# Applied Artificial Intelligence Master

# Computational Tools for Data Science

## Mathematical Optimization / Prescriptive Analytics / Decision Intelligence
## Data Science / Mathematical Modeling

### Solving Optimization Problems

IPCA/EST/DTCI
João Carlos Silva 2023/2024

# Advanced Analytics Tools

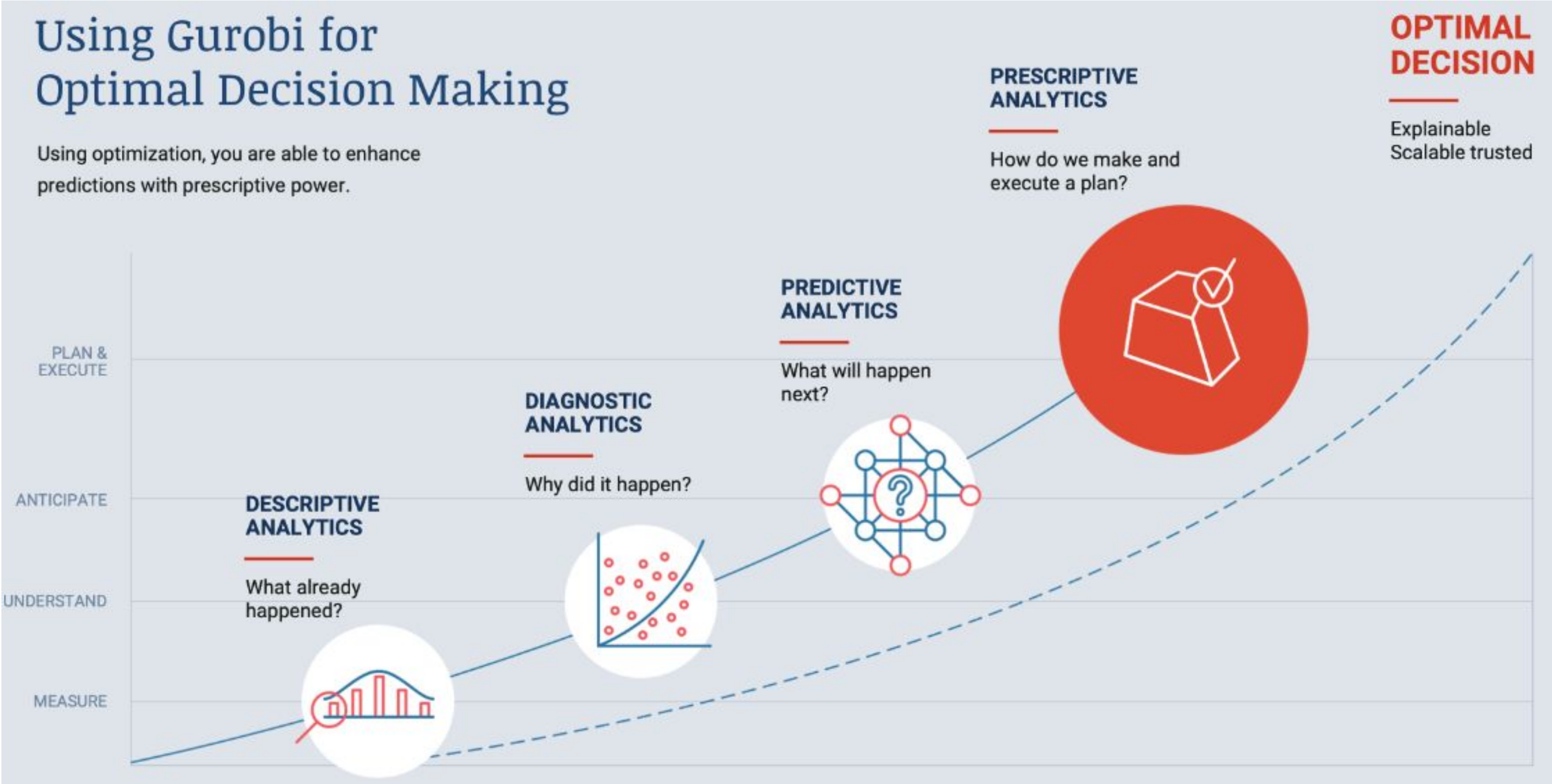
Advanced analytics tools types:

- **Descriptive** - which provide insights on what has happened in the past or is happening currently in business environment;
- **Diagnostic** - why did it happen;
- **Predictive** - which enable to predict what will happen in the future;
- **Prescriptive** - which help decide what to do in order to reach business goals.

While prescriptive analytics is growing, analytics overall is still dominated by descriptive (what happened in the past) and predictive (what is likely to happen in the future) tools.

*https://www.gurobi.com/events/prescriptive-analytics-the-data-science-master-key-to-a-turbulent-future/*

# Advanced Analytics Tools Types




https://www.gurobi.com/events/prescriptive-analytics-the-data-science-master-key-to-a-turbulent-future/

# Advanced Analytics Tools Types

Descriptive Analytics:

- Answers the Question: **What happened and why?**
- Primary Tools: Data aggregation and data mining
- Limitation: A snapshot of the past may have **limited ability to guide future decisions**
- Best Use: Summarize results for all or part of businesses

# Advanced Analytics Tools Types

Descriptive Analytics:

- Descriptive Analytics gives insight into the past and current state of business through the use of business intelligence tools. These tools can help to obtain a range of insights into business, such as:
    - How much of a given product is sold over a certain time-period
    - Current product inventory levels distribution centers
- Most business functions in company are already using descriptive analytics in the form of recurring or custom reports.

# Advanced Analytics Tools Types

Predictive Analytics:

- Answers the Question: **What might happen**?
- Primary Tools: Machine learning, Statistical models, and Simulation
- Best use: Predict what will happen in the future

# Advanced Analytics Tools Types

Predictive Analytics:

- Predictive Analytics seeks to provide insight into what the future may hold for businesses. It **takes existing data and applies statistical techniques** often using machine learning.
- Results (e.g., expected industry growth or raw material pricing), company-centric (e.g., revenue or profit growth), or operational (e.g., expected changes in demand by product line).

# Advanced Analytics Tools Types

Predictive Analytics:

Using machine learning everything isn't visible, understandable and explainable (**black box**);

Use of algorithms and statistical models by computer systems to perform a specific task without using explicit instructions;

Machine learning algorithms build a mathematical model based on sample data, known as ‘training data’, in order to make predictions or decisions without being explicitly programmed to perform the task;

The computer learns automatically, without human intervention or assistance.

# Advanced Analytics Tools Types

Predictive Analytics:

Machine learning can appear as a revolutionary approach at first;

Its lack of transparency and a large amount of data that is required in order for the system to learn are its two main flaws;

Companies now realize how important it is to have a **transparent AI** for ethical reasons;

# Advanced Analytics Tools Types

Prescriptive Analytics:

- Answers the Question: **What should we do?**
- Primary Tools: Mathematical Optimization and heuristics
- Best use: Make important, interdependent, complex decisions

# Advanced Analytics Tools Types

## Prescriptive Analytics:

- Prescriptive Analytics applies computational sciences, typically through **math programming models**, **to optimize a set of decisions** for directing a given business situation (Mathematical Optimization);
- Using a math programming solver, professionals can:
  - Explore an **astronomical number of possible combinations and options** and find **the proven best option**;
  - Apply a range of option **constraints** to **maximize or minimize objectives**;
  - **Reduce decision-making risk**;
  - **Free up time** for higher-value efforts such as performing scenario analysis or considering larger strategic questions.

# Advanced Analytics Tools Types

Prescriptive Analytics:

- For example, Prescriptive Analytics could answer these business decision questions:
  - **Which is the order to produce what products? In which manufacturing facilities? On what product lines? In what quantities?**
- Subject to a range of constraints:
  - **Minimum production of a given product, Required manufacturing time and cost of a particular machine, Raw material inventory, Finished goods inventory capacity**
- To maximize or minimize objectives
  - Total product costs
- The result enables to increase profitability and save time for businesses.

# Mathematical Optimization - Industry Use Cases

**Logistic**: Reduce costs and improve operational efficiency across workflows;

**Manufacturing**: Optimize production schedules and supply chains. Optimize quality control processes to reduce defects while minimizing inspection costs;

**Finance and Investment**: Construct portfolios that support maximization of returns while managing risk. Optimize lending decisions, balancing risk and return;

**Energy and Utilities**: Optimize the distribution of electricity or gas to minimize losses and improve reliability. Determine the most cost-effective placement of wind turbines or solar panels;

**Healthcare**: Optimize nurse and doctor scheduling problems to ensure adequate staffing while minimizing costs. Develop optimal drug formulations, balancing efficacy and cost.

# Mathematical Optimization

Mathematical optimization or mathematical programming is the selection of a **best element**, with regard to some **criterion**, from some **set of available alternatives**;

Divided into two subfields: **discrete optimization** and **continuous (linear) optimization**;

An optimization problem with discrete variables is known as a **discrete optimization**, in which an object such as an integer, permutation or graph must be found from a countable set;

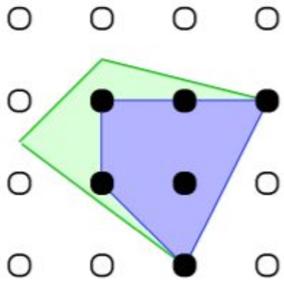
A problem with continuous variables is known as a **continuous optimization**, in which an optimal value from a continuous function must be found. They can include constrained problems and multimodal problems.

# Mathematical Optimization - 2D Geometrical Interpretation

**Solving a continuous optimization problem**: the set of constraints represents a polyhedron (green polyhedron), and the optimal solution is typically on one of the vertices of this polyhedron. The equations of the green lines are known, and therefore identifying the vertices is relatively easy.

**Solving a discrete optimization problem:** the set of feasible solutions is represented by the filled circles (purple polyhedron), and the optimal solution is achieved through branch-and-cut technique.

Discrete optimization is more difficult.



# Mathematical Optimization - Toy Example

Notation - Minimum and maximum value of a function

Consider the following notation: $\min_{x \in \mathbb{R}} \left(x^2 + 1\right)$

This denotes the minimum value of the objective function $x^2 + 1$, when choosing x from the set of real numbers.

The minimum value in this case is 1, occurring at x = 0.

# Mathematical Optimization

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} & A\mathbf{x} + \mathbf{s} = \mathbf{b}, \\ & \mathbf{s} \geq \mathbf{0}, \\ & \mathbf{x} \geq \mathbf{0}, \\ \text{and} & \mathbf{x} \in \mathbb{Z}^n, \end{array}$$

Major subfields:

- **Linear programming (LP)**, studies the case in which the objective function f is linear (f(x) = a + bx) and the constraints are specified using only linear equalities and inequalities;
- **Integer programming or Integer linear programming (ILP)** studies linear programs in which some or all variables are constrained to take on integer values (discrete optimization);
- **Quadratic programming** allows the objective function to have quadratic terms, while the feasible set must be specified with linear equalities and inequalities;
- **Mixed-integer programming or mixed-integer linear programming (MILP)** studies integer programming where some decision variables are not discrete;
- **Nonlinear programming** studies the general case in which the objective function or the constraints or both contain nonlinear parts.

