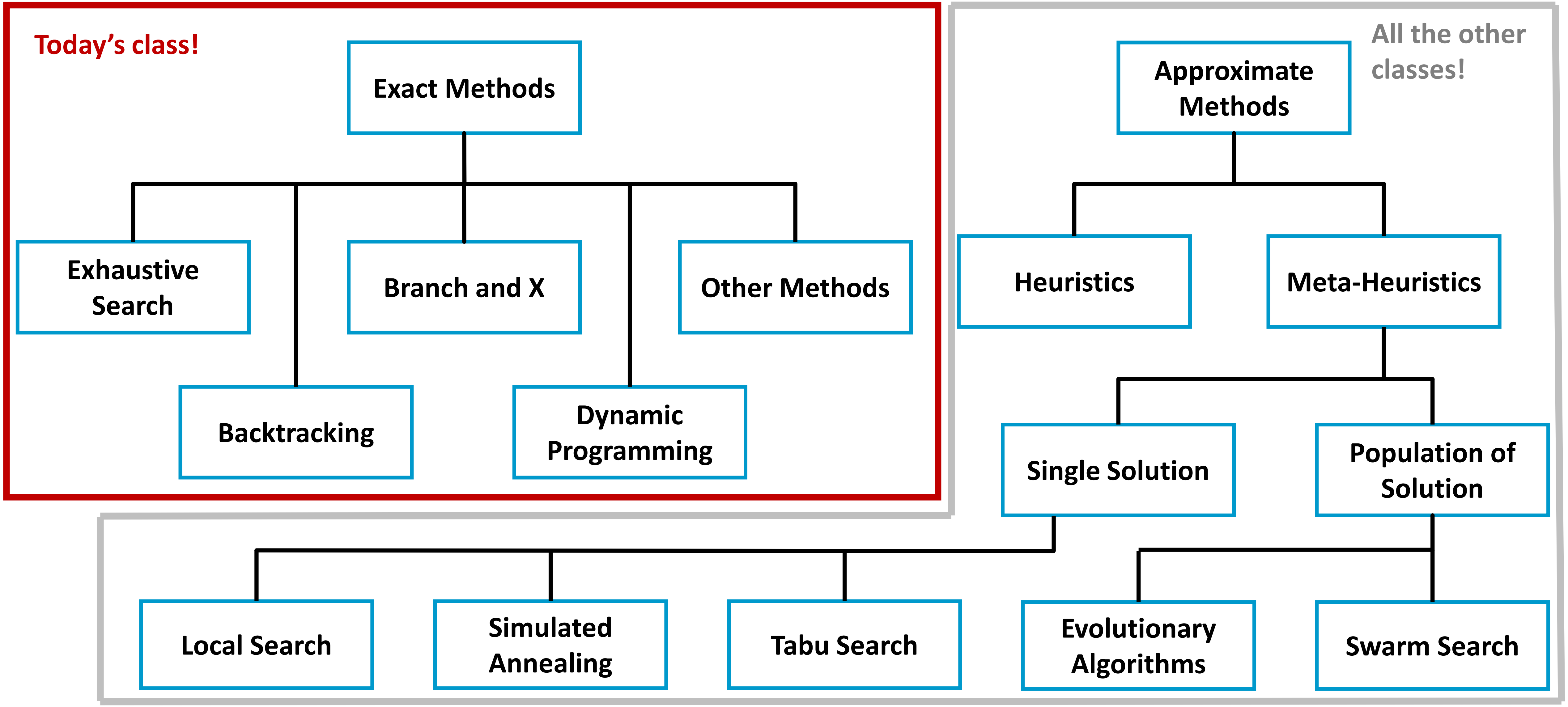
$$\max(f(x_1, x_2) = 5x_1 + 8x_2)$$

s.t. $\quad x_1 + x_2 \leq 6$

$$5x_1 + 9x_2 \leq 45$$

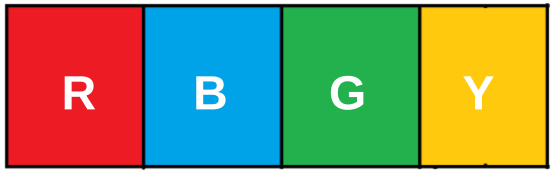
$$\mathbf{x_1, x_2 \geq 0 \;, integer}$$

Solving Discrete Optimization Problems

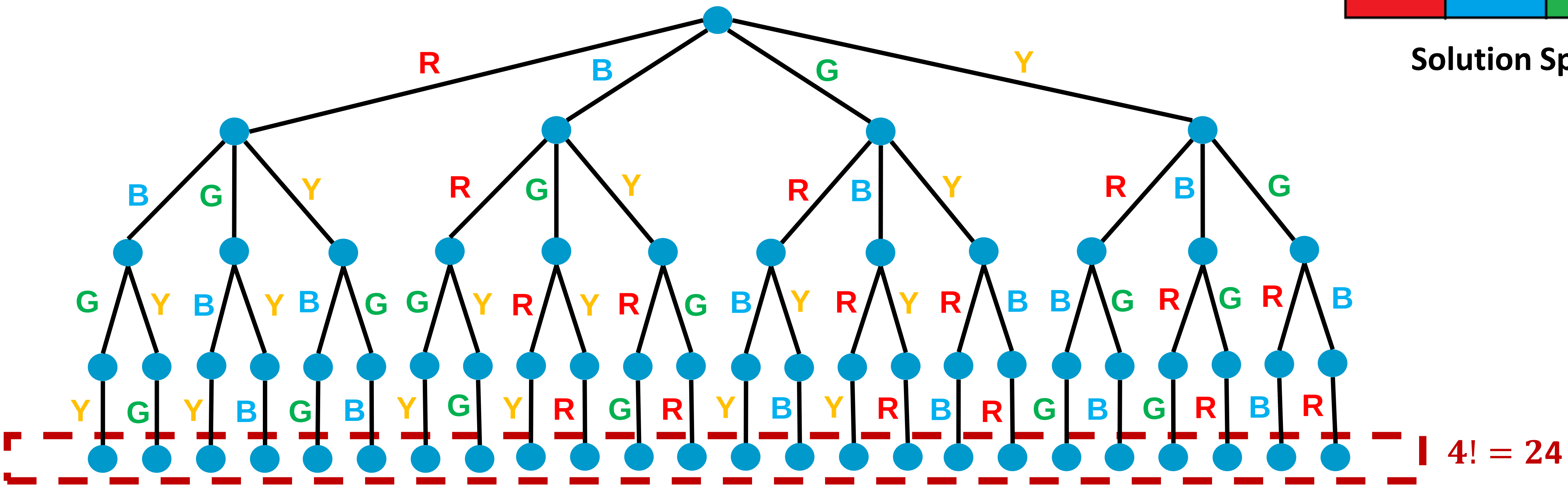


Exaustive Search

- Exhaustive Search is the simplest of the algorithms. It examines every possible combination of permitted levels of all attributes.
- Exhaustive Search is very ineffective and mostly unusable for a real-world problem due to time limitations
- Solutions are generally represented in a **search space tree**



Solution Space = $n!$



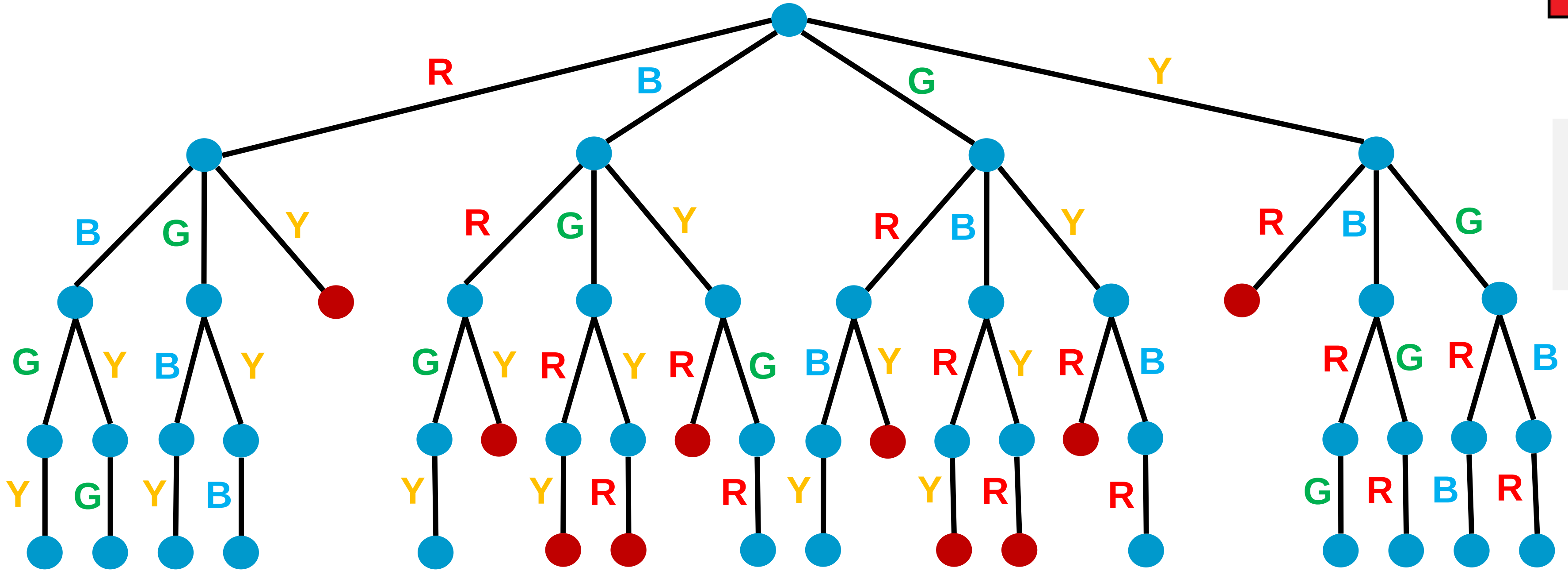
Backtracking

- Backtracking is an algorithmic technique where the goal is to get **one or multiple solutions** to a problem.
- Backtracking **depth-searches** for solutions and then backtracks to the most recent valid path as soon as an end node is reached (i.e., we can proceed no further).
- It is eventually faster than exhaustive search since the search space tree is cut whenever a **bounding constraint** is violated

R	B	G	Y

Solution Space = $n!$

Bounding Constraint
Yellow and red cannot be adjacent colours



12 feasible solutions

Example: Sum of Subsets Problem

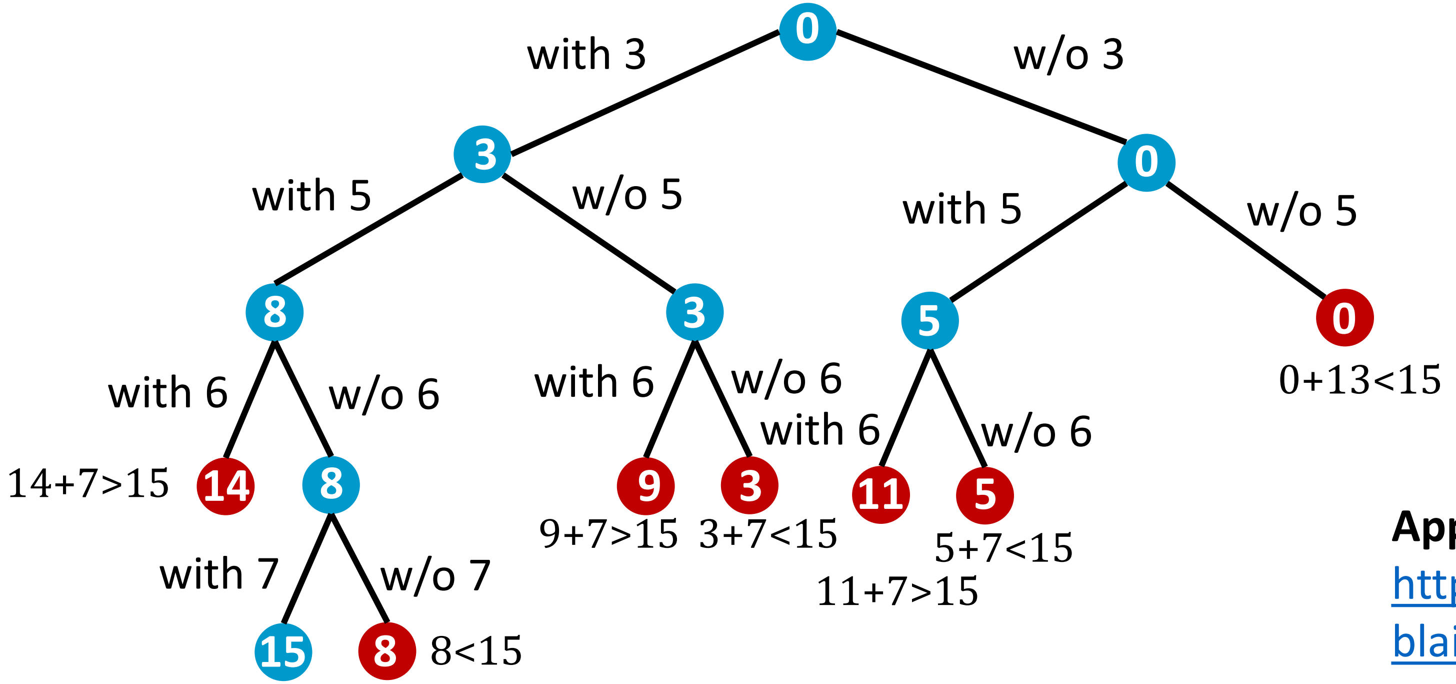
- Subset sum problem is to find subset of elements that are selected from a given set with n elements whose sum adds up to a given number m.
- We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

$$W[1:n] = \{3,5,6,7\}$$

$$m = 15$$

Solution Space = $2^n = 16$

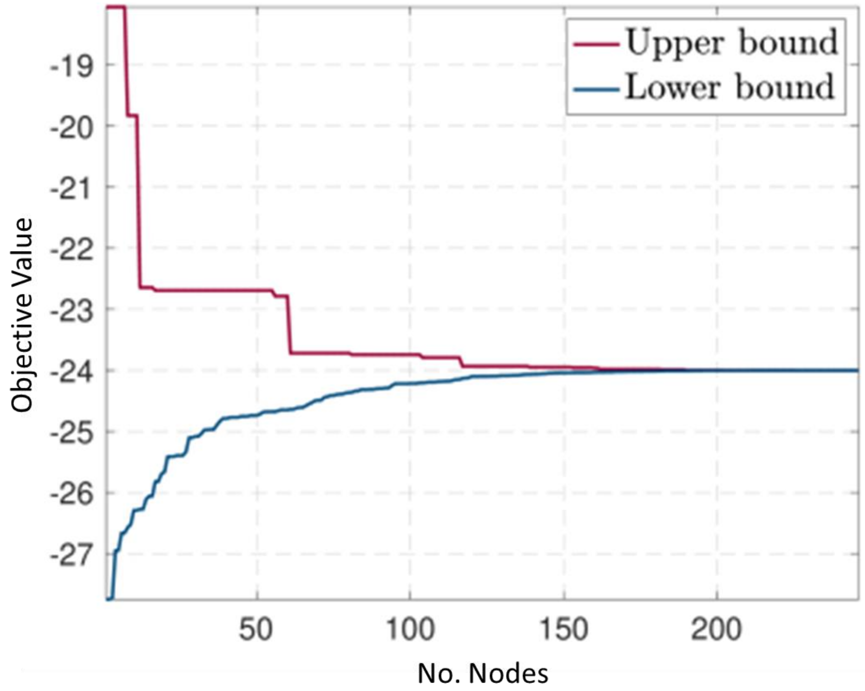
1 feasible solutions



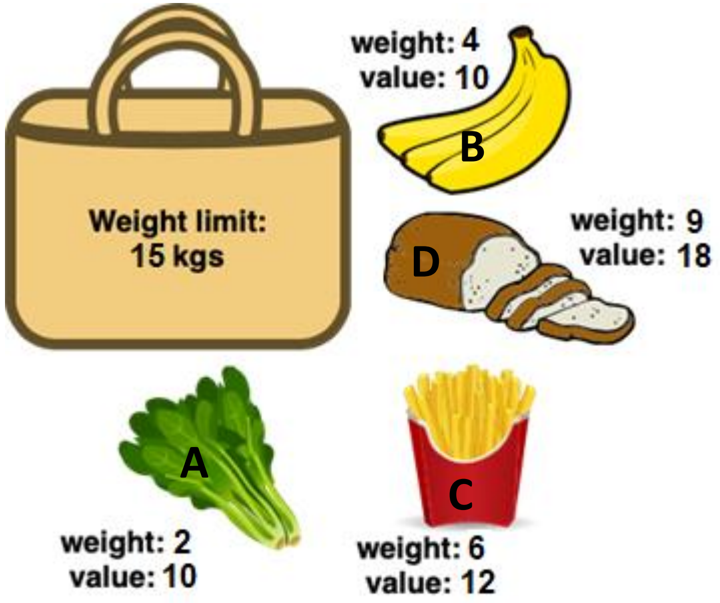
Applications (e.g. computer passwords):
http://www.math.stonybrook.edu/~scott/blair/Other_uses_subset_sum.html