# Mathematical Optimization vs Machine Learning

- Two highly sophisticated advanced analytics software technologies;
- Used in a vast array of applications;

https://www.gurobi.com/resources/4-key-differences-between-mathematical-optimization-and-machine-learning/

# Mathematical Optimization vs Machine Learning

Similarities:

- **Powerful AI problem-solving tools**;
- Run on **data and require extensive computing resources**;
- Benefited greatly from advancements over the past few decades in computing capability as well as data availability and quality, based on deep mathematics;
- Used to **solve complex business problems**.

# Mathematical Optimization vs Machine Learning

**Both Advanced analytics tools**:

- Machine learning — **predictive analytics tool** — is capable of processing massive amounts of historical "big data" to automatically identify patterns, learn from the past and make predictions about the future;
- Mathematical optimization — **prescriptive analytics tool** — leverages the latest available data, a mathematical model of business environment and an algorithm-based solver to generate solutions to most challenging business problems and empower to make the best possible business decisions;

*The output of machine learning — predictions — can be used to guide certain decisions, but machine learning isn't equipped to handle business problems that involve interconnected sets of decisions (some of which have more possible outcomes than there are atoms in the universe) like mathematical optimization can.*

# Mathematical Optimization vs Machine Learning

Applications:

- **Machine learning** — image and speech recognition, product recommendations, virtual personal assistants, fraud detection, and self-driving cars;
- **Mathematical optimization** — production planning, workforce scheduling, electric power distribution, and shipment routing;

# Mathematical Optimization vs Machine Learning

Adaptability:

- **Machine learning** — based on historical data — encountering sudden changes, machine learning predictions become less accurate. When this happens, machine learning models need to be retrained on new data;
- **Mathematical optimization** — based on the most up-to-date data — can easily adjust to changing conditions and give the visibility and agility needed to efficiently respond to disruption;

# Mathematical Optimization vs Machine Learning

**Maturity**:

- Machine Learning — according to Gartner has reached nowadays the “peak of expectations”;
- Mathematical Optimization — according to Gartner has reached the “peak of expectations” in the early 1970s. Proven technology that companies across industries have applied widely;

Both AI tools, Mathematical Optimization and Machine Learning, will have **an expanding impact on the world we live in for years to come**.

# Mathematical Optimization

Mathematical programming solvers help transform data and models into smarter business decisions.

- Users can state their business problems as **mathematical models**, then call a solver to automatically consider trillions or more possibilities and find the **best one**.
- Use a mathematical solver as a decision-making assistant, helping guide the choices of a skilled expert, or as a fully automated tool, making decisions without human intervention.
- Solvers rapidly consider large numbers of business constraints and decision variables within minutes, far exceeding the choices a human brain could consider over the course of many years.
- Solvers support companies' needs to refine the way they currently make decisions and enable them to efficiently and effectively take a wider array of factors and options into consideration than ever before. The end result in decisions that drive **better business results**.

https://www.gurobi.com/events/prescriptive-analytics-the-data-science-master-key-to-a-turbulent-future/

# Optimization Models

Optimization models define the **goals or objectives** for a system under consideration.

Optimization models are used to **analyze a wide range of scientific, business, and engineering applications**.

The **high availability of computing resources** has made the numerical analysis of optimization models commonplace.

The computational analysis of an optimization model requires the **specification of a model that is communicated to a solver software package**.

# Specifying Optimization models Through High-Level Languages

Without a high-level languages to specify optimization models, the process of writing input files, executing a solver, and extracting results from a solver is tedious and error-prone.

Mathematical solvers use many different input formats (MPS, SOL, NL low-level formats)

Application of multiple solvers to analyze a single optimization model introduces additional complexities.

# Algebraic Modeling Languages AML

AMLs are high-level languages for **describing and solving optimization problems**;

AMLs minimize the difficulties associated with analyzing optimization models enabling high-level specification of optimization problems;

AML software provides rigorous interfaces to external solver packages that are used to analyze problems;

Allows the user to interact with solver results in the context of their high-level model specification.

# AMLs-Based Proprietary Languages

AIMMS, AMPL, and GAMS implement optimization model specification languages

Intuitive and concise syntax for defining variables, constraints, and objectives.

Support specification of abstract concepts such as sparse sets, indices, and algebraic expressions which are essential when specifying large-scale, real-world problems with thousands or millions of constraints and variables.

These AMLs can represent a wide variety of optimization models.

# AML - Extending Standard Programming Language

Enables to formulate optimization models that are analyzed with solvers written in low level languages.

Support the specification of optimization models using an object-oriented design

Allow the user to leverage the flexibility of modern high-level programming languages

Link directly to high-performance optimization libraries and solvers,

**Examples:**

Pyomo (Python), FlopC++ (C++), OptimJ (Java), JuMP (Julia), Picat (Prolog/B-Prolog).

# Solving Optimization Problem

Steps:

1. Constructing a Model;
2. Determining the Optimization Problem Type;
3. Selecting the Software
   a. Software Commercial
   b. Open Source Solvers

# Constructing a Model

Modeling is the process of identifying and expressing in mathematical terms the **objective**, the **variables**, and the **constraints** of the problem:

1. An **objective** is a quantitative measure of the performance of the system that we want to minimize or maximize;
2. The **variables** or the unknowns are the components of the system for which we want to find values;
3. The **constraints** are the functions that describe the relationships among the variables and that define the allowable values for the variables.

# Determining the Optimization Problem Type

- Convex optimization:
  - Constrained
  - Unconstrained
- Nonconvex optimization:
  - Continuous optimization
  - Discrete optimization
- Optimization under uncertainty
  - Robust optimization
  - Stochastic programming

https://neos-guide.org/guide/types/

# Selecting a Software

- Selecting a software appropriate for the type of optimization problem to be solved
  - Solver software is concerned with finding a solution to a specific instance of an optimization model;
  - Modeling software is designed to help people formulate optimization models and analyze their solutions.
- Most modeling systems support a variety of solvers;
- Most popular solvers can be used with many different modeling systems

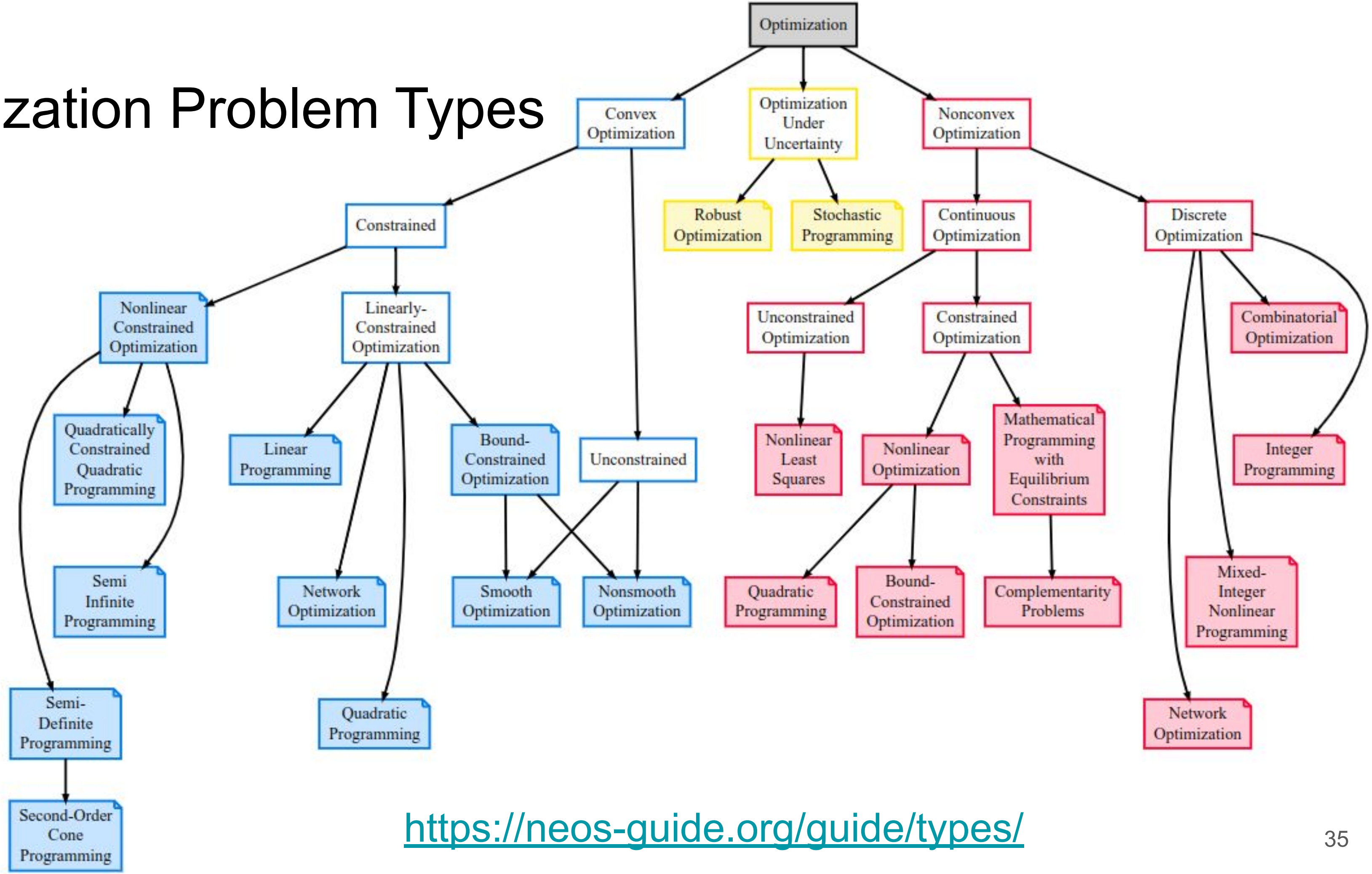
https://neos-guide.org/guide/types/

# Commercial vs. Open Source Solvers

- Commercial solvers:
  - Developed with considerable effort and, while usually more robust and reliable
  - Often quite expensive
- Open source solvers
  - Source code freely available under one of the standard open source licenses
  - Available as precompiled binaries for the more popular platforms.