Branch and Bound

- Rely on two subroutines that (efficiently) compute a lower and an upper bound on the optimal value
  - upper bound can be found by choosing any point in the region, or by a local optimization method
  - lower bound can be found from by applying some relaxation techniques (e.g. LP relaxation)
- Definitions
  - Upper bound: a feasible solution
  - Lower bound: a solution to an “easier” problem
  - Node elimination: (fathom nodes): when lower bound >= upper bound
- Branch and Bound assumes we are solving minimization problems



Example: Knapsack Problem



Item	A	B	C	D
Value	-10	-10	-12	-18
Weight	2	4	6	9

Example: Knapsack Problem

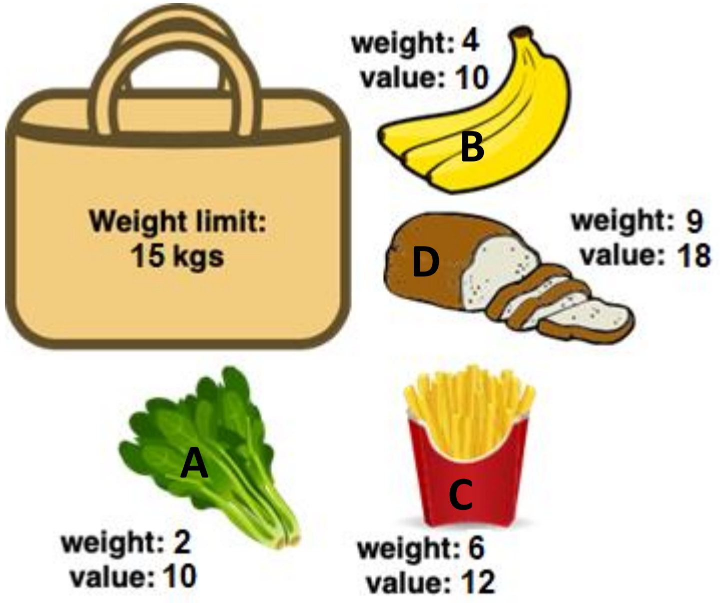
- Pick a random initial feasible solution **ABC**

$$Value = 10 + 10 + 12 = 32$$

$$Weight = 2 + 4 + 6 = 12$$

Upper Bound = $10 + 10 + 12 = 32$
(best you could achieve so far)

Lower Bound= ?
(best you could achieve in theory)



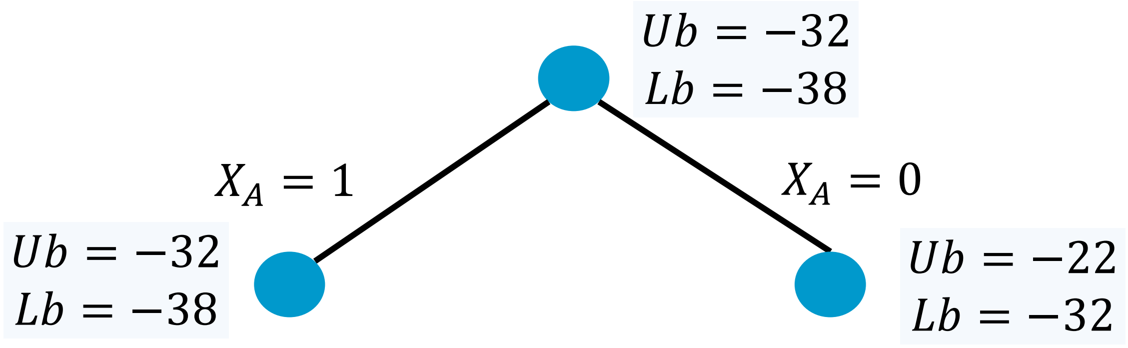
Let's imagine you could bring a portion of item **D**. What would be the portion to add to solution **ABC**

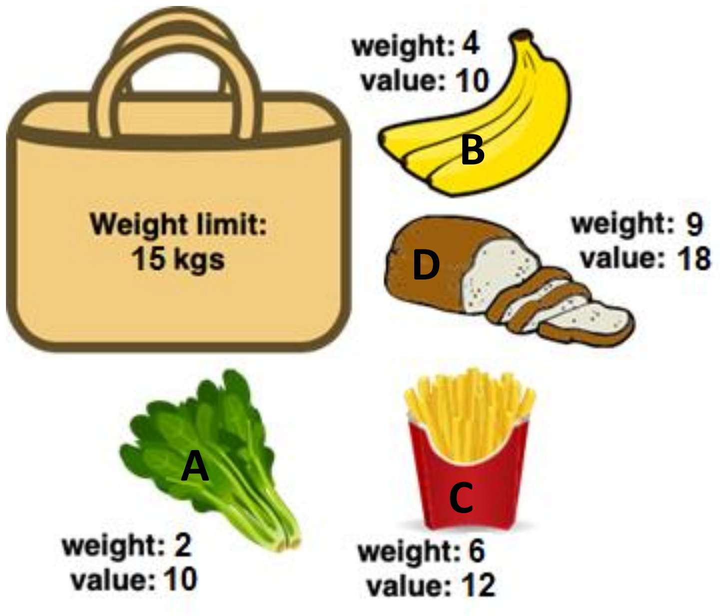
Item	A	B	C	D
Value	-10	-10	-12	-18
Weight	2	4	6	9

$$Weight = 2 + 4 + 6 = 12 + 3 = 15$$
$$Portion = 3 \div 9$$
$$Value = 10 + 10 + 12 + 18 \times 3 \div 9 = \mathbf{38}$$

Example: Knapsack Problem

$Upper = -32$
$Lower = -38$





Item	A	B	C	D
Value	-10	-10	-12	-18
Weight	2	4	6	9

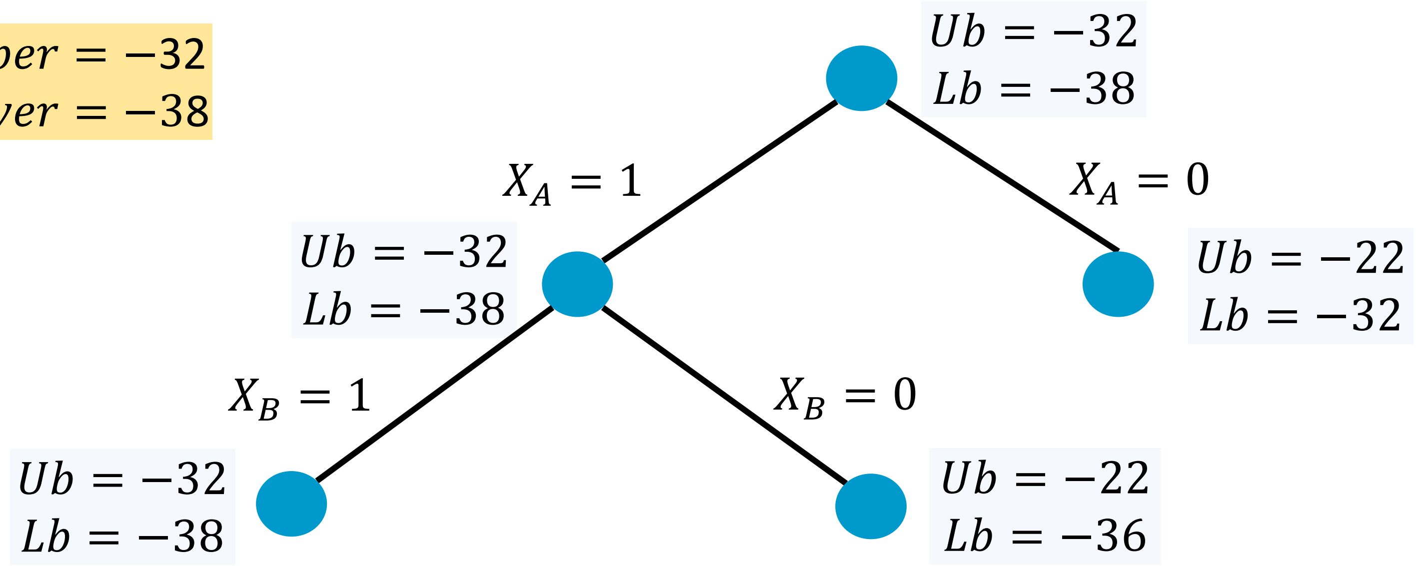
$\text{Weight} = 4 + 6 = 10\ ;\ \text{Weight} = 15 - 10 = 5$

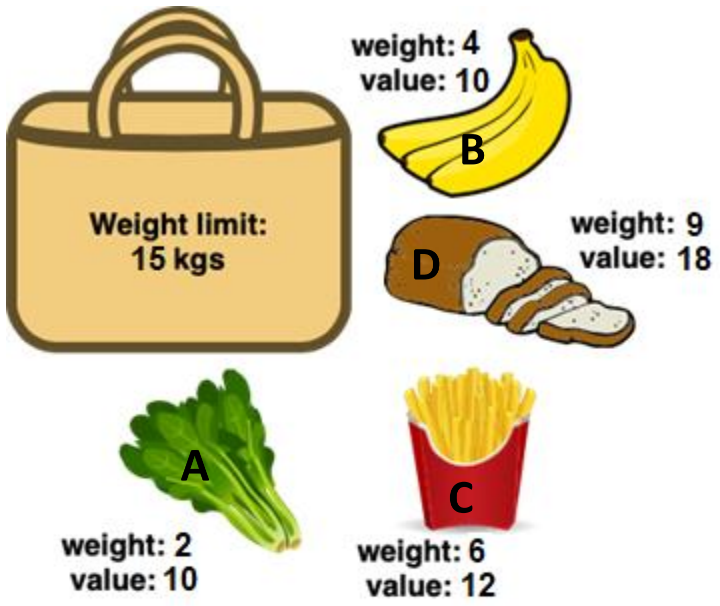
$\text{Upper Bound} = 10 + 12 = \mathbf{22}$

$$Lower\ Bound = 10 + 12 + \frac{18 \times 5}{9} = \mathbf{32}$$

Example: Knapsack Problem

$Upper = -32$
$Lower = -38$





Item	A	B	C	D
Value	10	10	12	18
Weight	2	4	6	9

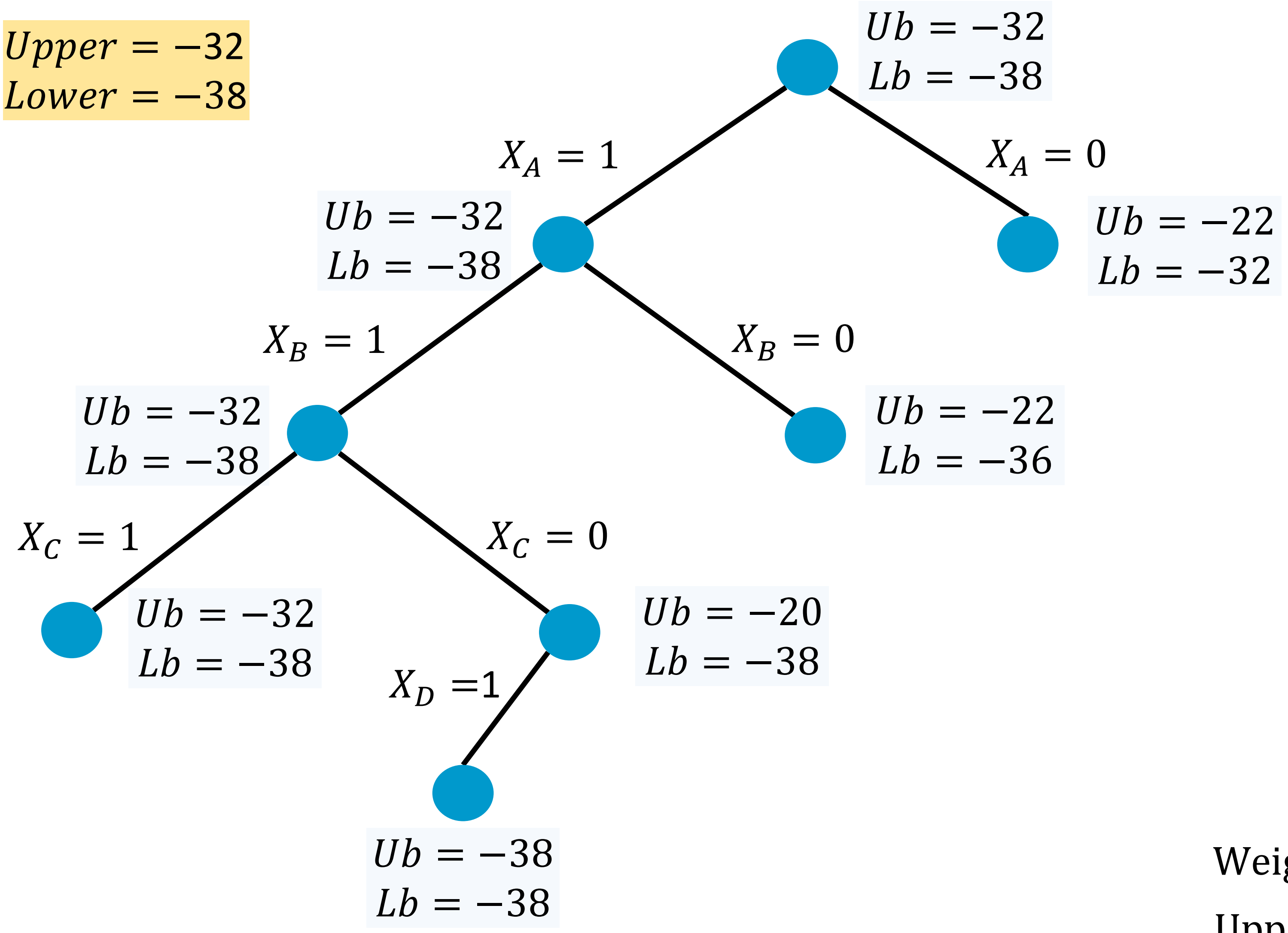
$\text{Weight} = 2 + 6 = 8$; $\text{Weight} = 15 - 8 = 7$