https://neos-guide.org/guide/types/

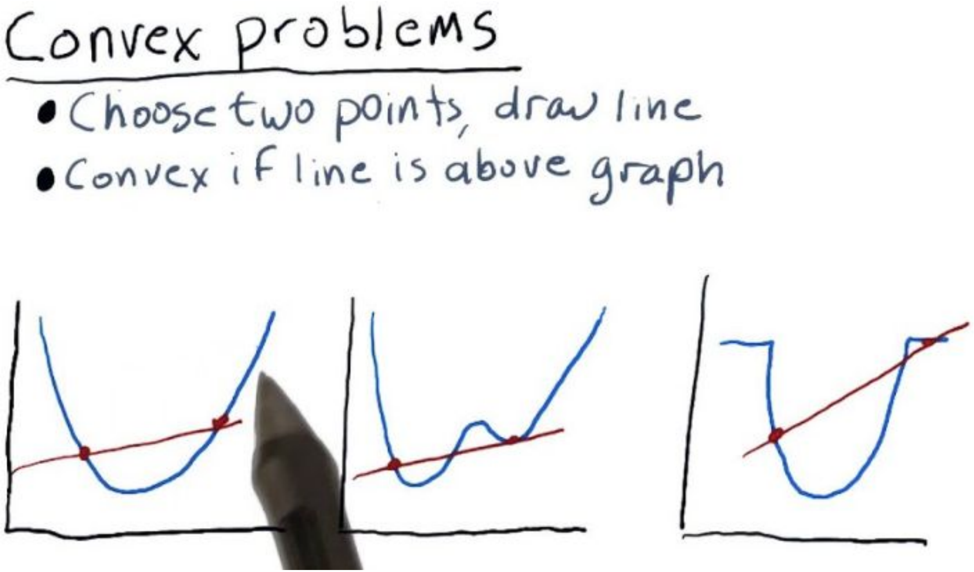
# Optimization Problem Types

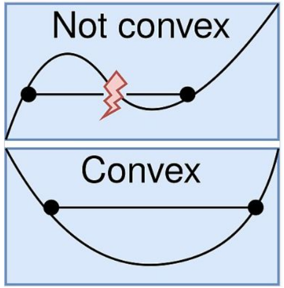


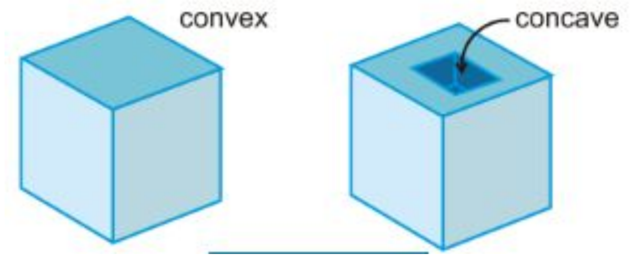

https://neos-guide.org/guide/types/

# Convex vs Non-convex Optimizations

Convex: Convex functions are important in the study of optimization problems where they are distinguished by a number of convenient properties. For instance, a convex function has no more than one minimum.



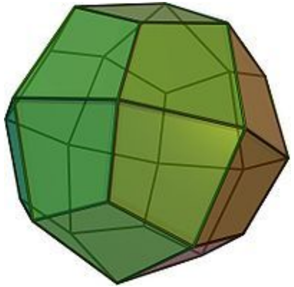
# Linear Programming

Special case of mathematical programming;

Linear programming is a technique for the **optimization of a linear objective function, subject to linear equality and linear inequality constraints**;

Its feasible region is a convex polytope;

A linear programming algorithm finds a point in the polytope where this function has the largest (or smallest) value if such a point exists.

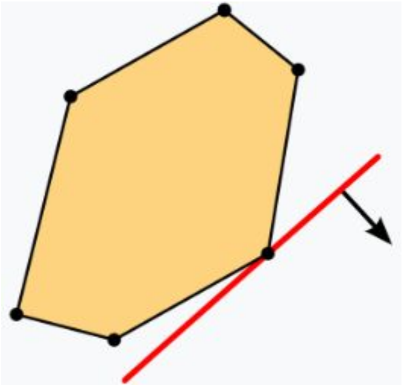


# Linear Programming

Pictorial representation of a simple linear program with **two variables** and **six inequalities**;

Set of feasible solutions depicted in yellow forms a polygon, a 2-dimensional polytope - geometric object with flat sides (faces);

The optimum of the linear cost function is where the red line intersects the polygon.



https://www.geogebra.org/3d

# Linear Programming

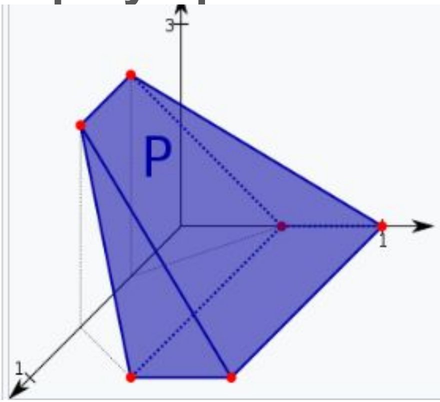
Pictorial representation of a simple linear program with **three variables** and **five inequalities**;

The surfaces giving a fixed value of the objective function are **planes** (not shown);

Set of feasible solutions depicted in purple forms a 3-dimensional polytope - geometric object with flat sides (faces);

**The linear programming problem is to find a point on the polytope that is on the plane with the highest possible value.**

Example: maximize $x + y + z$
subject to $x + y >= 1$, $x+2z < 5$
$x + y - z < 3$




https://www.geogebra.org/3d

# Quadratic Programming

Process of solving certain mathematical optimization problems involving **quadratic functions** (polynomial of degree two in one or more variables);

Type of nonlinear programming;

Degree 0 – non-zero constant[5]

Degree 1 – linear

Degree 2 – quadratic

Degree 3 – cubic

Degree 4 – quartic (or, if all terms have even degree, biquadratic)

Degree 5 – quintic

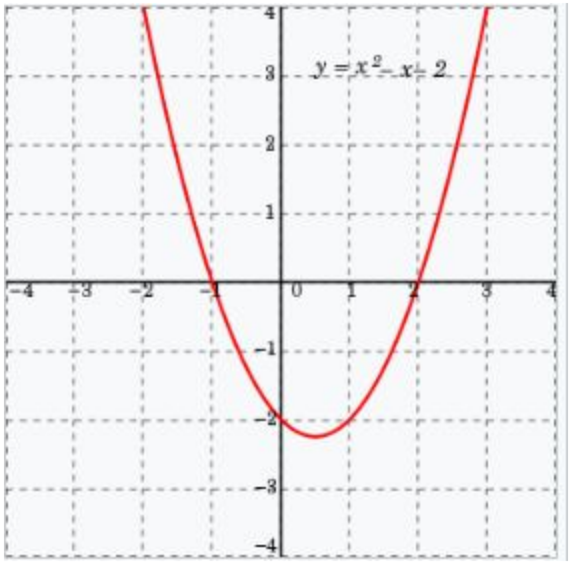
Degree 6 – sextic (or, less commonly, hexic)

Degree 7 – septic (or, less commonly, heptic)

Degree 8 – octic

Degree 9 – nonic

Degree 10 – decic



# Mixed Integer (Linear) Programming (MIP or MILP)

**Some** of the unknown variables are **required to be integers**;

Advanced algorithms for solving integer linear programs include:

- cutting-plane method
- Branch and bound
- Branch and cut
- Branch and price

# Integer (Linear) Programming (IP or ILP)

All of the unknown variables are **required to be integers**

Three types of integer programming problems: linear programs with integrality restrictions; nonlinear programs with integrality restrictions; and discrete optimisation problems;

The most popular integer programming algorithm is branch-and-bound which combines exhaustive search and bounding methods in order to explore all possible solutions efficiently.

# Gurobi Optimizer

https://www.gurobi.com/

# Gurobi Optimizer

Dr. Zonghao **Gu**, Dr. Edward **Ro**thberg, and Dr. Robert **Bi**xby founded Gurobi in 2008, coming up with the name by combining the first two initials of their last names (from CPLEX staff)

In 2023, **Air France** used Gurobi to power its decision-support tool, which recommends **optimal flight and aircraft assignments** and can take constraints like fuel consumption and an aircraft’s flying hours into account.

# Gurobi Optimizer

Mathematical optimization software library for solving the following problems types:

- Linear programming (LP);
- Quadratic programming (QP);
- Quadratically constrained programming (QCP);
- Mixed-integer linear programming (MILP);
- Mixed-integer quadratic programming (MIQP);
- Mixed-integer quadratically constrained programming (MIQCP).

https://www.gurobi.com/academia/academic-program-and-licenses/

https://pypi.org/project/gurobipy/

https://www.gurobi.com/documentation/

# Gurobi Optimizer language - A Simple Example

```
# Solve the following MIP:
#  maximize
#        x +   y + 2 z
#  subject to
#        x + 2 y + 3 z <= 4
#        x +   y       >= 1
#        x, y, z binary

import gurobipy as gp
# Create a new model
m = gp.Model()

# Create variables
x = m.addVar(vtype='B', name="x")
y = m.addVar(vtype='B', name="y")
z = m.addVar(vtype='B', name="z")

# Set objective function
m.setObjective(x + y + 2 * z, gp.GRB.MAXIMIZE)

# Add constraints
m.addConstr(x + 2 * y + 3 * z <= 4)
m.addConstr(x + y >= 1)

# Solve it!
m.optimize()
print(f"Optimal objective value: {m.objVal}")
print(f"Solution values: x={x.X}, y={y.X}, z={z.X}")
```

# NEOS Server https://neos-server.org/

Server managed by the Wisconsin Institute for Discovery at the University of Wisconsin-Madison;

Internet-based client-server application that provides free access to a library of optimization solvers;

Includes more than 60 commercial, free and open source solvers;

Solvers can be applied to mathematical optimization problems of more than 12 different types, including linear programming, integer programming and nonlinear optimization;

Most of the solvers are hosted by the University of Wisconsin in Madison. A smaller number of solvers are hosted by partner organizations: Arizona State University, the University of Klagenfurt in Austria, and the **University of Minho** in Portugal.

# NEOS Server

Solvers run on distributed, high-performance machines;

The server accepts optimization models described in modeling languages, programming languages, and problem-specific formats;

Most of the linear programming, integer programming and nonlinear programming solvers accept input from AMPL and/or GAMS;

Jobs can be submitted via a web page, email, XML RPC, Kestrel[7] or indirectly via third party submission tools SolverStudio for Excel, OpenSolver, Pyomo, JuMP (through the Julia package NEOS) and the R package rneos;

# NEOS Server

https://neos-guide.org/

https://neos-guide.org/guide/
Constructing a Model, Determining the Problem Type, Selecting Software, Commercial vs. Open Source Solvers

Solvers: https://neos-server.org/neos/solvers/

Case studies: https://neos-guide.org/case-studies/

Usage statistics: https://neos-server.org/neos/report.html

Google Analytics Reports by Month: https://neos-server.org/neos/stats/

# Modeling with Pyomo

https://www.pyomo.org
https://pyomo.readthedocs.io/en/stable/index.html
https://jckantor.github.io/ND-Pyomo-Cookbook

# Modeling with Pyomo

- Platform for specifying optimization models that embodies central ideas found in modern AMLs;
- AML that extends Python to include objects for optimization modeling;
- Used to **specify optimization models** and translate them into **various formats that can be processed by external solvers**.

# Pyomo - Modeling Language for Optimization

Modeling Languages for Optimization;

Tool for mathematical modeling: the ***Py**thon **O**ptimization **M**odeling **O**bjects* (Pyomo) software package;

Supports the formulation and analysis of mathematical models for complex optimization applications;

Pyomo implements a rich set of modeling and analysis capabilities, and provides access to these capabilities within Python, a full-featured, high-level programming language with a large set of supporting libraries.

# Pyomo Installation - Anaconda Distribution

Anaconda includes:

- A Python interpreter;
- A user interface Anaconda Navigator providing access to software development tools;
- Pre-installed versions of major python libraries;
- The conda package manager to manage python packages and environments.

# Installing a Pyomo/Python Development Environment

https://jckantor.github.io/ND-Pyomo-Cookbook/notebooks/01.01-Installing-Pyomo.html

Step 1. **Install Anaconda** (https://docs.anaconda.com/free/anaconda/install/linux/)

Step 2. Install Pyomo: **pip install pyomo**

Step 3. Install solvers:

**conda install -c conda-forge coin-or-clp**

**conda install -c conda-forge coincbc**

**conda install -c conda-forge ipopt**

**conda install -c conda-forge glpk**

Step 4. Install Gurobi (https://www.gurobi.com/)

Gurobi will be installed outside of the the default Anaconda installation. Need to specify the actual Gurobi executable. On Linux, for example, the executable is /usr/local/bin/gurobi.sh.

# Pyomo - Expressive Modeling Capability

Pyomo's modeling components can be used to express a wide range of optimization problems, including but not limited to:

linear programs, quadratic programs, nonlinear programs,

mixed-integer linear programs, mixed-integer quadratic programs,

generalized disjunctive programs, mixed-integer stochastic programs,

dynamic problems with differential algebraic equations, and

mathematical programs with equilibrium constraints.

# Pyomo - Solver Integration

Supports both tightly and loosely coupled solver interfaces.

Tightly coupled modeling tools directly access optimization solver libraries (e.g., via static or dynamic linking)

Loosely coupled modeling tools apply external optimization executables (e.g., through the use of system calls).

Many optimization solvers read problems from well-known data formats (e.g., the AMPL nl format).

# Pyomo - Getting Started

Python 3.6 or higher

Pyomo 6.0 Solvers:

- GLPK solver
- IPOPT solver
- Z3 solver
- Gurobi solver
- CPLEX solver

# Pyomo - COIN-OR Clp Linear Programming Solver

Multi-threaded open-source solver;

Written in C++;

Clp is generally a good choice for linear programs that do not include any binary or integer variables;

Generally a superior alternative to GLPK for linear programming applications;

**SolverFactory('clp').solve(model, tee=True).write()**

**display_solution(model)**

# Pyomo - COIN-OR Cbc linear programming solver

Coin-or branch and cut mixed-integer linear programming solver written in C++;

Cbc is a good choice for a general purpose MILP solver for medium to large scale problems;

Generally a superior alternative to GLPK for mixed-integer linear programming applications;

**SolverFactory('cbc').solve(model, tee=True).write()**

**display_solution(model)**

# Pyomo - COIN-OR Ipopt Nonlinear Optimization Solver

COIN-OR Ipopt is an open-source Interior Point Optimizer for large-scale nonlinear optimization;

Ipopt can solve medium to large scale nonlinear programming problems without integer or binary constraints;

**SolverFactory('ipopt').solve(model, tee=True).write()**

**display_solution(model)**

# Pyomo - COIN-OR Bonmin Nonlinear Mixed-integer Solver

COIN-OR Bonmin is a basic open-source solver for nonlinear mixed-integer programming problems (MINLP);

Utilizes CBC and Ipopt for solving relaxed subproblems;

**SolverFactory('bonmin').solve(model, tee=True).write()**

**display_solution(model)**

# Pyomo - COIN-OR Couenne Nonlinear Mixed-integer Solver

COIN-OR Couenne is attempts to find global optima for mixed-integer nonlinear programming problems (MINLP);

**SolverFactory('couenne').solve(model, tee=True).write()**

**display_solution(model)**

# Pyomo - Mathematical Concepts

**Variables**: These represent unknown or changing parts of a model (e.g., decisions to take, or the characteristic of a system outcome).

**Parameters**: These are symbolic representations for real-world data, and might vary for different problem instances or scenarios.

**Relations**: These are equations, inequalities, or other mathematical relationships defining how different parts of a model are related to each other.

# Pyomo - Optimization models

Optimization models are mathematical models with functions representing goals or objectives for the system being modeled.

Find solutions to optimize system objectives.

These models can be used for a wide range of scientific, business, and engineering applications.

# Pyomo - Linear Optimization Models

An expression in an optimization model is said to be linear if it is composed only of sums of decision variables and/or decision variables multiplied by data.

A linear expression is a non-constant, linear function of the decision variables.

Linear expressions examples:

$$\sum_{i \in \mathcal{A}} c_i x_i$$

$$\sum_{i \in \mathcal{A}} x_i$$

$$x_2$$

$$c_3 x_2 + c_2 x_3$$

$$c_3 x_2 + c_2 x_3 + 4$$

# Pyomo - Nonlinear Optimization Models

The following expressions are not linear $x_i^2$, $x_2 x_3$ and $\text{cosine}(x_2)$

Linear expressions often result in problems that can be solved with **much less computational effort** than similar models with nonlinear expressions

- Effort to use linear expressions as much as possible

- Develop linear approximations to nonlinear models in hopes of finding “good enough” solutions to the original nonlinear model.

# Solving a Pyomo Model

Pyomo provides automated methods to:

- Combine the model and data;
- Send the resulting model instance to a solver;
- Recover the results for display and further use.

# Pyomo Overview

Definition of optimization models

Object-oriented design

Basic modeling components:

- **Var** - optimization variables in a model
- **Objective** - expressions that are minimized or maximized in a model
- **Constraint** - constraint expressions in a model
- **Set** - set data that is used to define a model instance

# Pyomo - basic steps of a simple modeling process

1. Create an instance of a model using Pyomo modeling components.
2. Pass this instance to a solver to find a solution.
3. Report and analyze results from the solver.

# Pyomo - **Example1a.py** with Neos Server

```
from pyomo.environ import *

import os

# provide an email address
os.environ['NEOS_EMAIL'] = 'jcsilva@ipca.pt'

model = ConcreteModel()

# declare decision variables
model.x = Var(domain=NonNegativeReals)

# declare objective
model.profit = Objective(
    expr = 40*model.x,
    sense = maximize)

# declare constraints
model.laborA = Constraint(expr = model.x <= 80)
model.laborB = Constraint(expr = 2*model.x <= 100)
```

Linear Program: **max 40x**

**s.t. x <= 80**

**2x <= 100**

**x >= 0**

LP can be easily expressed in Pyomo/Neos:

**pyomo solve --solver=cplex --solver-manager=neos example1.py**

**pyomo solve --solver=cbc --solver-manager=neos example1.py**

# Pyomo - Simple Example with Neos - Output

```
jcsilva@asusux:~/Pyomo/Examples$ pyomo solve --solver=cplex --solver-manager=neos teste.p
[    0.00] Setting up Pyomo environment
[    0.00] Applying Pyomo preprocessing actions
[    0.00] Creating model
[    0.00] Applying solver
[    3.78] Processing results
    Number of solutions: 1
    Solution Information
      Gap: None
      Status: optimal
      Function Value: 2000.0
    Solver results file: results.yml
[    3.78] Applying Pyomo postprocessing actions
[    3.78] Pyomo Finished
```

Results file: results.yml

# Pyomo - Simple Example with Neos - Output

Results file: results.yml


```
Solution:
- number of solutions: 1
  number of solutions displayed: 1
- Gap: None
  Status: optimal
  Message: CPLEX 22.1.1.0\x3a optimal solution; objective 2000; 0 dual simplex iterations (0 in phase I)
  Objective:
    profit:
      Value: 2000
  Variable:
    x:
      Value: 50
  Constraint: No values
```

# Pyomo Model **Example2.py**

The source code defines:

- 2 variables declarations:
  - x, y
- 1 objective declarations:
  - profit
- 3 constraint declarations:
  - demand
  - laborA
  - laborB

```python
from pyomo.environ import *

# create a model
model = ConcreteModel()

# declare decision variables
model.x = Var(domain=NonNegativeReals)
model.y = Var(domain=NonNegativeReals)

# declare objective
model.profit = Objective(expr = 40*model.x + 30*model.y, sense=maximize)

# declare constraints
model.demand = Constraint(expr = model.x <= 40)
model.laborA = Constraint(expr = model.x + model.y <= 80)
model.laborB = Constraint(expr = 2*model.x + model.y <= 100)

model.pprint()

def display_solution(model):

    # display solution
    print('\nProfit = ', model.profit())

    print('\nDecision Variables')
    print('x = ', model.x.value)
    print('y = ', model.y())

    print('\nConstraints')
    print('Demand  = ', model.demand())
    print('Labor A = ', model.laborA())
    print('Labor B = ', model.laborB())

SolverFactory('clp').solve(model, tee=True).write()
#SolverFactory('cbc').solve(model, tee=True).write()
#SolverFactory('ipopt').solve(model, tee=True).write()
#SolverFactory('bonmin').solve(model, tee=True).write()
#SolverFactory('couenne').solve(model, tee=True).write()

display_solution(model)
```

# Pyomo - Predefined Virtual Sets for Variable Definition

`Any` = all possible values

`Reals` = floating point values

`PositiveReals` = strictly positive floating point values

`NonPositiveReals` = non-positive floating point values

`NegativeReals` = strictly negative floating point values

`NonNegativeReals` = non-negative floating point values

`PercentFraction` = floating point values in the interval [0,1]

`UnitInterval` = alias for PercentFraction

`Integers` = integer values

`PositiveIntegers` = positive integer values

`NonPositiveIntegers` = non-positive integer values

`NegativeIntegers` = negative integer values

`NonNegativeIntegers` = non-negative integer values

`Boolean` = Boolean values, which can be represented as False/True, 0/1, 'False'/'True' and 'F'/'T'

`Binary` = the integers {0, 1}

https://pyomo.readthedocs.io/en/stable/pyomo_modeling_components/Sets.html#predefined-virtual-sets