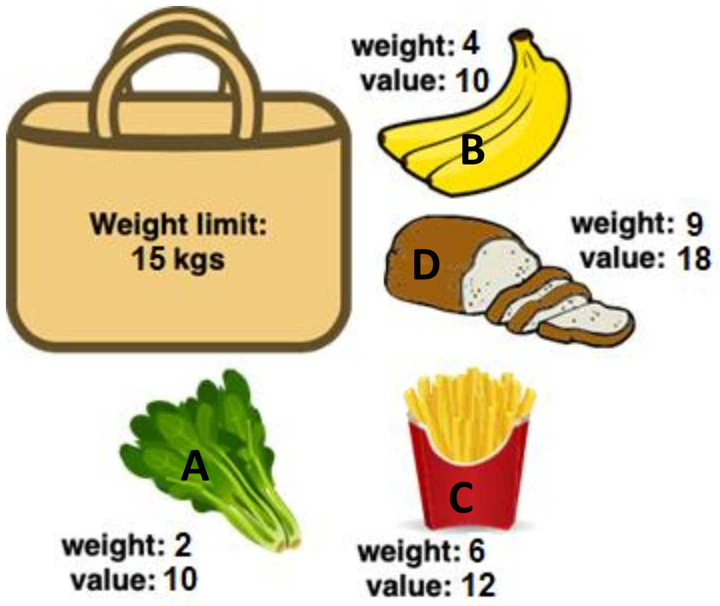
$\text{Upper Bound} = 10 + 12 = \mathbf{22}$

$$Lower\ Bound = 10 + 12 + \frac{18 \times 7}{9} = \mathbf{36}$$

Example: Knapsack Problem

$Upper = -32$
$Lower = -38$





Item	A	B	C	D
Value	-10	-10	-12	-18
Weight	2	4	6	9

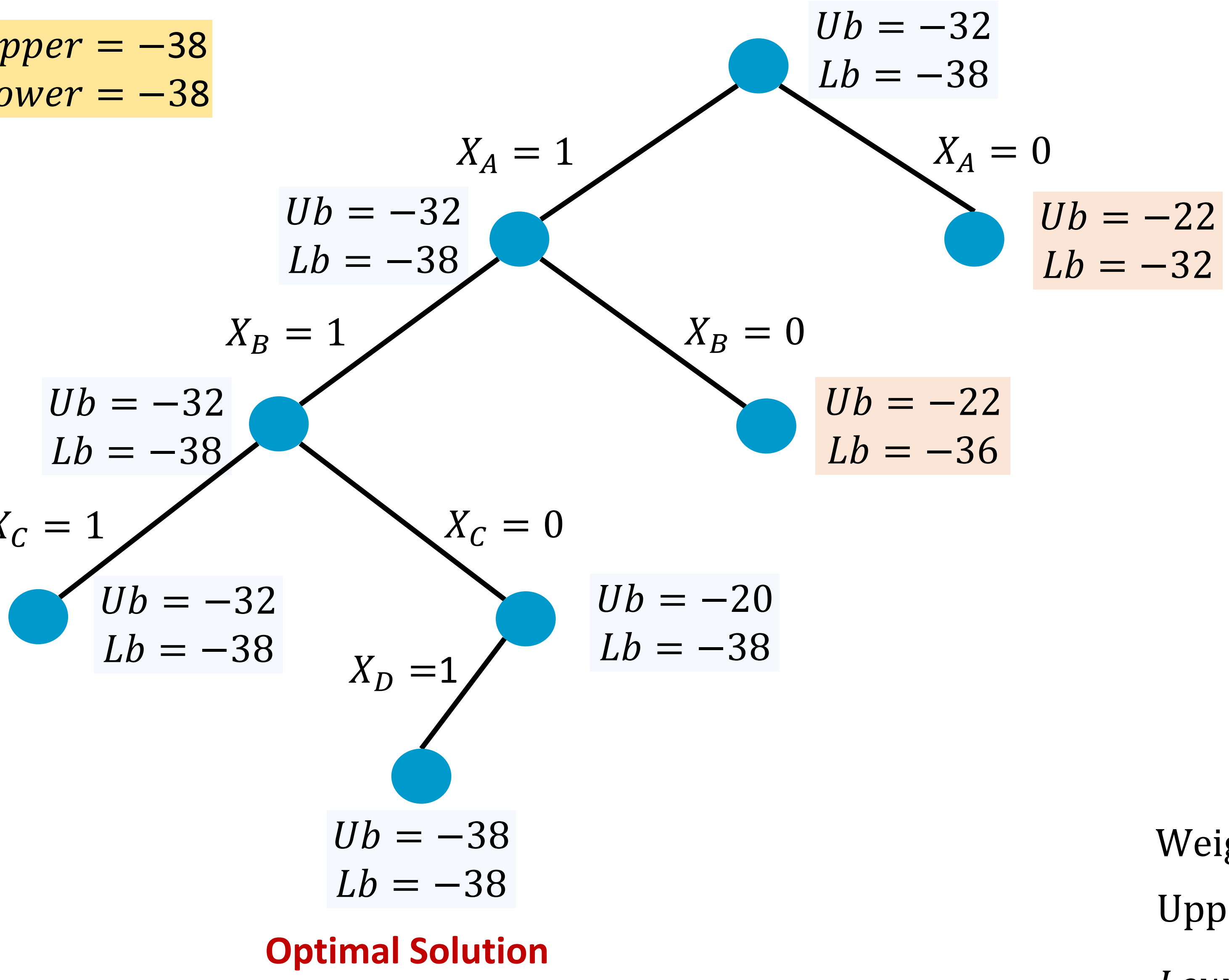
$\text{Weight} = 2 + 4 = 6$; $\text{Weight} = 15 - 6 = 9$

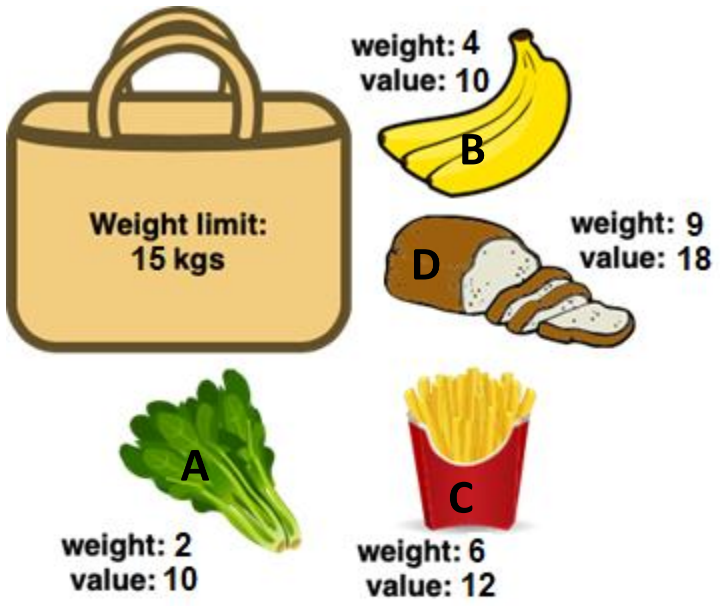
$\text{Upper Bound} = 10 + 10 = \mathbf{20}$

$$Lower\ Bound = 10 + 10 + \frac{18 \times 9}{9} = \mathbf{38}$$

Example: Knapsack Problem

$Upper = -38$
$Lower = -38$





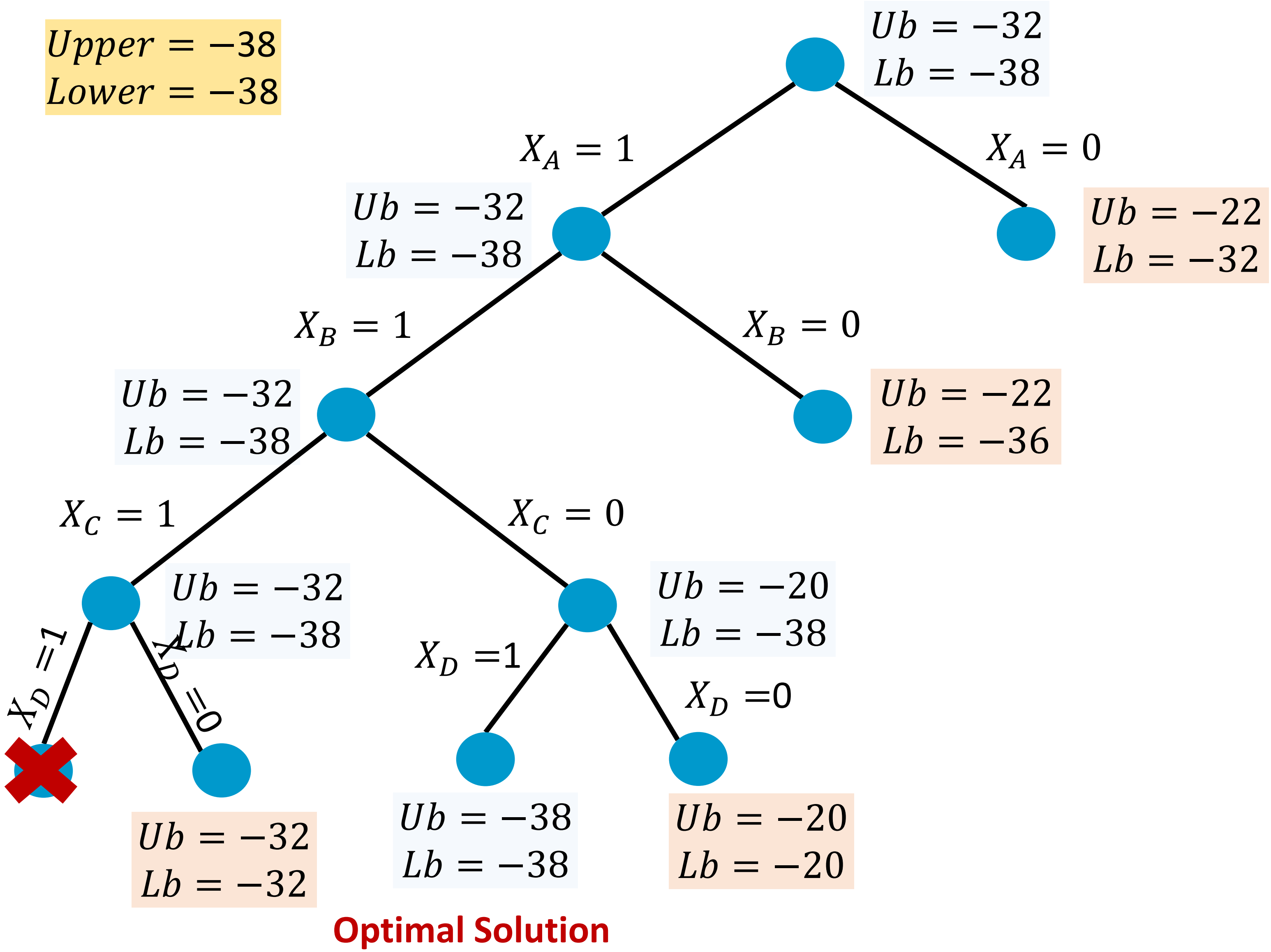
Item	A	B	C	D
Value	-10	-10	-12	-18
Weight	2	4	6	9

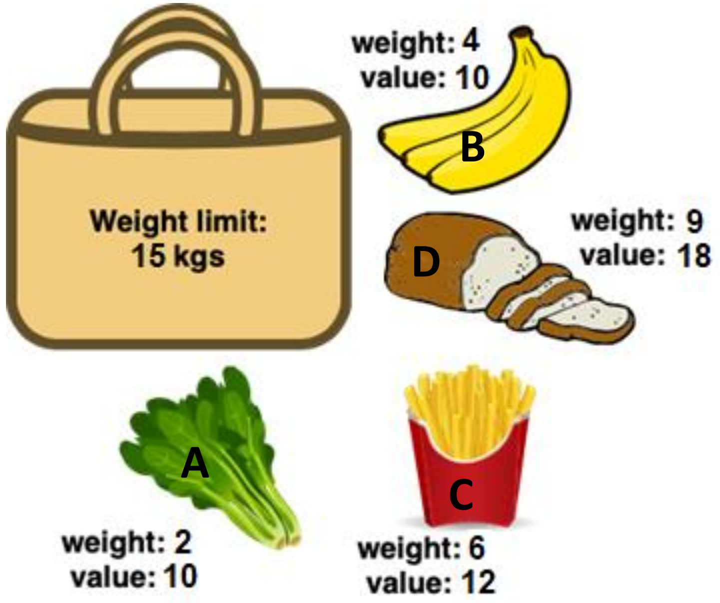
$\text{Weight} = 2 + 4 + 9 = 15$; $\text{Weight} = 15 - 15 = 0$