# Pyomo - Exercise

Solve the following optimization problem through NEOS solvers / local solvers.
Show processing time for each solvers.

$$\text{minimize} \quad x_3$$

subject to:

$$12x_1 + 9x_2 - x_3 = 0$$

$$x_{1,2} \in (-inf, +inf)$$

$$x_3 \geq 1, \quad x_{1,2,3} \in Z$$

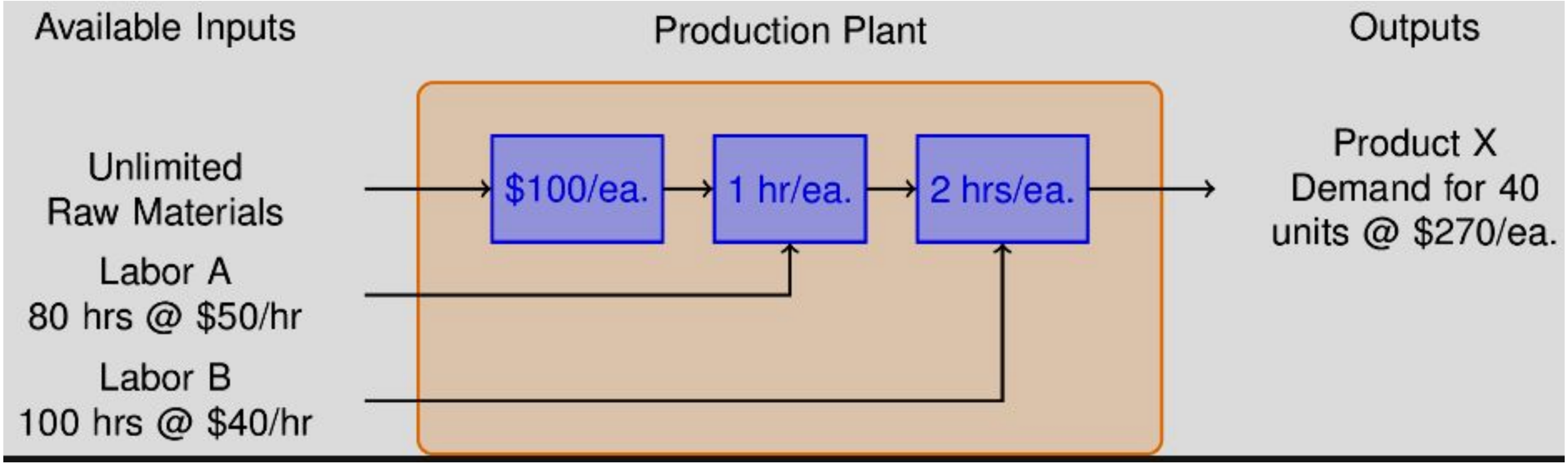
# Pyomo - Linear Programming Example

Use of linear programming to maximize profit for a simple model of a multiproduct production facility:

- Business to produce Product X;
- There is a market for X of up to 40 units per week at a price of USD 270 each;
- The production of each unit requires USD 100 of raw materials, 1 hour of type A labor, and 2 hours of type B labor;
- Only 80 hours per week of labor A at a cost of USD 50/hour, and 100 hours per week of labor B at a cost of USD 40 per hour;
- **What is the maximum weekly profit?**

# Pyomo - Linear Programming Example

Production plan for a single product plant:



# Pyomo - Linear Programming Example

Production plan for a single product plant

The essential decision we need to make is how many units of product X to produce each week. That's our decision variable which we denote as *x*

The weekly revenues are then: R = $270*x*

The costs include the value of the raw materials and each form of labor. If we produce x units a week, then the total cost is:

$$\text{Cost} = \underbrace{\$100x}_{\text{Raw Material}} + \underbrace{\$50x}_{\text{Labor A}} + \underbrace{2 \times \$40x}_{\text{Labor B}} = \$230x$$

# Pyomo - Linear Programming Example

The profit is just:

$$\begin{aligned} \text{Profit} &= \text{Revenue} - \text{Cost} \\ &= \$270x - \$230x \\ &= \$40x \end{aligned}$$

There are three constraints that limit how many units can be produced. There is market demand for no more than 40 units per week. Producing units per week will require 40 hours per week of Labor A, and 80 hours per week of Labor B. Checking those constraints we see that we have enough labor of each type.

The maximum profit will be:

$$\max \text{Profit} = \$40 \text{ per unit} \times 40 \text{ units per week} = \$1600 \text{ per week}$$

# Pyomo - Linear Programming **Example3.py**

Analyse the Pyomo model that generates a solution to the problem of production plan for a single product plant (file Example3.py).

# Pyomo - Linear Programming

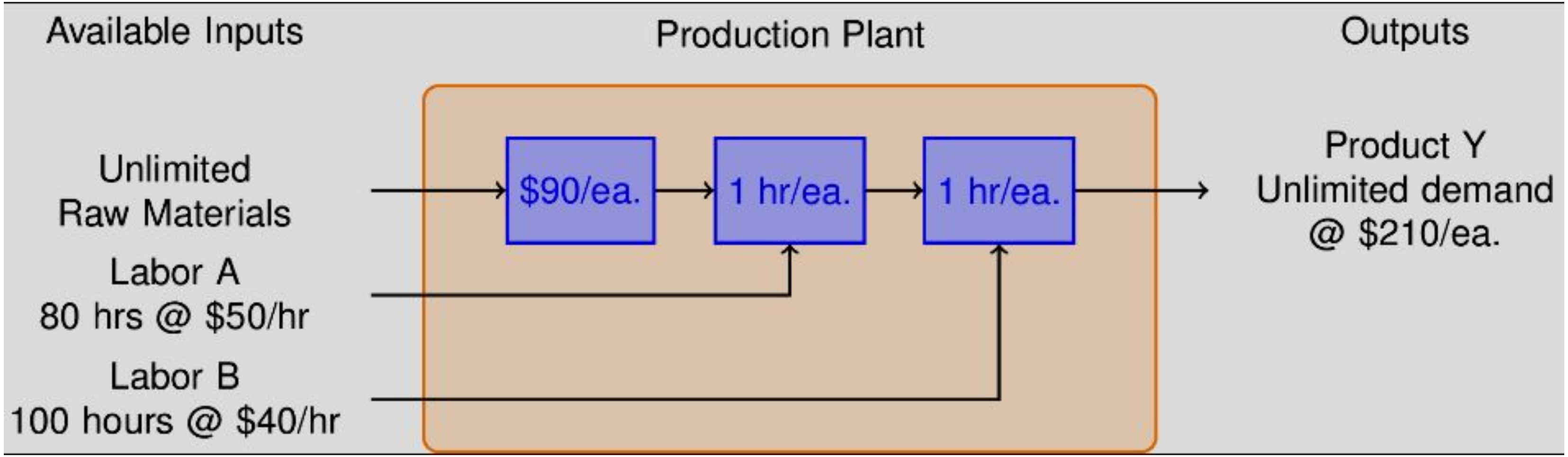
## Exercises

1. Suppose the demand could be increased to 50 units per week. What would be the increased profits?
2. What if the demand increased to 60 units per week?
3. Increase the cost of LaborB. At what point is it no longer financially viable to run the plant?

# Pyomo - Linear Programming - **Example4.py**

New product called Y. The product sells at a price of USD 210/each, requiring only USD 90 in raw materials, 1 hour of Labor type A at USD 50 per hour, and 1 hour of Labor B at USD 40 per hour.

What is the potential weekly profit? See file Example4.py



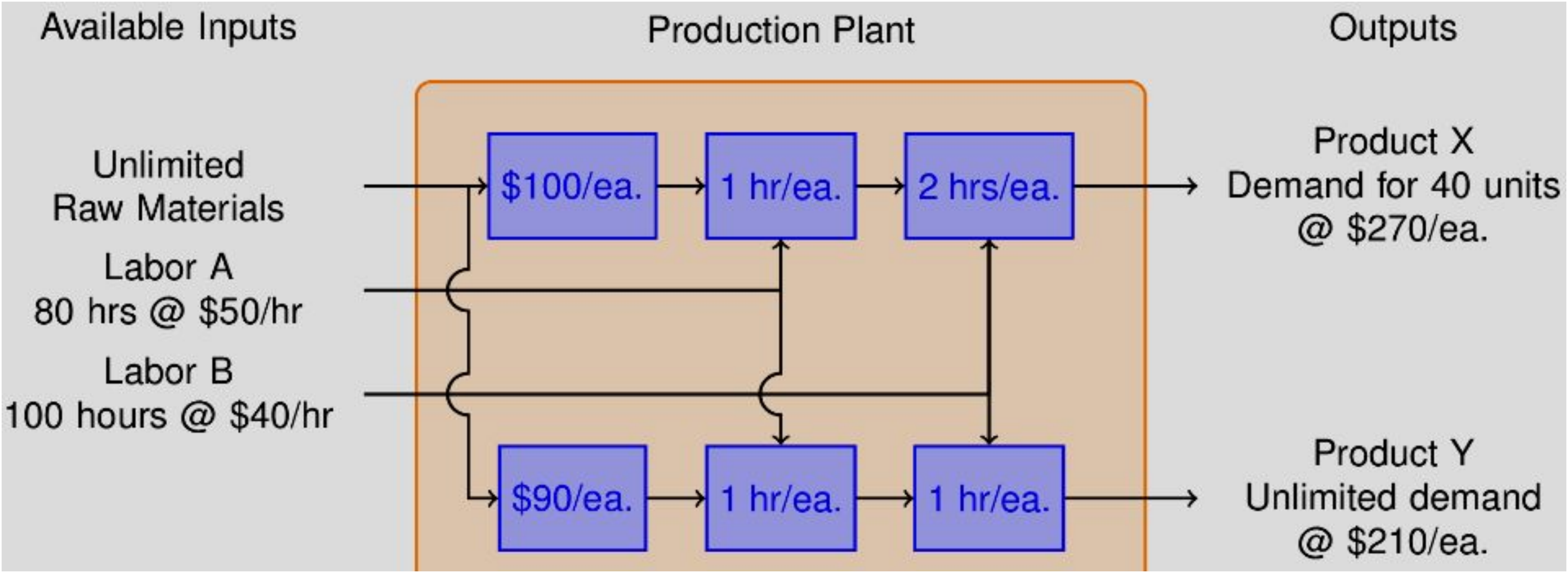
# Pyomo - Linear Programming - **Example5.py**

We can make $1,600 per week by manufacturing product X, and $2,400 per week manufacturing product Y.

**Is it possible to do even better?**

To answer this question, we consider the possibility of manufacturing both products in the same plant. The marketing department assures us that product Y will not affect the sales of product X. So the same constraints hold as before, but now we have two decision variables, *x* and *y*.