$\text{Upper Bound} = 10 + 10 + 18 = \mathbf{38}$

$$Lower\ Bound = 10 + 10 + \frac{18 \times 9}{9} = \mathbf{38}$$

Example: Knapsack Problem



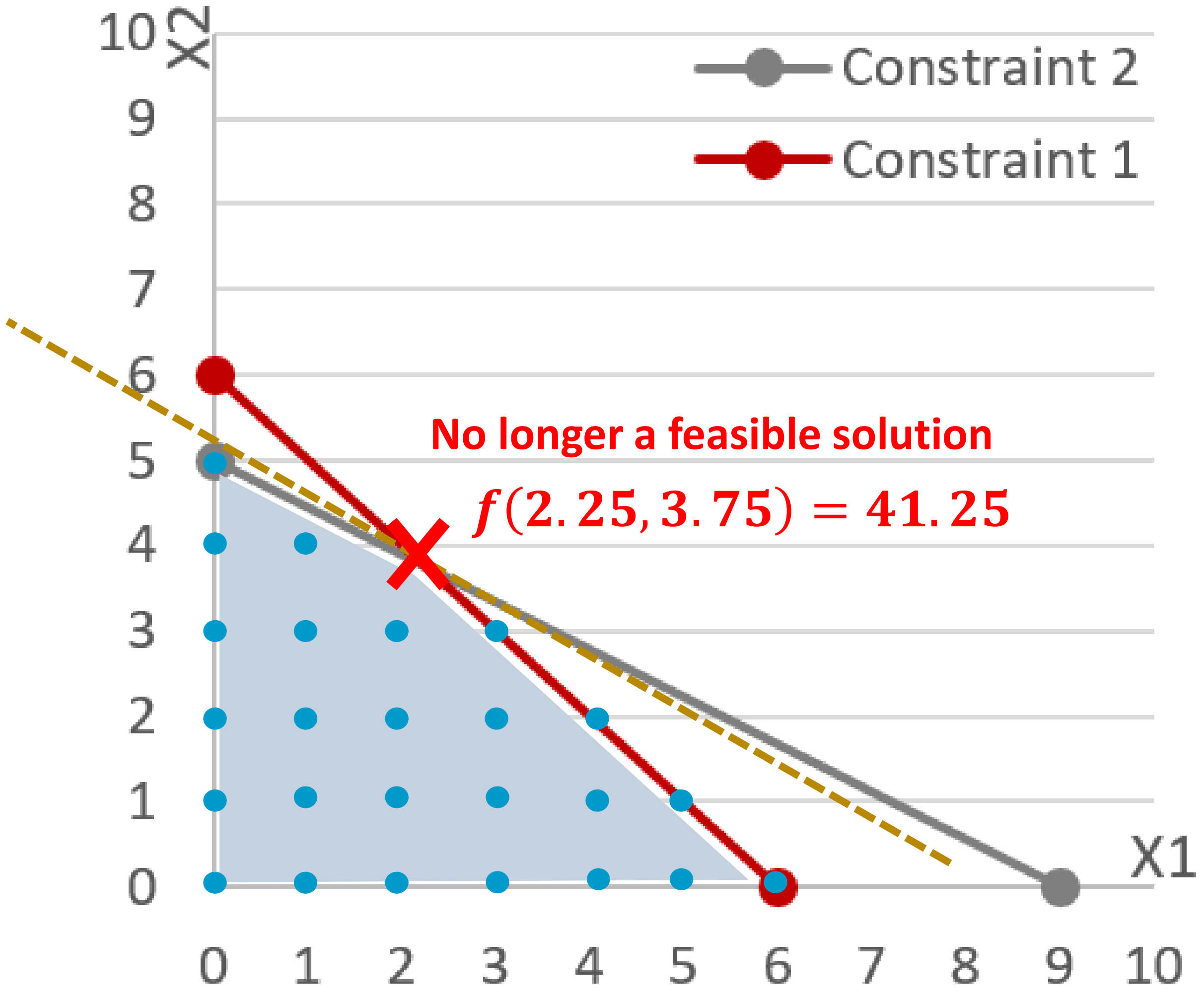


Item	A	B	C	D
Value	-10	-10	-12	-18
Weight	2	4	6	9

Branch and Bound + Simplex

- **Branch and bound builds on the top of LP methods**
  - If the LP is infeasible (i.e., has no solution), then the IP is also infeasible
  - If the LP problem has an integer-valued optimal solution, then the solution is equal to the optimal solution for the IP problem.
  - If the LP problem has a non-integer-value, then the problem needs to be decomposed
- **Branch and bound uses 2 heuristics:**
  - The branch heuristic:
    - Used to force the Simplex Method away from a non-integer valued solution - Many different LP problems are generated in the process
  - The bound heuristic:
    - Used to limit the number of LP problems generated by the branch heuristic

Branch and Bound + Simplex



$$\max(f(x_1, x_2) = 5x_1 + 8x_2)$$

$$\text{s.t.} \quad x_1 + x_2 \leq 6$$

$$5x_1 + 9x_2 \leq 45$$

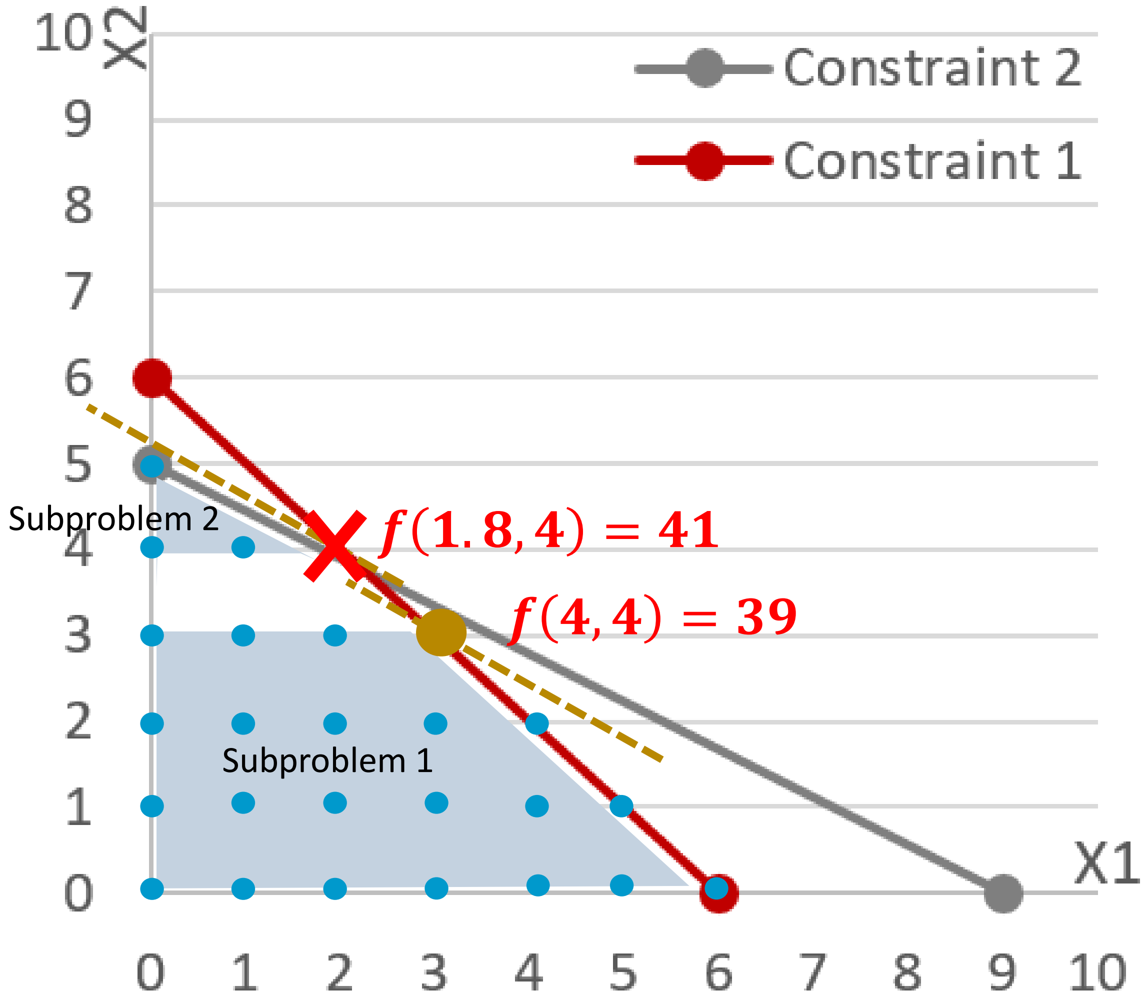
$$\mathbf{x_1, x_2 \geq 0 \text{ , integer}}$$

- **We can force Simplex Method to avoid using x_2=3.75**
- The **optimal integer solution** have be ***one*** **of the following**:

$$\mathbf{x_2, \leq 3}$$

$$\mathbf{x_2 \geq 4}$$

Branch and Bound + Simplex



$$\max(f(x_1,x_2) = 5x_1 + 8x_2)$$

$$\text{s.t.} \quad x_1 + x_2 \leq 6$$

$$5x_1 + 9x_2 \leq 45$$

$$\mathbf{x_1, x_2 \geq 0 \text{ , integer}}$$

- **We can force Simplex Method to avoid using x_2=3.75**
- The **optimal integer solution** have be ***one*** **of the following**:

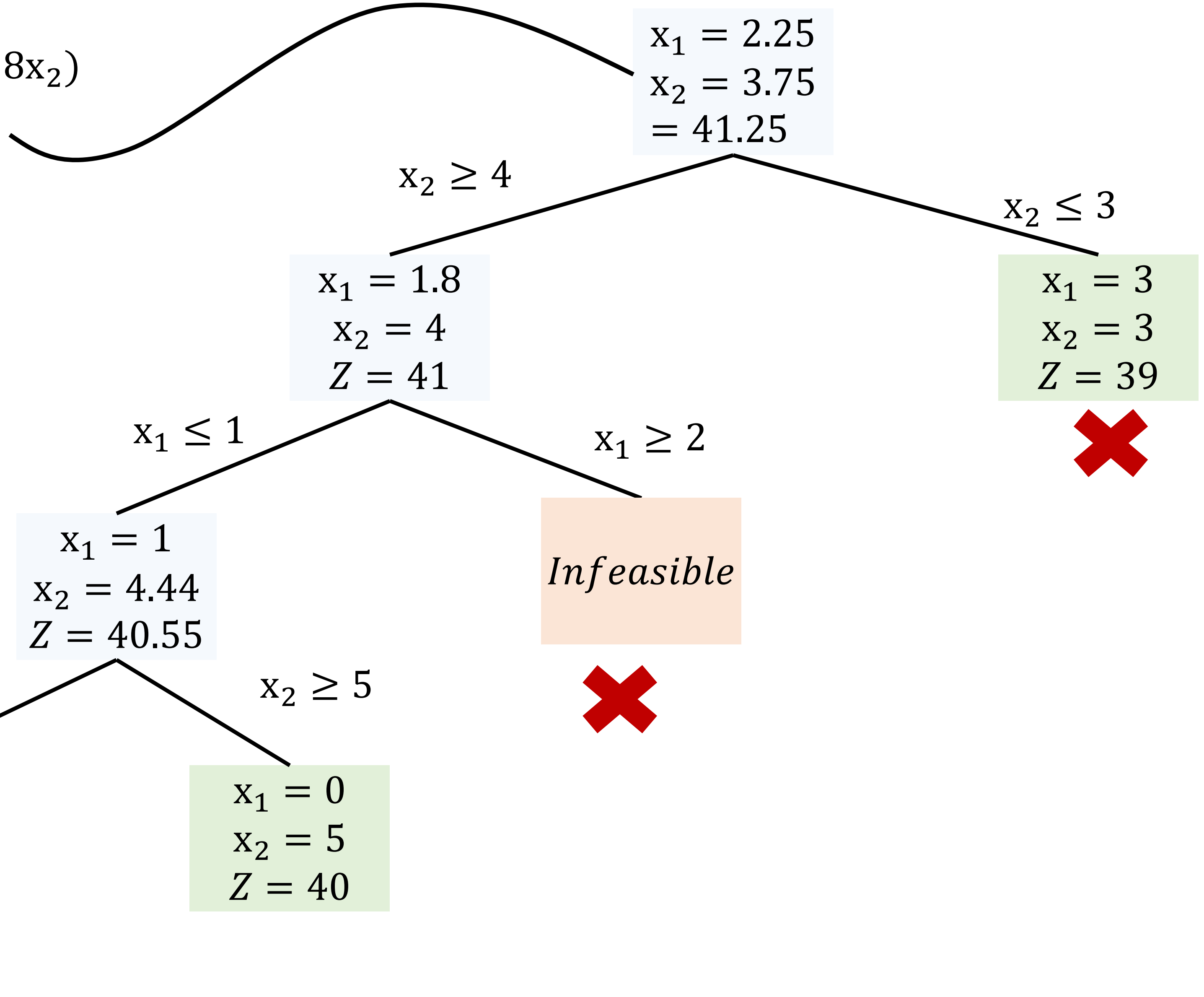
$$\mathbf{x_2, \leq 3}$$

$$\mathbf{x_2 \geq 4}$$

Branch and Bound + Simplex

LP Relaxation:

$$\max(f(x_1, x_2) = 5x_1 + 8x_2)$$
$$x_1 + x_2 \le 6$$
$$5x_1 + 9x_2 \le 45$$
$$x_1, x_2 \ge 0$$



$Upper = inf$
$Lower = -41.25$

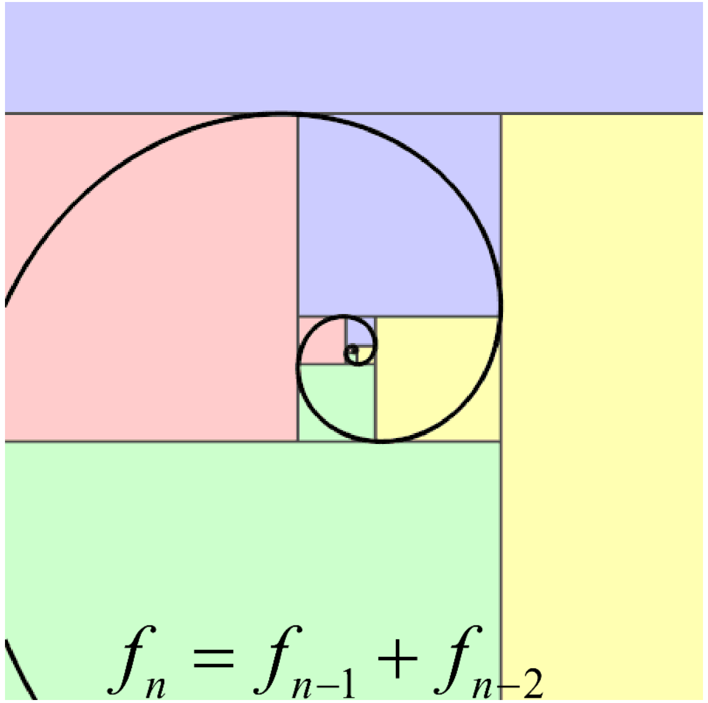
$Upper = -39$
$Lower = -41$

$Upper = -39$
$Lower = -40.55$

$Upper = -40$
$Lower = -40$

Dynamic Programming

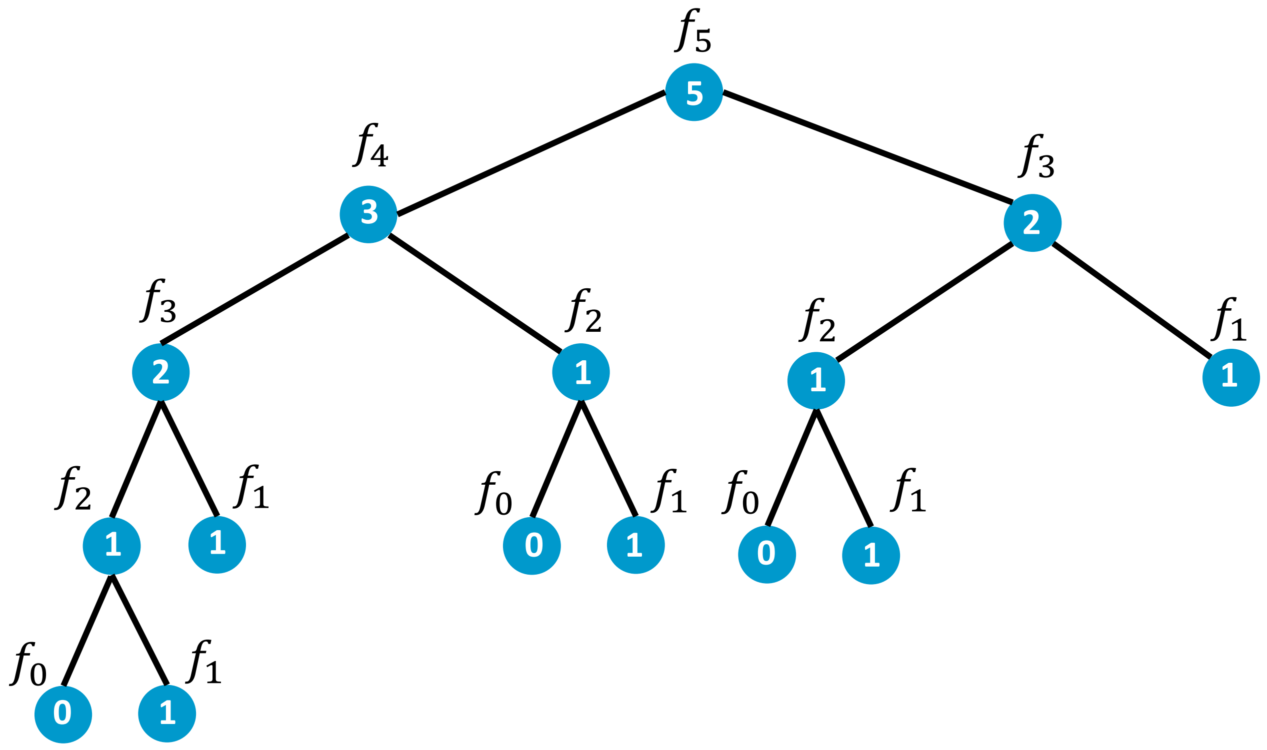
- Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.
- DP relies on memoization (not memorization!) by storing past results and reusing it so as to not repeat expensive computation
- Let us compute Fibonacci of 5 using DP



https://stemettes.org/zine/articles/fibonacci-in-nature/

Example: Computing Fibonacci Sequence

- In mathematics, the Fibonacci numbers, commonly denoted f_n, form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1.