# Pyomo - Linear Programming - **Example5.py**

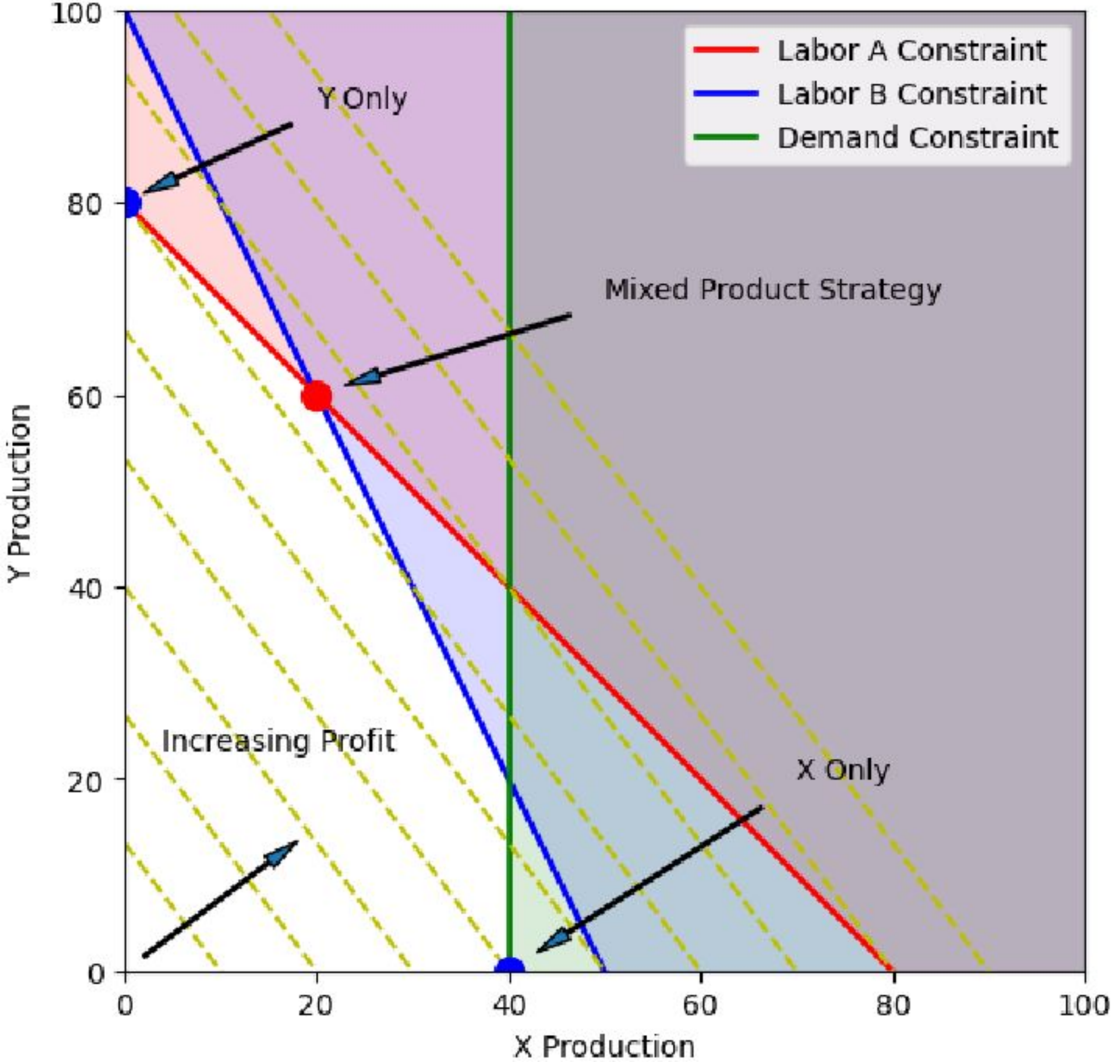


# Pyomo - Linear Programming - **Example5.py**

Analyse the Pyomo model that generates a solution to the problem of manufacturing both products X and Y in the same plant (file Example5.py).

# Pyomo - Linear Programming - **Example6.py**

Geometric representation



# Pyomo - Linear Programming - **Example7.py**

A plant produces three products in the amounts x, y, and z with unit profit of \$40, \$30, and \$50, respectively. There are several constraints imposed by product demand and the availability of specialized labor:

$$
\begin{aligned}
x &\leq 40 \\
x + y &\leq 80 \\
2x + z &\leq 100 \\
z &\leq 50
\end{aligned}
$$

In addition, the plant receives bonuses for meeting production targets:

If the plant produces more than 20 y items, then the unit profit for y

will be \$50 plus a fixed bonus profit of \$200.

If the plant produces more than 30 z items, then the unit profit for z will be \$60

plus a fixed bonus profit \$300.

**Find the optimal production targets.**

**Use of Gurobi solver**

# Pyomo - Linear Programming - **Example8.py**

The transportation problem deals with the distribution of a commodity from a set of sources to a set of destinations;

The object is to **minimize total transportation costs while satisfying constraints on the supplies available at each of the sources, and satisfying demand requirements at each of the destinations**;

Transportation costs between sources and destinations are given in units of €/ton of goods shipped, and list in the following table along with source capacity and demand requirements.

Two factories and six customer sites located in 8 European cities as shown in the following map.

# Pyomo - Linear Programming - **Example8.py**

The customer sites are labeled in red, the factories are labeled in blue.

# Pyomo - Linear Programming - **Example8.py**

Table of transportation costs, customer demand, and available supplies:

| Customer\Source | Arnhem [€/ton] | Gouda [€/ton] | Demand [tons] |
|---|---|---|---|
| London | - | 2.5 | 125 |
| Berlin | 2.5 | - | 175 |
| Maastricht | 1.6 | 2.0 | 225 |
| Amsterdam | 1.4 | 1.0 | 250 |
| Utrecht | 0.8 | 1.0 | 225 |
| The Hague | 1.4 | 0.8 | 200 |
| **Supply [tons]** | 550 tons | 700 tons | |

# Pyomo - Job Shop Scheduling - **Example9.py**

A job shop consists of a set of distinct machines that process jobs;

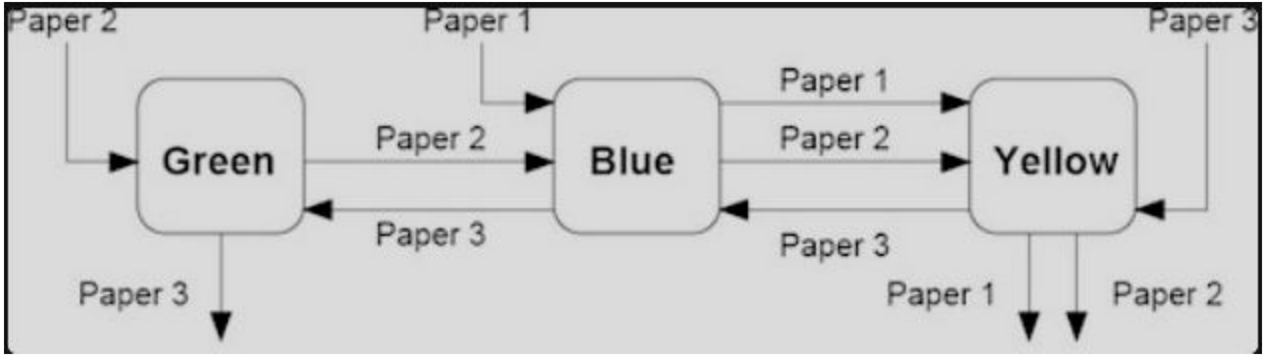
Each job is a series of tasks that require use of particular machines for known duration, and which must be completed in specified order;

The job shop scheduling problem is to schedule the jobs on the machines to minimize the time necessary to process all jobs (i.e, the makespan);

Job shop scheduling is one of the classic problems in Operations Research.

# Pyomo - Job Shop Scheduling - **Example9.py**

Three printed paper products that must pass through color printing presses in a particular order;



# Pyomo - Job Shop Scheduling - **Example9.py**

Data showing, in minutes, the amount of time each job requires on each machine:

| Machine | Color | Paper 1 | Paper 2 | Paper 3 |
|---|---|---|---|---|
| 1 | Blue | 45 | 20 | 12 |
| 2 | Green | - | 10 | 17 |
| 3 | Yellow | 10 | 34 | 28 |

What is the minimum amount of time (i.e, what is the makespan) for this set of jobs?

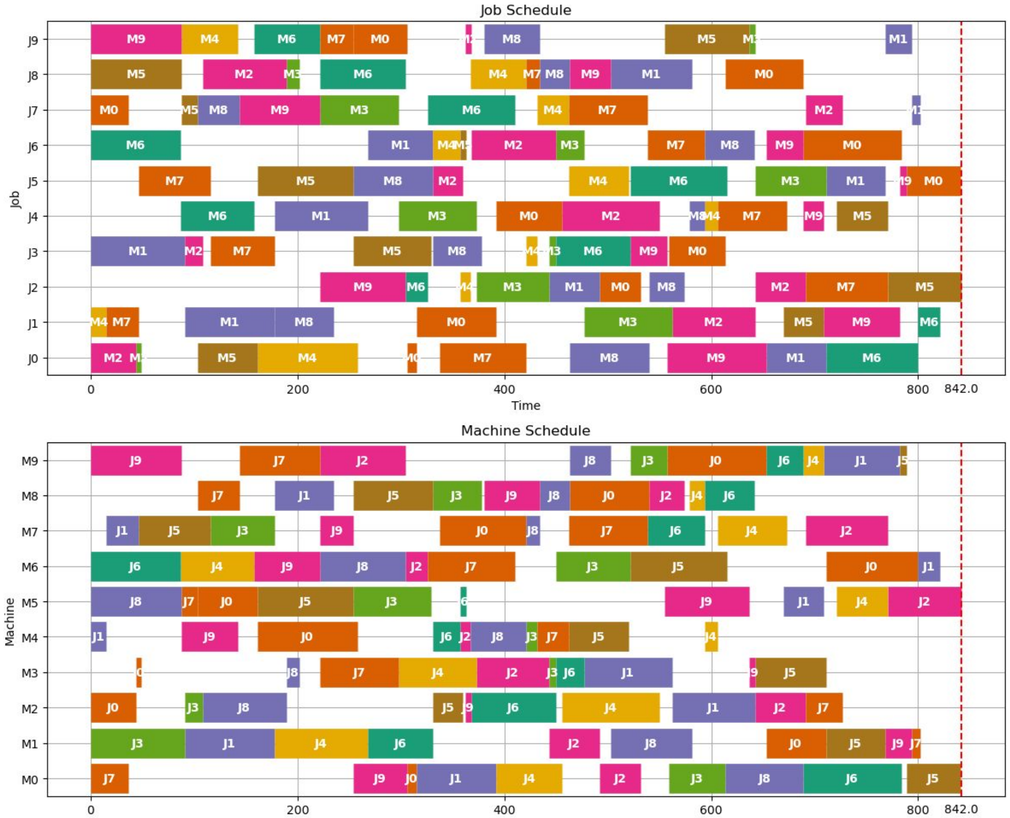
# Pyomo - Job Shop Scheduling - **Example9.py**