$$f_n = f_{n-1} + f_{n-2}$$

f_0	f_1	f_2	f_3	f_4	f_5
0	1	1	2	3	5

Non-Linear Optimization

- A **non-linear program** can have a linear or nonlinear objective function with linear and/or nonlinear constraints

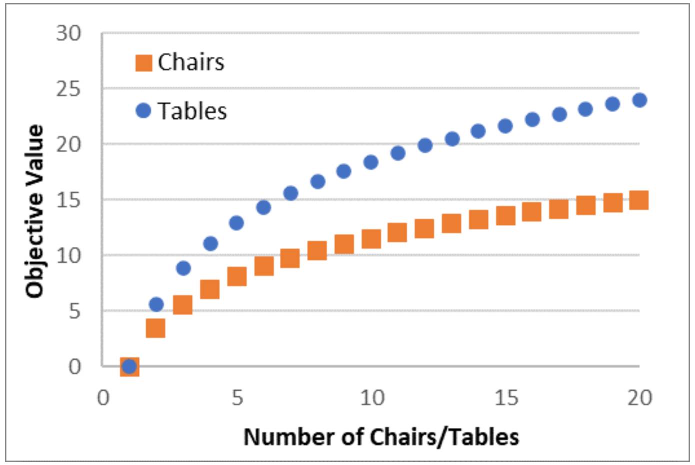
Optimize: $\max(f(x_1, x_2) = 5\ln(x_1) + 8\ln(x_2))$

Constraints: $x_1 + x_2 \leq 6$

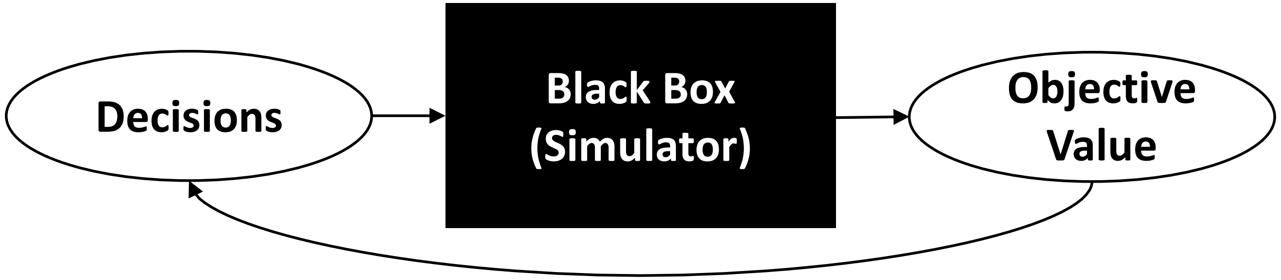
$5x_1 + 9x_2 \leq 45$

$x_1, x_2 \geq 0, \text{integer}$

Example: Non-linear behaviour due to economies of scale



- **"Black Box" optimization** refers to a problem setup in which an optimization algorithm is supposed to optimize (e.g., minimize) an objective function through a so-called black-box interface (simulation ; machine learning ; etc.)



Optimization Software

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN | Engineering Systems and Design

Optimization Software

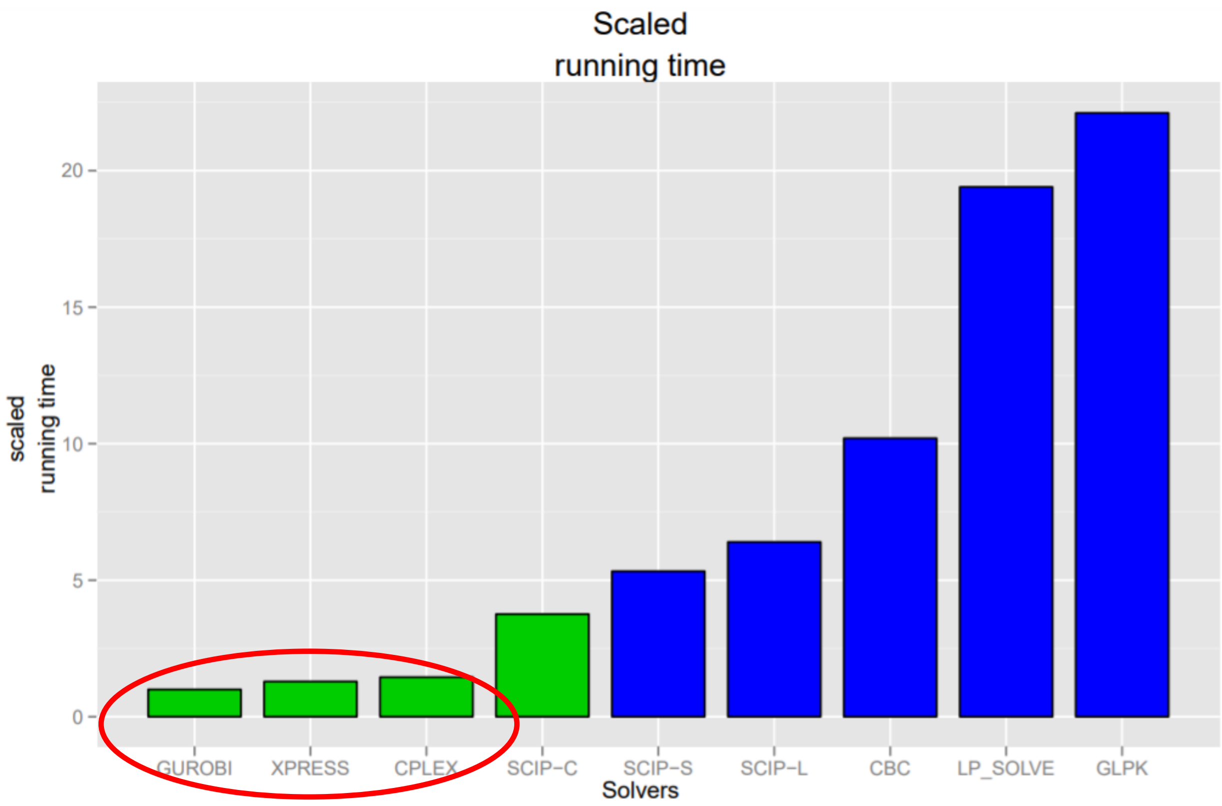
Optimization Solver

- Gurobi
- Cplex
- Xpress

Paid Software
Free Academic License

- GLPK
- LP_SOLVE
- CLP
- SCIP
- SoPlex

Free Software



Running times of several solvers applied to benchmark test data

Source: Analysis of commercial and free and open source solvers for linear optimization problems B. Meindl and M. Templ

Optimization Software

Optimization Solver	Programming Language	
Gurobi	Gurobi	Paid Software Free Academic License
Cplex	IBM CPLEX	
Xpress	Mosel	
GLPK	GAMS	Paid Software
LP_SOLVE	AMPL	
CLP	AIMSS	
SCIP	**Pyomo - Python**	Free Software
SoPlex	Google OR Tools – Python, C++, Java	
	NEOS*	Online Server

*https://neos-server.org/neos/

Pyomo Overview

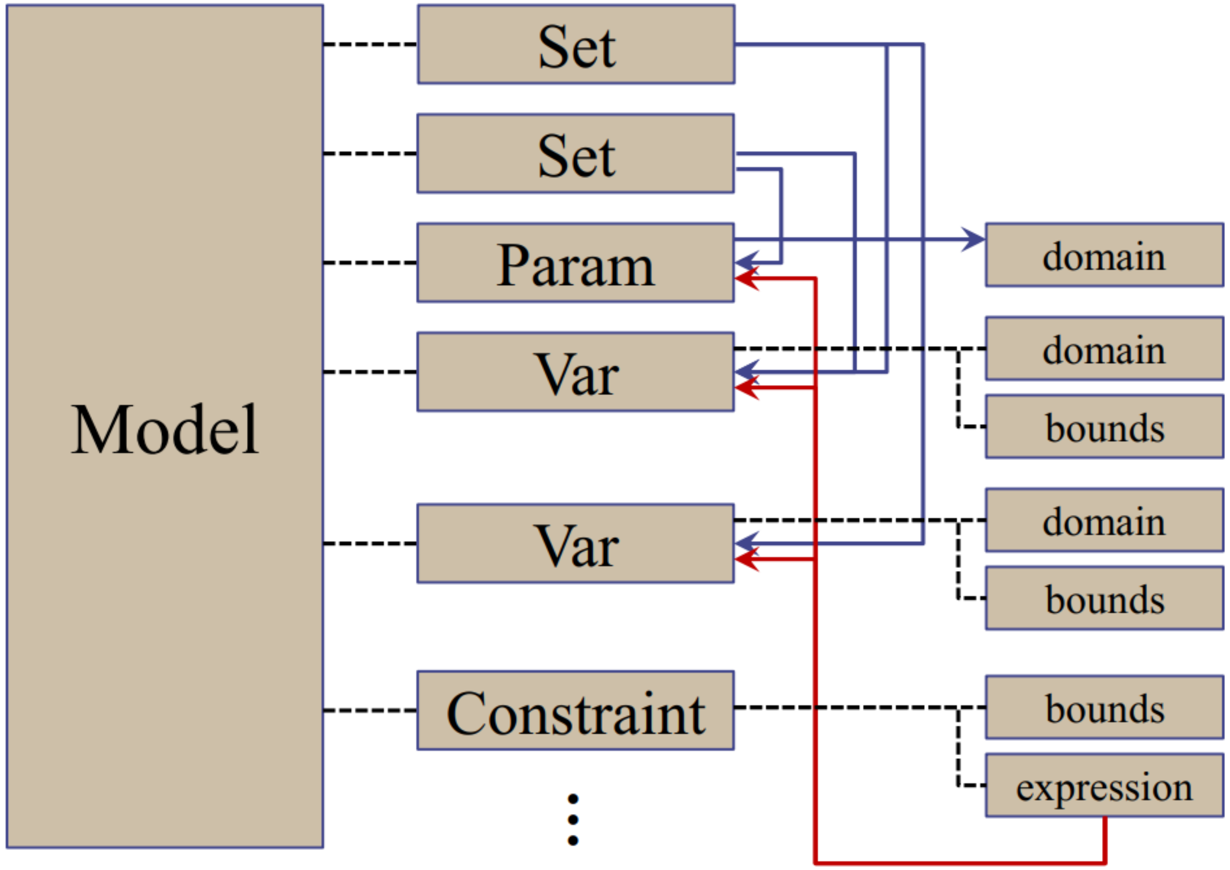
- Pythonic framework for formulating optimization models
  - Provide a natural syntax to describe mathematical models
  - Formulate large models with a concise syntax
  - Separate modeling and data declarations
  - Enable data import and export in commonly used formats

- Pyomo Documentation:
  - Pyomo Documentation 6.1.2

```
# simple.py
from pyomo.environ import *

M = ConcreteModel()
M.x1 = Var()
M.x2 = Var(bounds=(-1,1))
M.x3 = Var(bounds=(1,2))
M.o  = Objective(
         expr=M.x1**2 + (M.x2*M.x3)**4 + \
              M.x1*M.x3 + \
              M.x2*sin(M.x1+M.x3) + M.x2)

model = M
```

Fundamental Pyomo Components

- Pyomo is an object model for describing optimization problems

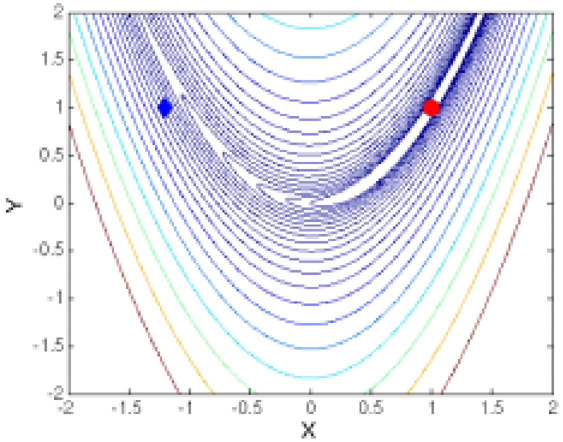


Source: https://www.ima.umn.edu/materials/2017-2018.2/W8.21-25.17/26326/3_PyomoFundamentals.pdf

A simple Pyomo Model

- Rosenbrock.py

```
from pyomo.environ import *

model = ConcreteModel()

model.x = Var( initialize=-1.2, bounds=(-2, 2) )
model.y = Var( initialize= 1.0, bounds=(-2, 2) )

model.obj = Objective(
    expr= (1-model.x)**2 + 100*(model.y-model.x**2)**2,
    sense= minimize )
```



Source: https://www.ima.umn.edu/materials/2017-2018.2/W8.21-25.17/26326/3_PyomoFundamentals.pdf

Getting Started: the Model

- Import pyomo environment and create model instance

```
from pyomo.environ import *

model = ConcreteModel()
```

Every Pyomo model starts with this; it tells Python to load the Pyomo Modeling Environment

Create an instance of a *Concrete* model
- Concrete models are immediately constructed
- Data must be present *at the time* components are defined

Local variable to hold the model we are about to construct
- While not required, by convention we use "`model`"
- If you choose to name your model something else, you will need to tell the Pyomo script the object name through the command line

Source: https://www.ima.umn.edu/materials/2017-2018.2/W8.21-25.17/26326/3_PyomoFundamentals.pdf

Modeling the Decision Variables

- Declare the decision variables

```
model.a_variable = Var(within = NonNegativeReals)
```

The name you assign the object to becomes the object's name, and must be unique in any given model.

"`within`" is optional and sets the variable domain ("`domain`" is an alias for "`within`")

Several pre-defined domains, e.g., "`Binary`"

```
model.a_variable = Var(bounds = (0, None))
```

Same as above: "`domain`" is assumed to be `Reals` if missing

Source: https://www.ima.umn.edu/materials/2017-2018.2/W8.21-25.17/26326/3_PyomoFundamentals.pdf

Modeling the Objective Function

- Formulate the objective function

```
model.x = Var( initialize=-1.2, bounds=(-2, 2) )
model.y = Var( initialize= 1.0, bounds=(-2, 2) )

model.obj = Objective(
    expr= (1-model.x)**2 + 100*(model.y-model.x**2)**2,
    sense= minimize )
```

If "sense" is omitted, Pyomo assumes minimization

Note that the Objective expression is not a *relational expression*

"expr" can be an expression, or any function-like object that returns an expression

Source: https://www.ima.umn.edu/materials/2017-2018.2/W8.21-25.17/26326/3_PyomoFundamentals.pdf

Modeling the Constraints

- Formulate the model constraints

```
model.a = Var()
model.b = Var()
model.c = Var()
model.c1 = Constraint(
    expr = model.b + 5 * model.c <= model.a )
```

> “expr” can be an expression, or any function-like object that returns an expression

Source: https://www.ima.umn.edu/materials/2017-2018.2/W8.21-25.17/26326/3_PyomoFundamentals.pdf

Modeling the Sets (Indices)

- Declare the sets of the decision variables and parameters

```
model.IDX = range(10)
model.a = Var()
model.b = Var(model.IDX)
model.c1 = Constraint(
    expr = sum(model.b[i] for i in model.IDX) <= model.a )
```

Python *list comprehensions* are very common for working over indexed variables and nicely parallel mathematical notation:

$$\sum_{i \in IDX} b_i \leq a$$

Source: https://www.ima.umn.edu/materials/2017-2018.2/W8.21-25.17/26326/3_PyomoFundamentals.pdf

Solving Optimization Problems using Pyomo

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

Facility Location Optimization Problem

- Location planning involves specifying the physical position of facilities that provide demanded services.
  - Urban and Regional Planning - location of schools, hospitals, bus stops, electric charging stations, solid waste landfills, etc.
  - Business Logistics and Supply Chains – location of industrial facilities, warehouses, distribution centres, hubs, offices, etc.
  - Defence and National Security – location of military bases, anti-missile systems, fire watchtowers, etc.
  - Electronics Industry- placement of interconnected electronic components onto a printed circuit board or on a microchip
  - Clustering techniques - cluster analysis problems can be viewed as facility location problems. The objective is to partition data points into equivalence classes such that points assigned ot the same class are close to one another
- There are a variety of different models to solve this problem (p-median problem, quadratic assignment problem, capacitated location problem, etc.)

P-Median Formulation

- **Sets**

$Set\ of\ candidate\ locations: i = 1,2,\ldots,i$

$Set\ of\ costumers: j = 1,2,\ldots,j$

- **Parameters**

$d_j = demand\ of\ costumer\ j$

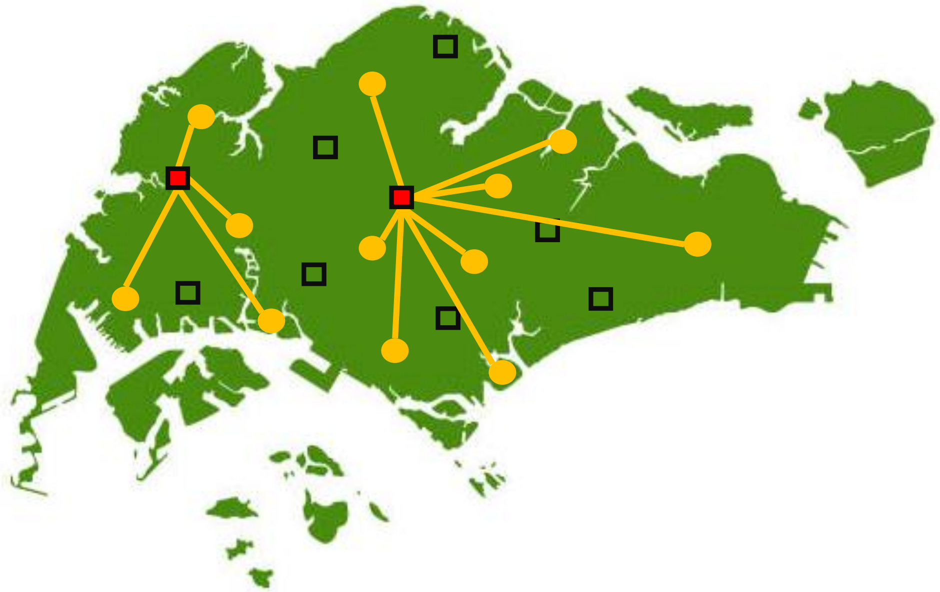
$c_{ij} = unit\ cost\ of\ satisfying\ customer\ j\ from\ facility\ i$

$p = number\ of\ locations\ to\ open$

- **Decision Variables**

$x_{ij} = 1\ if\ customer\ j\ is\ supplied\ by\ location\ i, 0\ otherwise$

$y_i = 1\ if\ a\ facility\ is\ located\ at\ location\ i, 0\ otherwise$



P-Median Formulation

$$\min z = \sum_{i \in I} \sum_{j \in J} d_j c_{ij} x_{ij}$$

Minimize the demand-weighted total cost

$$s.t. \qquad \sum_{i \in I} x_{ij} = 1\,, \qquad \forall j \in J$$

All of the demand for customer j must be satisfied

$$\sum_{i \in I} y_i = p$$

Exactly p facilities are located

$$x_{ij} \le y_i\,, \qquad \forall i \in I, j \in J$$

Demand nodes can only be assigned to open facilities

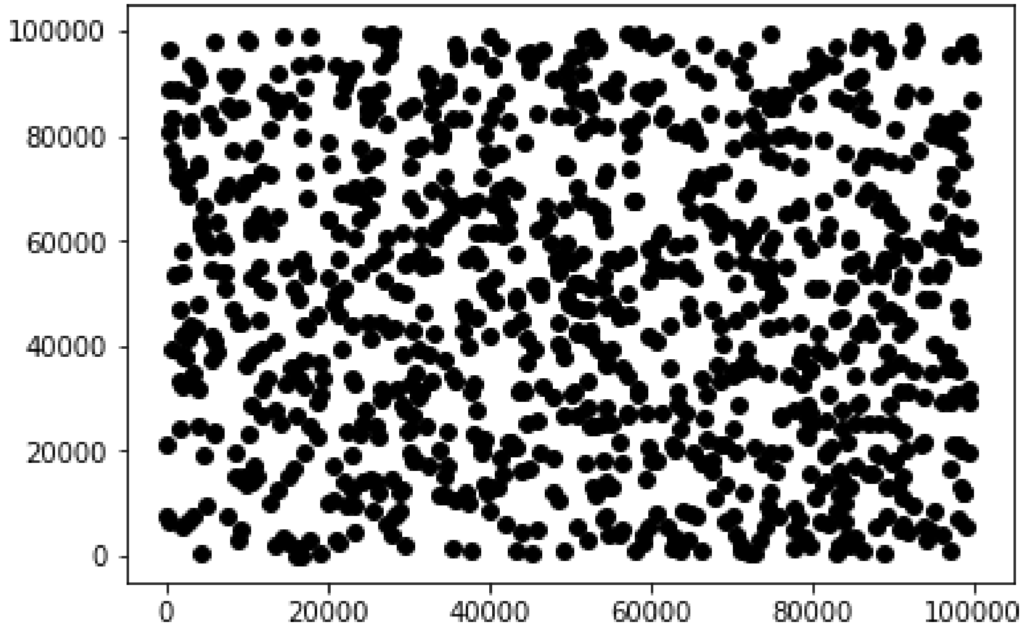
$$x_{ij}, y_i \; is\ binary$$

All variables are binary

Pyomo P-Median Formulation

Inputs

```
#Generate Data Inputs

# Select random seed
random.seed(1)

# Number of candidate locations
#n=100
n=1000

#Number of locations to open
openfac=30

#Coordinate Range
rangelct=100000

#Generate random locations
coordlct_x = random.choices(range(0, rangelct), k=n)
coordlct_y = random.choices(range(0, rangelct), k=n)

#Compute distance between locations
distancelct=np.empty([n, n])
for i_index in range(n):
    for j_index in range(n):
        distancelct[i_index,j_index]=(math.sqrt(((coordlct_x[i_index]-coordlct_x[j_index])**2) +((coordlct_y[i_index]-coordlct_y

df = pd.DataFrame(distancelct)
df.index += 1
df.columns += 1
cij_model=df.stack().to_dict()

#Generate demand between locations
demandlct = random.choices(range(1, 50), k=n)
demanddf = pd.DataFrame(demandlct)
demanddf.index += 1
dj_model=demanddf.to_dict()
dj_model=dj_model[0]
```



Pyomo P-Median Formulation

Sets

```
# Create Model
model = AbstractModel()

# Set of candidate locations
model.M = RangeSet(n)
# Set of customer nodes
model.N = RangeSet(n)
```

$$Set\ of\ candidate\ locations: i = 1,2,\ldots,i$$

$$Set\ of\ costumers: j = 1,2,\ldots,j$$

Parameters

```
# Number of facilities
model.p = openfac
# d[j] - demand of customer j
model.d = Param(model.N, initialize=dj_model)
# c[i,j] - unit cost of satisfying customer j from facility i
model.c = Param(model.M, model.N, initialize=cij_model)
```

$$p = number\ of\ locations\ to\ open$$

$$d_j = demand\ of\ costumer\ j$$

$$c_{ij} = unit\ cost\ of\ satisfying\ customer\ j\ from\ facility\ i$$

Decision Variables

```
# x[i,j] - 1 if customer j is supplied by location i
model.x = Var(model.M, model.N, within=Binary)

# y[i] - a binary value that is 1 if a facility is located at location i
model.y = Var(model.M, within=Binary)
```

$$x_{ij} = 1\ if\ customer\ j\ is\ supplied\ by\ location\ i, 0\ otherwise$$

$$y_i = 1\ if\ a\ facility\ is\ located\ at\ location\ i, 0\ otherwise$$

Pyomo P-Median Formulation

Objective Function

```
# Minimize the demand-weighted total cost
def cost_(model):
    return sum(model.d[j]*model.c[i,j]*model.x[i,j] for i in model.M for j in model.N)
model.cost = Objective(rule=cost_)
```

$$min\, z = \sum_{i \in I} \sum_{j \in J} d_j c_{ij} x_{ij}$$

Constraints

```
# All of the demand for customer j must be satisfied
def demand_(model, j):
    return sum(model.x[i,j] for i in model.M) == 1.0
model.demand = Constraint(model.N, rule=demand_)
```

$$\sum_{i \in I} x_{ij} = 1, \qquad \forall j \in J$$

```
# Exactly p facilities are located
def facilities_(model):
    return sum(model.y[i] for i in model.M) == model.p
model.facilities = Constraint(rule=facilities_)
```

$$\sum_{i \in I} y_i = p$$

```
# Demand nodes can only be assigned to open facilities
def openfac_(model, i, j):
    return model.x[i,j] <= model.y[i]
model.openfac = Constraint(model.M, model.N, rule=openfac_)
```

$$x_{ij} \le y_i, \qquad \forall i \in I, j \in J$$

Pyomo P-Median Solve Model

Solve Model

```
instance = model.create_instance()
opt = pyo.SolverFactory('gurobi')
opt.solve(instance, options={'TimeLimit': 10000},tee=True)
```

```
Solved with barrier

Root relaxation: objective 1.613801e+08, 30112 iterations, 157.47 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 1.6138e+08    0  538 2.3800e+08 1.6138e+08  32.2%     -  180s
H    0     0                    1.615296e+08 1.6138e+08  0.09%     -  182s
H    0     0                    1.613936e+08 1.6138e+08  0.01%     -  188s

Explored 1 nodes (30112 simplex iterations) in 188.72 seconds
Thread count was 4 (of 4 available processors)

Solution count 3: 1.61394e+08 1.6153e+08 2.37999e+08

Optimal solution found (tolerance 1.00e-04)
Best objective 1.613935998492e+08, best bound 1.613800660278e+08, gap 0.0084%
```

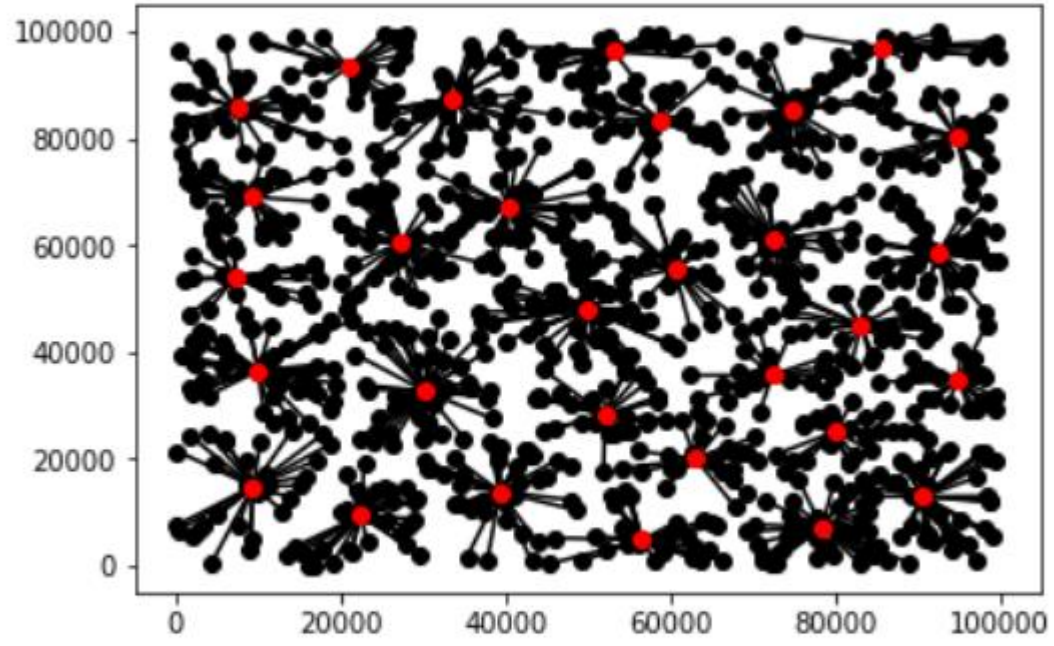
Pyomo P-Median Outputs

Read output variables

```
xij_dic = {i:int(value(j)) for (i,j) in instance.x.items()}
xij = pd.DataFrame(xij_dic.values(), index = xij_dic.keys()).unstack()
linkindex = np.where(xij == 1)
```

Plot results

```
def connectpoints(x,y,p1,p2):
    x1, x2 = x[p1], x[p2]
    y1, y2 = y[p1], y[p2]
    plt.plot([x1,x2],[y1,y2],'k-')

for i_index in range(len(linkindex[0])):
    connectpoints(coordlct_x,coordlct_y,linkindex[0][i_index],linkindex[1][i_index])

plt.plot(coordlct_x, coordlct_y, 'o', color='black');

yi_dic = {i:int(value(j)) for (i,j) in instance.y.items()}
yi = pd.DataFrame(yi_dic.values(), index = yi_dic.keys()).unstack()
facilityindex = np.where(yi == 1)

for i_index in range(len(facilityindex[0])):
    plt.plot(coordlct_x[facilityindex[0][i_index]], coordlct_y[facilityindex[0][i_index]], 'o', color='red');
```



Travelling Salesman Problem

- The Traveling Salesman Problem is one of the most intensively studied problems in computational mathematics.
- Given a collection of cities and the cost of travel between each pair of them, the traveling salesman problem, or TSP for short, is to find the cheapest way of visiting all of the cities and returning to your starting point.