# Pyomo - Job Shop Scheduling - **Example10.py**

The data on each line is a sequence of (machine number, time) pairs showing the order in which machines process each job.

```
data = """
2  44  3   5  5  58  4  97  0   9  7  84  8  77  9  96  1  58  6  89
4  15  7  31  1  87  8  57  0  77  3  85  2  81  5  39  9  73  6  21
9  82  6  22  4  10  3  70  1  49  0  40  8  34  2  48  7  80  5  71
1  91  2  17  7  62  5  75  8  47  4  11  3   7  6  72  9  35  0  55
6  71  1  90  3  75  0  64  2  94  8  15  4  12  7  67  9  20  5  50
7  70  5  93  8  77  2  29  4  58  6  93  3  68  1  57  9   7  0  52
6  87  1  63  4  26  5   6  2  82  3  27  7  56  8  48  9  36  0  95
0  36  5  15  8  41  9  78  3  76  6  84  4  30  7  76  2  36  1   8
5  88  2  81  3  13  6  82  4  54  7  13  8  29  9  40  1  78  0  75
9  88  4  54  6  64  7  32  0  52  2   6  8  54  5  82  3   6  1  26
"""
```

# Pyomo - Job Shop Scheduling - **Example10.py**



# Pyomo - Time Table Scheduling - **Example11.zip**

Using Pyomo and mathematical modeling (Mixed Integer Programming) to solve scheduling problem;

Create a schedule that suits all teachers’ preferences, constraints, and can also take into account other requirements such as the maximum number of hours per subject and day or the capacity of the classrooms.

https://fran-espiga.medium.com/mixed-integer-programming-for-time-table-scheduling-eee326deda75

https://cienciadedatos.net/documentos/py38-optimizacion-horarios-python.html

# Picat - Logic-based multi-paradigm programming language

http://picat-lang.org/

# Picat - Logic-based multi-paradigm programming language

**Logic-based** programming language. First official release in 2013.

Program by writing a set of equations and ask the program to find values for each variable.

For example, if I want to concatenate two strings in Picat [1,2] ++ [3, 4] = Z.

It will then try to find a matching value for Z: Z = [1,2,3,4]

This is bidirectional: [1,2] ++ Y = [1,2,3,4].     gives  Y = [3,4]

member(X, [1, Y]) gives both X=1 and Y=X as possible solutions.

# Picat - Logic-based multi-paradigm programming language

**Multiparadigm** programming language.

**Logic programming language**: most of its syntax from Prolog. Finding values that match a set of equations.

**Constraint solving paradigm**: Like logic programming, constraint solving is about finding values that match a set of equations. Unlike logic programming, we're exclusively talking numbers. Usually we're also trying to find the solution that mini/maximizes some other metric.

**Imperative programming**

# Picat - Data Types

**Variables – plain and attributed**: X1 _ _ab

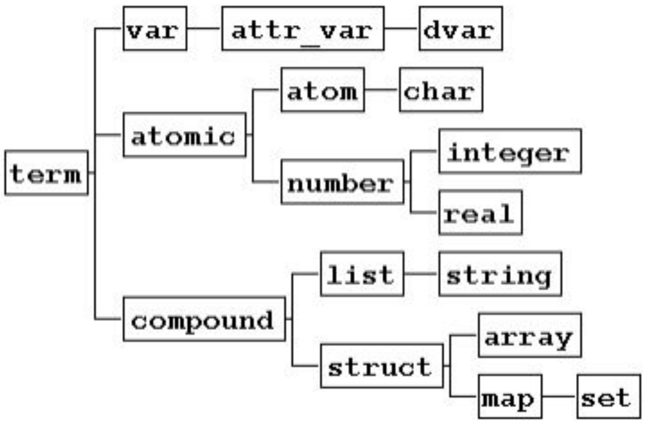
**Primitive values**

Integer and float
Atom: x1 '_' '_ab' '$%'

**Compound values**

List: [17,3,1,6,40]
Structure: $triangle(0.0,13.5,19.2)

## The Type Hierarchy



# Picat - Load and Run Programs fib.pi

```
main =>
        S = 0,
        I = 1,
        F = fib(I),
        while (F <= 4000000)
                if (F mod 2 == 0) then S := S+F
                end,
                I := I   +1,
                F := fib(I)
        end,
        printf("Sum of the even-valued terms is %w%n",S).

main([A1]) => printf("fib(%s)=%w%n",A1,A1.to_integer().fib()).

table
fib(1) = 1.
fib(2) = 2.
fib(N) = fib(N-1)+fib(N-2).
```

# Picat - Load and Run Programs fib.pi

Start Picat.

Compile and load the file using cl(fib).

Type main to run the program.

Or call the function fib by typing a query such as X=fib(100).

# Picat - Debug Program fib.pi

Start Picat.

Enable debug mode with debug.

Compile and run the file using cl(fib).

Type main to run the program.

At the entrance and exit of each call, the debugger displays the call and waits for a command. For the available debugging commands, type the question mark ?.

Use the command spy fib to set a spy point on the fib function. Note that only programs compiled in debug mode can be traced or spied on.

# Picat - Debug Program fib.pi

Run Programs Directly

Type the command picat fib. The Picat system will execute the main/0 predicate defined in fib.pi.

Type the command picat fib 100. The Picat system will execute the main/1 predicate, which calls the fib function.

If the command line contains arguments after the file name, then the Picat system calls main/1, passing all the arguments after the file name to the predicate as a list of strings.

# Picat - Creating Structures and Lists

**Generic Structure**

```
Picat> P = new_struct(point, 3)
P = point(_3b0,_3b4,_3b8)
Picat> S = $student(marry,cs,3.8)
```

**List Comprehension**

```
Picat> L = [E : E in 1..10, E mod 2 != 0]
L = [1,3,5,7,9]
```

**Range**

```
Picat> L = 1..2..10
L = [1,3,5,7,9]
```

# Picat - Creating Structures and Lists

**String**

```
Picat> write(“hello ”++“world”)
[h,e,l,l,o,’ ’,w,o,r,l,d]
```

**Arrays**

```
{2, 3, 5, 7, 11, 13, 17, 19}
Picat> A = new_array(2,3)
A = {{_3d0,_3d4,_3d8}, {_3e0,_3e4,_3e8}}
```

**Map**

```
Picat> M = new_map([alpha= 1, beta=2])
M = (map)[alpha = 1,beta = 2]
Picat> S = new_set([a,b,c])
S = (map)[c,b,a]
Picat> S = new_set([a,b,c]), S.has_key(b)
```

# Picat - Built-ins

```
Picat> integer(2)
yes
Picat> integer(2.0)
no
Picat> real(3.0)
yes
Picat> not real(3.0)
no
Picat> var(X)
yes
Picat> X = 5, var(X)
no
Picat> true
yes
Picat> fail
no
```

# Picat - Built-ins

```
Picat> X = to_binary_string(5), Y = to_binary_string(13)
           X = [‘1’, ‘0’, ‘1’]
           Y = [‘1’, ‘1’, ‘0’, ‘1’]

% X is an attributed variable
Picat> put_attr(X, age, 35), put_attr(X, weight, 205), A = get_attr(X, age)
A = 35

% X is a map
Picat> X = new_map([age=35, weight=205]), put(X, gender, male)
X = map([age=35, weight=205, gender=male])

Picat> S = $point(1.0, 2.0), Name = name(S), Arity = length(S)
Name = point
Arity = 2

Picat> I = read_int(stdin) % Read an integer from standard input
123
I = 123
```

# Picat - Index Notation

X[I1,…,In] : X references a compound value

```
Picat> L = [a,b,c,d], X = L[2]
X = b

Picat> S = $student(marry,cs,3.8), GPA=S[3]
GPA = 3.8

Picat> A = {{1, 2, 3}, {4, 5, 6}}, B = A[2, 3]
B = 6
```

# Picat - List Comprehension

[T : E1 in D1, Condn , . . . , En in Dn, Condn]

```
Picat> L = [X : X in 1..5].
       L = [1,2,3,4,5]

Picat> L = [(A,I): A in [a,b], I in 1..2].
       L = [(a,1),(a,2),(b,1),(b,2)]

Picat> L = [X : I in 1..5] % X is local
       L = [_bee8,_bef0,_bef8,_bf00,_bf08]

Picat> X=X, L = [X : I in 1..5] % X is non-local
       L = [X,X,X,X,X]
```

# Picat - OOP Notation

```
Picat> Y = 13.to_binary_string()
        Y = ['1', '1', '0', '1']

Picat> Y = 13.to_binary_string().reverse()
        Y = ['1', '0', '1', '1']

% X becomes an attributed variable
Picat> X.put_attr(age, 35), X.put_attr(weight, 205), A = X.get_attr(age)
A = 35

%X is a map
Picat> X = new_map([age=35, weight=205]), X.put(gender, male)
X = (map)([age=35, weight=205, gender=male])

Picat> S = $point(1.0, 2.0), Name = S.name, Arity = S.length
Name = point
Arity = 2

Picat> I = math.pi % module qualifier
I = 3.14159
```

# Picat - Explicit Unification

```
Picat> X=1
X=1

Picat> $f(a,b) = $f(a,b)
yes

Picat> [H|T] = [a,b,c]
H=a
T=[b,c]

Picat> $f(X,Y) = $f(a,b)
X=a
Y=b

Picat> $f(X,b) = $f(a,Y)
X=a
Y=b
```

# Picat - Predicates

**Relation with pattern-matching rules**

```
fib(0,F) => F=1.
fib(1,F) => F=1.
fib(N,F),N>1 => fib(N-1,F1),fib(N-2,F2),F=F1+F2.
fib(N,F) => throw $error(wrong_argument,fib,N).
```

# Picat - Predicates

**Backtracking (explicit non-determinism)**

```
member(X,[Y|_]) ?=> X=Y.
member(X,[_|L]) => member(X,L).
Picat> member(X,[1,2,3])
X = 1;
X = 2;
X = 3;
no
```

**Control backtracking**

```
Picat> once(member(X,[1,2,3]))
```

# Picat - Predicate Facts

```
index(+,-) (-,+)
edge(a,b).
edge(a,c).
edge(b,c).
edge(c,b).

edge(a,Y) ?=> Y=b.
edge(a,Y) => Y=c.
edge(b,Y) => Y=c.
edge(c,Y) => Y=b.
edge(X,b) ?=> X=a.
edge(X,c) ?=> X=a.
edge(X,c) => X=b.
edge(X,b) => X=c.
```