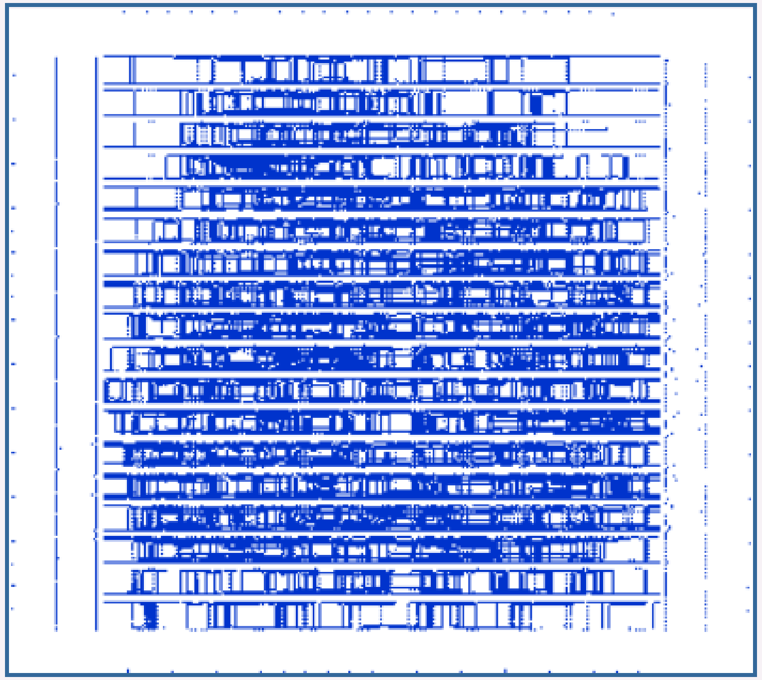
Travelling Salesman Problem

- Much of the work on the TSP is motivated by its use as a platform for the study of general methods that can be applied to a wide range of discrete optimization problems. This is not to say, however, that the TSP does not find applications in many fields.
- The TSP naturally arises as a **subproblem in many transportation and logistics applications**, for example the problem of arranging school bus routes to pick up the children in a school district.
- Other applications: scheduling of service calls at cable firms, the delivery of meals to homebound persons, the scheduling of stacker cranes in warehouses, the routing of trucks for parcel post pickup
- Also: genome sequencing, NASA Starlight space interferometer program, semi-conductor manufacturing, compute DNA sequences, fiber optical networks, deliver power to electronic devices
- **https://www.math.uwaterloo.ca/tsp/apps/index.html**

Solving the Travelling Salesman Problem

24,978 Cities in Sweden
Solved in 2004

15,112 Cities in Germany
Solved in 2001

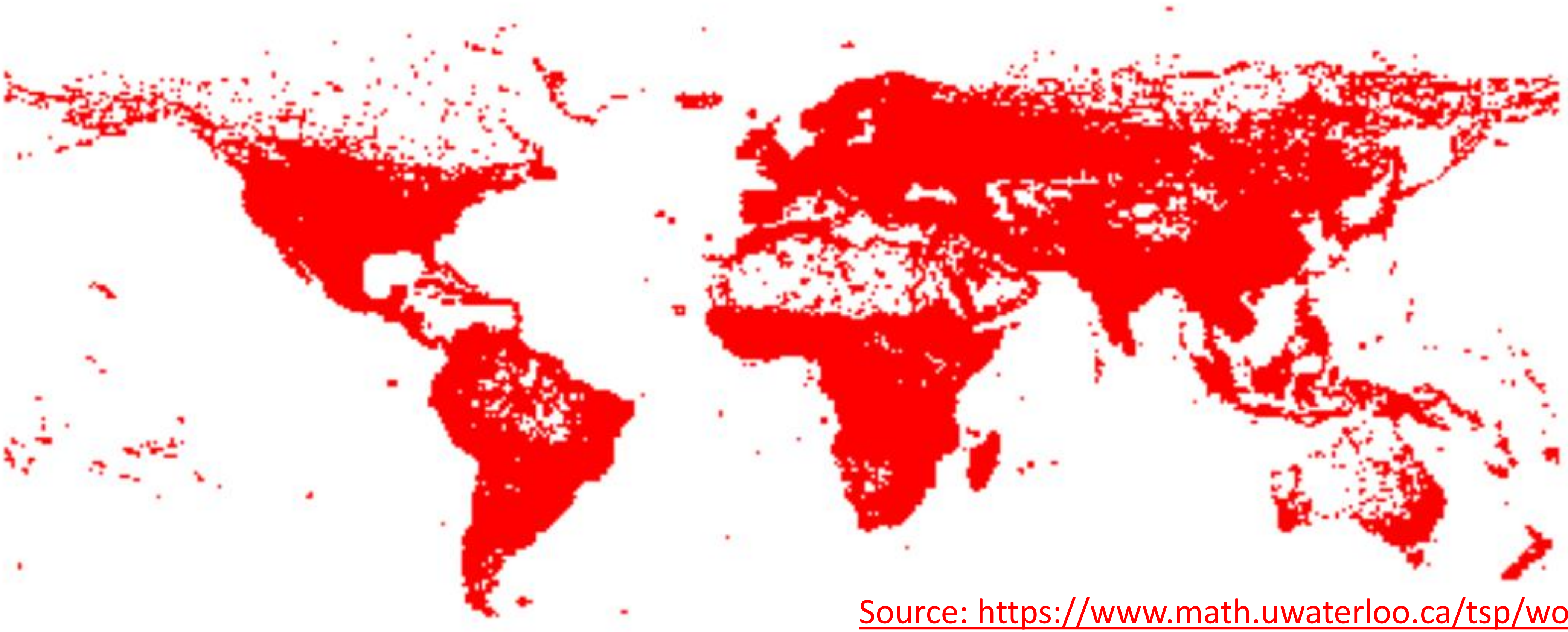


85,900 Locations in a VLSI Application
Solved in 2006

1954	1962	1977	1987	1987	1987	1994	1998	2001	2004
n=49	n=33	n=120	n=532	n=666	n=2392	n=7397	n=13509	n=15112	n=24978

https://www.math.uwaterloo.ca/tsp/apps/index.html

Solving the Travelling Salesman Problem



Source: https://www.math.uwaterloo.ca/tsp/world/

- Size: 1,904,711-cities
- Best lower bound: 7,512,218,268 (June, 2007)
- Best solution: 7,515,755,956 (February, 2021) --- Gap 0.0471%

TSP Formulation

- **Sets**

$Set\ of\ cities\ to\ visit: i = 1,2,\ldots,i$

- **Parameters**

$c_{ij} = distance\ from\ i\ and\ j$

- **Decision Variables**

$x_{ij} = 1\ if\ city\ j\ is\ visited\ right\ after\ city\ i, 0\ otherwise$

$u_i = Auxiliary\ variable\ indicating\ tour\ ordering$

TSP Formulation

$$min\, z = \sum_{i,j \in I} c_{ij} x_{ij}$$

Minimize the total distance

$$s.t. \qquad \sum_{j \in I} x_{ij} = 1\,, \qquad \forall i \in I$$

There is only one departure from each city

$$\sum_{i \in I} x_{ij} = 1\,, \qquad \forall j \in I$$

There is only one arrival to each city

$$u_i - u_j + n x_{ij} \leq (n-1)\,, \qquad \forall i, j \in I \mid 2 \leq i \neq j \leq n$$

$$n = card(J)$$

$$x_{ij}, is\ binary$$

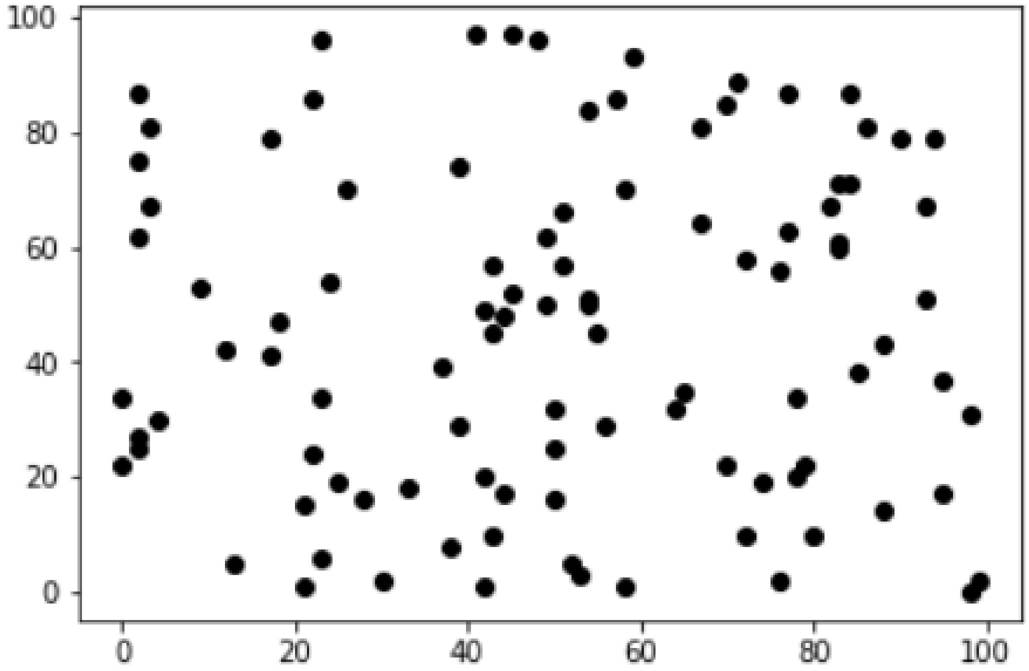
$$0 \leq u_i \leq n, \qquad \forall i \in I$$

There is only a single tour covering all cities, and not two or more disjointed tours

Pyomo TSP Formulation

Inputs

```
#Generate Data Inputs

# Select random seed
random.seed(1)

# Number of cities
n=100

#Coordinate Range
rangelct=100

#Generate random locations
coordlct_x = random.choices(range(0, rangelct), k=n)
coordlct_y = random.choices(range(0, rangelct), k=n)

#Compute distance between locations
distancelct=np.empty([n, n])
for i_index in range(n):
    for j_index in range(n):
        distancelct[i_index,j_index]=(math.sqrt(((coordlct_x[i_index]-coordlct_x[j_index])**2) +((coordlct_y[i_index]-coordlct_y

distancelct[np.diag_indices_from(distancelct)] = 99999

df = pd.DataFrame(distancelct)
df.index += 1
df.columns += 1
cij_model=df.stack().to_dict()
```



Pyomo TSP Formulation

Sets

```
# Create Model
model = AbstractModel()

# Set of cities to visit
model.N = RangeSet(n)
model.U= RangeSet(2,n)
```

$Set\ of\ cities\ to\ visit: i = 1,2, \dots, i$

$Set\ of\ cities\ to\ visit\ \ w/o\ \ considering\ origin: i = 2,3, \dots, i$

Parameters

```
# c[i,j] - distance from i and j
model.c = Param(model.N, model.N, initialize=cij_model)
```

$c_{ij} = distance\ from\ i\ and\ j$

Decision Variables

```
# x[i,j] - 1 if city j is visited right after city i, 0 otherwise
model.x = Var(model.N, model.N, within=Binary)

# u[i] - auxiliary variable indicating tour ordering
model.u = Var(model.N, within=Integers,bounds=(0,n))
```

$x_{ij} = 1\ if\ city\ j\ is\ visited\ right\ after\ city\ i, 0\ otherwise$

$u_i = Auxiliary\ variable\ indicating\ tour\ ordering$

Pyomo TSP Formulation

Objective Function

```
# Minimize total distance
def cost_(model):
    return sum(model.c[i,j]*model.x[i,j] for i in model.N for j in model.N)
model.cost = Objective(rule=cost_)
```

$$min\, z = \sum_{i,j \in I} c_{ij} x_{ij}$$

Constraints

```
#  There is only one departure from each city
def arrive_(model, j):
    return sum(model.x[i,j] for i in model.N if i!=j ) == 1
model.arrive = Constraint(model.N, rule=arrive_)
```

$$\sum_{i \in I} x_{ij} = 1\,, \qquad \forall j \in I$$

```
# There is only one arrival to each city
def depart_(model, i):
    return sum(model.x[i,j] for j in model.N if j!=i ) == 1
model.depart = Constraint(model.N, rule=depart_)
```

$$\sum_{j \in I} x_{ij} = 1\,, \qquad \forall i \in I$$

```
#  There is only a single tour covering all cities, and not two or more disjointed tours
def singletour_(model,i,j):
    if i!=j:
        return model.u[i] - model.u[j] + model.x[i,j] * n <= n-1
    else:
        return model.u[i] - model.u[j] == 0

model.singletour = Constraint(model.U,model.N,rule=singletour_)
```

$$u_i - u_j + n x_{ij} \leq (n-1)$$

$$,\forall i,j \in I \mid 2 \leq i \neq j \leq n$$

Pyomo TSP Formulation

```
instance = model.create_instance()
opt = pyo.SolverFactory('gurobi')
opt.solve(instance, options={'TimeLimit': 3600*500},tee=True)
```

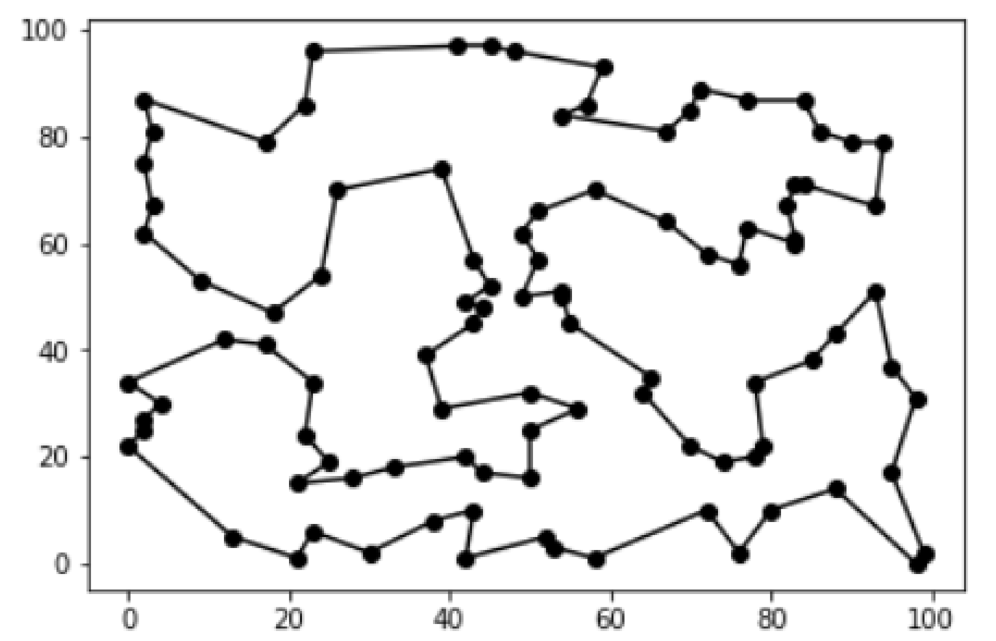
```
Root relaxation: objective 6.532166e+02, 355 iterations, 0.02 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  653.21664    0  195 4921.99004  653.21664  86.7%     -    0s
     0     0  743.92129    0  254 4921.99004  743.92129  84.9%     -    1s
     0     0  743.95035    0  271 4921.99004  743.95035  84.9%     -    1s
     0     0  743.95035    0  273 4921.99004  743.95035  84.9%     -    1s
     0     0  751.90069    0  243 4921.99004  751.90069  84.7%     -    2s
     0     0  752.86330    0  233 4921.99004  752.86330  84.7%     -    2s
     0     0  752.89659    0  240 4921.99004  752.89659  84.7%     -    2s
     0     0  752.89659    0  240 4921.99004  752.89659  84.7%     -    2s


   969   603  809.00378  200  244  809.00378  775.49354  4.14%  31.4  110s
H  975   576                    794.3991143  775.77367  2.34%  31.3  112s
   998   592  794.39911  248  118  794.39911  775.86216  2.33%  39.9  115s
H 1077   604                    792.4346674  776.39951  2.02%  44.2  119s
  1280   648  789.86207   74  187  792.43467  776.59202  2.00%  42.8  120s
  2661   443     cutoff   80       792.43467  779.77907  1.60%  48.2  125s
* 4316   300              76     786.7827327  784.41635  0.30%  47.4  129s
  4611   272  786.45151   88   62  786.78273  784.70306  0.26%  46.0  130s
* 4716   230              72     785.9523981  784.75775  0.15%  45.6  130s
* 4849    86              81     785.5122306  785.15006  0.05%  45.1  130s
```

Pyomo TSP Outputs

Read output variables

```
xij_dic = {i:int(np.round(value(j))) for (i,j) in instance.x.items()}
xij = pd.DataFrame(xij_dic.values(), index = xij_dic.keys()).unstack()
linkindex = np.where(xij == 1)
```

```
ui_dic = {i:int(value(j)) for (i,j) in instance.u.items()}
ui = pd.DataFrame(ui_dic.values(), index = ui_dic.keys()).unstack()
sequence = np.where(ui == 1)
uidf=pd.DataFrame(ui)
uidf=uidf.sort_values(by=[0])

pd.set_option('display.max_rows', uidf.shape[0]+1)
```



Vehicle Routing Problem

- In the Vehicle Routing Problem (VRP), the goal is to find optimal routes for multiple vehicles visiting a set of locations. (When there's only one vehicle, it reduces to the Travelling Salesman Problem)

TSP Formulation

- **Sets**

$Set\ of\ cities\ to\ visit: i = 1,2, \ldots, i$

- **Parameters**

$c_{ij} = distance\ from\ i\ and\ j$

$d_i = demand\ of\ customer\ in\ city\ i$

$nvhc = number\ of\ vehicles$

$cvhc = vehicle\ capacity$

- **Decision Variables**

$x_{ij} = 1\ if\ city\ j\ is\ visited\ right\ after\ city\ i, 0\ otherwise$

$u_i = Auxiliary\ variable\ indicating\ tour\ ordering$

VRP Formulation

$$\min z = \sum_{i,j \in I} c_{ij} x_{ij}$$

Minimize the total distance

$$s.t. \quad \sum_{j \in I} c_{ij} x_{ij} = 1\,, \qquad \forall i \in I$$

Each city there is a departure to exactly one other city

$$\sum_{i \in I} c_{ij} x_{ij} = 1\,, \qquad \forall j \in I$$

Each city is arrived at from exactly one other city

$$\sum_{j \in I} x_{nj} = nvhc$$

Only nvhc vehicles leave the depot (last city)

$$\sum_{i \in I} x_{in} = nvhc$$

Only nvhc vehicles return to the depot (last city)

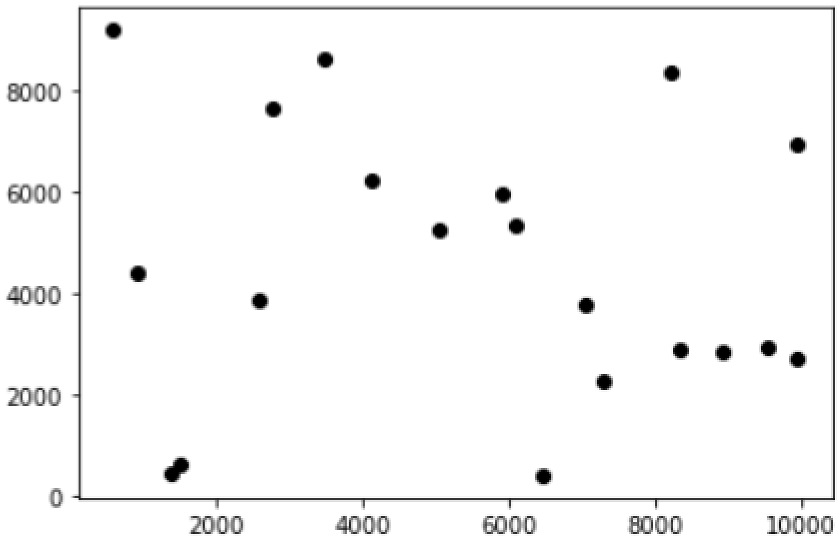
$$u_i - u_j + n x_{ij} \leq (n-1)\, y_i\,, \qquad \forall i,j \in I \mid 1 \leq i \neq j \leq n-1$$

All tours start and end in the depot (i.e. no subtours)

$$n = card(J) \qquad x_{ij}, is\ binary \qquad 0 \leq u_i \leq n, \qquad \forall, ji \in I$$

Pyomo VRP Formulation

```
#Generate Data Inputs

# Number of cities
n=20

# Number of Vehicles
nvhc=3

# Vehicle capacity
cvhc=230

#Coordinate Range
rangelct=10000

coordlct_x = random.sample(range(0, rangelct), n)

coordlct_y = random.sample(range(0, rangelct), n)

distancelct=np.empty([n, n])
for i_index in range(n):
    for j_index in range(n):
        distancelct[i_index,j_index]=(math.sqrt(((coordlct_x[i_index]-coordlct_x[j_index])**2) +((coordlct_y[i_index]-coordlct_y

distancelct[np.diag_indices_from(distancelct)] = 99999

distancelctdf = pd.DataFrame(distancelct)
distancelctdf.index += 1
distancelctdf.columns += 1
cij_model=distancelctdf.stack().to_dict()

cij_model

#Generate Demand
demandlct = random.sample(range(1, 50), n)
demanddf = pd.DataFrame(demandlct)
demanddf.index += 1
dj_model=demanddf.to_dict()
dj_model=dj_model[0]
```



Pyomo VRP Formulation

Sets

```
model = AbstractModel()

# Set of candidate cities
model.N = RangeSet(n)
model.M = RangeSet(n-1)
```

Parameters

```
# c[i,j] - distance from i and j
model.c = Param(model.N, model.N, initialize=cij_model)

# d[j] - demand of customer j
model.d = Param(model.N, initialize=dj_model)
```

Decision Variables

```
# x[i,j] - 1 if city j is visited right after city i, 0 otherwise
model.x = Var(model.N, model.N, within=Binary)

# u[i] - auxiliary variable indicating tour ordering
model.u = Var(model.N, within=NonNegativeReals)
```

Pyomo VRP Formulation

Objective Function

```
# Minimize the demand-weighted total cost
def cost_(model):
    return sum(model.c[i,j]*model.x[i,j] for i in model.N for j in model.N)
model.cost = Objective(rule=cost_)
```

Constraints

```
#  Only 1 departs from each city
def departs_(model, j):
    return sum(model.x[i,j] for i in model.N if i!=j ) == 1
model.departs = Constraint(model.M, rule=departs_)
```

```
#  Only 1 arrives from each city
def arrives_(model, i):
    return sum(model.x[i,j] for j in model.N if j!=i ) == 1
model.arrives = Constraint(model.M, rule=arrives_)
```

```
#  Only nvhc vehicles arrive to the depot  (city with index n)
def arrivesdepot_(model):
    return sum(model.x[i,n] for i in model.N) == nvhc
model.departsdepot = Constraint(rule=arrivesdepot_)
```

```
#  Only nvhc vehicles depart to the depot (city with index n)
def departdepot_(model):
    return sum(model.x[n,j] for j in model.N) == nvhc
model.arrivesdepot = Constraint(rule=departdepot_)
```

```
def singletour_(model,i,j):
    if i!=j:
        return model.u[i] - model.u[j] +cvhc*model.x[i,j] <= cvhc-model.d[i]
    else:
        return model.u[n]== 0

model.singletour = Constraint(model.M,model.M,rule=singletour_)
```

Pyomo VRP Formulation

Solve Model

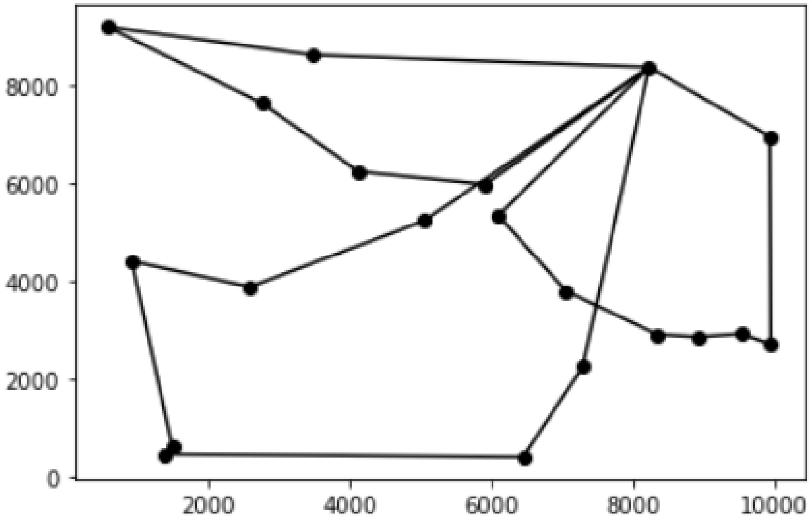
```
instance = model.create_instance()
opt = pyo.SolverFactory('gurobi')
opt.solve(instance, options={'TimeLimit': 10000},tee=True)
```

```
 1898020 119535      cutoff   48       58996.2502 57755.5710  2.10%  10.2  550s
 1918105 114070 58376.2388   53   24 58996.2502 57796.2967  2.03%  10.2  555s
 1936846 108570      cutoff   60       58996.2502 57835.7144  1.97%  10.2  560s
 1955533 102850 58968.5048   52   22 58996.2502 57877.0488  1.90%  10.1  565s
 1976102 96126      cutoff   50       58996.2502 57926.6680  1.81%  10.1  570s
 1993836 89929 58775.7200   57   23 58996.2502 57970.7462  1.74%  10.1  575s
 2009394 84243      cutoff   87       58996.2502 58011.0204  1.67%  10.1  580s
 2023335 78812 infeasible   60       58996.2502 58051.1434  1.60%  10.1  585s
 2036836 73501 58926.5236   49   15 58996.2502 58090.8971  1.53%  10.1  590s
 2049888 67946 58912.9060   61   12 58996.2502 58135.4420  1.46%  10.1  595s
 2063176 61976 infeasible   50       58996.2502 58181.1793  1.38%  10.1  600s
 2077785 55086      cutoff   48       58996.2502 58237.1841  1.29%  10.1  605s
 2090984 48282 infeasible   69       58996.2502 58295.4634  1.19%  10.0  610s
 2107780 38888      cutoff   51       58996.2502 58381.3247  1.04%  10.0  615s
 2123286 29412 58612.6235   58   18 58996.2502 58481.8104  0.87%  10.0  620s
 2136164 20201      cutoff   62       58996.2502 58594.4752  0.68%  10.0  625s
 2151717  7282      cutoff   67       58996.2502 58804.9760  0.32%  10.0  630s

Cutting planes:
```

Pyomo VRP Formulation

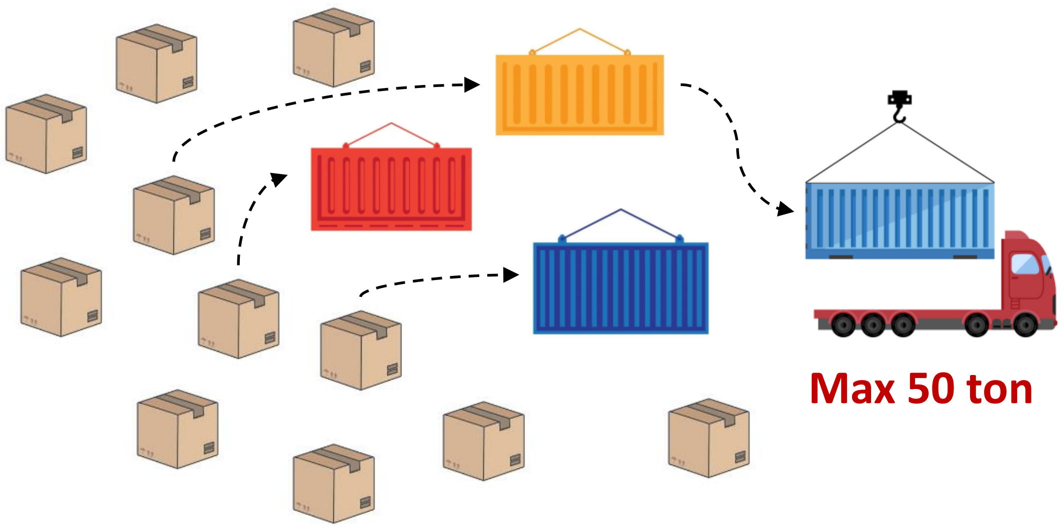
Plot results

```
def connectpoints(x,y,p1,p2):
    x1, x2 = x[p1], x[p2]
    y1, y2 = y[p1], y[p2]
    plt.plot([x1,x2],[y1,y2],'k-')

for i_index in range(len(linkindex[0])):
    connectpoints(coordlct_x,coordlct_y,linkindex[0][i_index],linkindex[1][i_index])

plt.plot(coordlct_x, coordlct_y, 'o', color='black');
```



Activity 1

- Consider the following problem: Given a set of n packages with profit p_j and weight w_j , and a set of m containers with weight capacity c_i, select m disjoint subsets of packages so that the total profit of the selected packages is maximum, while ensuring the containers' capacity is never exceeded
- Exercise 1: Formulate the problem mathematically
- Exercise 2: Solve the problem using pyomo (instances in the next slide)



Activity 1 - Instances

- Instance 1

```
random.seed(1)
n = 100 #number of objects
b= 5 #number of bins
cap=50

#Generate random locations
value = random.choices(range(10, 100), k=n)
weights = random.choices(range(5, 20), k=n)
```

- Instance 2

```
random.seed(1)
n = 10000 #number of packages
m= 200 #number of containers
cap=50

#Generate random locations
profit = random.choices(range(10, 100), k=n)
weights = random.choices(range(5, 20), k=n)
```

Happy Chinese New Year!