# Picat - Functions

Always succeed with a return value

```
power_set([]) = [[]].
power_set([H|T]) = P1++P2 =>
        P1 = power_set(T),
        P2 = [[H|S] : S in P1].

perm([]) = [[]].
perm(Lst) = [[E|P] : E in Lst, P in perm(Lst.delete(E))].

matrix_multi(A,B) = C =>
        C = new_array(A.length,B[1].length),
        foreach(I in 1..A.length, J in 1..B[1].length)
                C[I,J] = sum([A[I,K]*B[K,J] : K in 1..A[1].length])
        end.
```

# Picat - Patterns in Heads

As-patterns

```
merge([],Ys) = Ys.
merge(Xs,[]) = Xs.
merge([X|Xs],Ys@[Y|_])=[X|Zs],X<Y => Zs=merge(Xs,Ys).
merge(Xs,[Y|Ys])=[Y|Zs] => Zs=merge(Xs,Ys).
```

# Picat - Conditional Statements

**If-then-else**

```
fib(N)=F =>
    if (N=0; N=1) then F=1
    elseif N>1 then F=fib(N-1)+fib(N-2)
    else throw $error(wrong_argument,fib,N)
end.
```

**Prolog-style if-then-else** (C -> A; B)

**Conditional Expressions** fib(N) = cond((N==0;N==1), 1, fib(N-1)+fib(N-2))

# Picat - Loops

**Types**

foreach(E1 in D1, …, En in Dn) Goal end

while (Cond) Goal end

do Goal while (Cond)

```
sum_list(L)=Sum =>
        S=0,
        foreach (X in L)
                S:=S+X
        end,
        Sum=S.
```

```
Picat> S=sum_list([1,2,3])
S=6
```

# Picat - Tabling

Tabling memorizes calls and their answers in order to prevent infinite loops and to limit redundancy

```
table
fib(0)=1.
fib(1)=1.
fib(N)=fib(N-1)+fib(N-2).
```

Without tabling, fib(N) takes exponential time in N
With tabling, fib(N) takes linear time

# Picat - Mode-Directed Tabling

A table mode declaration instructs the system on what answers to table
table(M1,M2,…,Mn) where Mi is:

+: input
-: output
min: output, corresponding variable should be minimized
max: output, corresponding variable should be maximized
nt: not-tabled (only the last argument can be nt)

Mode-directed tabling is useful for dynamic programming problems

# Picat - Dynamic Programming

Shortest Path

```
table(+,+,-,min)
shortest_path(X,Y,Path,W) ?=>
      Path = [(X,Y)],
      edge(X,Y,W).
shortest_path(X,Y,Path,W) =>
      Path = [(X,Z)|PathR],
      edge(X,Z,W1),
      shortest_path(Z,Y,PathR,W2),
      W = W1+W2.
```

# Picat - Modules (example)

**% In file qsort.pi**
```
module qsort.
sort([]) = [].
sort([H|T]) = sort([E : E in T, E <= H) ++ [H] ++ sort([E : E in T, E > H).
```

**% In file isort.pi**
```
module isort.
sort([]) = [].
sort([H|T]) = insert(H, sort(T)).

private
insert(X,[]) = [X].
insert(X,Ys@[Y|_]) = Zs, X=<Y => Zs=[X|Ys].
insert(X,[Y|Ys]) = [Y|insert(X,Ys)].
```

**% another file test_sort.pi**
```
import qsort,isort.

sort1(L)=S => S=sort(L).
sort2(L)=S =>S=qsort.sort(L).
sort3(L)=S => S=isort.sort(L).
```

# Picat - Higher-Order Calls

Functions and predicates that take calls as arguments call(S,A1,…,An) Calls the named predicate with the specified arguments

apply(S,A1,…,An) Similar to call, except apply returns a value

findall(Template, Call) Returns a list of all possible solutions of Call in the form Template. findall forms a name scope like a loop.

```
Picat> C = $member(X), call(C, [1,2,3])
X = 1;
X = 2;
X = 3;
no
Picat> L = findall(X, member(X, [1, 2, 3]))
L = [1,2,3]
```

# Picat - Higher-Order Functions

```
map(_F,[]) = [].
map(F,[X|Xs])=[apply(F,X)|map(F,Xs)].

map2(_F,[],[]) = [].
map2(F,[X|Xs],[Y|Ys])=[apply(F,X,Y)|map2(F,Xs,Ys)].

fold(_F,Acc,[]) = Acc.
fold(F,Acc,[H|T])=fold(F, apply(F,H,Acc),T).
```

List comprehensions are significantly faster than higher-order calls

# Picat - Constraint programming language

A constraint program normally poses a problem in three steps:

(1) generate variables;

(2) generate constraints over the variables;

(3) call solve to find a valuation for the variables that satisfies the constraints and possibly optimizes an objective function.

# Picat - Basic Constraint Modeling

Picat provides four solver modules to solve constraint satisfaction and optimization problems (CSP):

- **cp** (Constraint Programming) - constraints on integer-domain variables
- **sat** (Satisfiability) - constraints on integer-domain variables
- **smt** (Satisfiability Modulo Theory) - constraints on integer-domain variables
- **mip** (Mixed Integer Programming) - constraints on integer-domain and real-domain variables (Gurobi, CBC or GLPK)

**Possibility to use the same model and syntax for four different solver modules.**

# Picat - Basic Constraint Modeling

Solver modules to solve Constraint Satisfaction Problem (CSP):

- **cp** (Constraint Programming) - best choice for problems in which effective global constraints and/or problem-specific labeling strategies are available.
- **sat** (Satisfiability) - well-suited to problems that can be clearly represented as Boolean expressions or have efficient CNF (Conjunctive Normal Form) encodings
- **smt** (Satisfiability Modulo Theory) - constraints on integer-domain variables
- **mip** (Mixed Integer Programming) - best choice for many kinds of Operations Research problems

Instructive to test all three solvers on the same problem.

# Picat - Basic Constraint Modeling

Picat provides interfaces to the external solvers:

- Kissat (https://github.com/arminbiere/kissat)
- Maxsat (compliant with the MaxSAT Evaluation's input and output formats.)
- Gurobi by Gurobi Optimization, Inc (www.gurobi.com)
- CBC by John Forrest (https://coin-or.github.io/Cbc/intro.html)
- GLPK by Andrew Makhorin (https://www.gnu.org/software/glpk)
- Z3 by Microsoft (https://www.microsoft.com/en-us/research/project/z3-3/)
- CVC4 (https://cvc4.github.io/)

# Picat - Constraint Modeling

Domain Variables

*Vars :: Exp*

This predicate restricts the domain or domains of V ars to Exp

For integer-domain variables, *Exp* must result in a list of integer values.

For real-domain variables for the mip module, *Exp* must be an interval in the form *L..U* , where *L* and *U* are real values.

# Picat - Constraint Modeling

Domain Variables

*Vars notin Exp*

This predicate excludes values *Exp* from the domain or domains of *Vars*

For integer-domain variables, *Exp* must result in a list of integer values.

This constraint cannot be applied to real-domain variables.

# Picat - Constraint Modeling

Domain Variables

*fd_disjoint(FDVar1,FDVar2)*: This predicate is true if FDV ar1's domain and FDVar2's domain are disjoint.

*fd_dom(FDV ar) = List*: This function returns the domain of FDVar as a list, where FDVar is an integer-domain variable. If FDVar is an integer, then the returned list contains the integer itself.

*fd_false(FDVar,Elm)*: This predicate is true if the integer Elm is not an element in the domain of FDVar.

*fd_true(FDVar,Elm)*: This predicate is true if the integer Elm is an element in the domain of FDVar.

# Picat - Constraint Modeling

Domain Variables

*fd_max(FDVar) = Max:* This function returns the upper bound of the domain of FDVar, where FDVar is an integer-domain variable.

*fd_min(FDVar) = Min:* This function returns the lower bound of the domain of FDVar, where FDVar is an integer-domain variable.

*fd_min_max(FDVar,Min,Max):* This predicate binds M in to the lower bound of the domain of FDVar, and binds Max to the upper bound of the domain of FDVar, where FDVar is an integer-domain variable.

# Picat - Constraint Modeling

Domain Variables

*fd_next(FDVar,Elm) = NextElm*: This function returns the next element of Elm in FDVar's domain. It throws an exception if Elm has no next element in FDVar's domain.

*fd_prev(FDVar,Elm) = PrevElm*: This function returns the previous element of Elm in FDVar's domain. It throws an exception if Elm has no previous element in FDVar's domain.

# Picat - Constraint Modeling

Domain Variables

*fd_size(F DV ar) = Size*: This function returns the size of the domain of FDVar, where FDVar is an integer-domain variable.

*new_dvar() = FDVar*: This function creates a new domain variable with the default

domain, which has the bounds -72057594037927935..72057594037927935 on

64-bit computers and -268435455..268435455 on 32-bit computers.

# Picat - Constraint Modeling

Table constraints

*table_in(DVars,R)*

*table_notin(DVars,R)*

where DVars is either a tuple of variables {X1 , . . . , Xn } or a list of tuples of variables, and R is a list of tuples in which each tuple takes the form {a1 , . . . , an }, where ai is an integer or the don't-care symbol ∗.

# Picat - Constraint Modeling

Table constraints example

```
import cp.

crossword(Vars) =>
    Vars = [X1,X2,X3,X4,X5,X6,X7],
    Words2 = [{ord('I'),ord('N')},
              {ord('I'),ord('F')},
              {ord('A'),ord('S')},
              {ord('G'),ord('O')},
              {ord('T'),ord('O')}],
    Words3 = [{ord('F'),ord('U'),ord('N')},
              {ord('T'),ord('A'),ord('D')},
              {ord('N'),ord('A'),ord('G')},
              {ord('S'),ord('A'),ord('G')}],
    table_in([{X1,X2},{X1,X3},{X5,X7},{X6,X7}], Words2),
    table_in([{X3,X4,X5},{X2,X4,X6}], Words3),
    solve(Vars),
    writeln([chr(Code) : Code in Vars]).
```

# Picat - Constraint Modeling

Arithmetic Constraints

*Exp1 Rel Exp2*

where Exp1 and Exp2 are arithmetic expressions, and Rel is one of the constraint operators: #=, #!=, #<, #=<, #<=, #>, or #>=.

An arithmetic expression is made from integers, variables, arithmetic functions, and constraints.

The following arithmetic functions are allowed: + (addition), - (subtraction), * (multiplication), / (truncated integer division), // (truncated integer division), count, div (floored integer division), mod, ** (power), abs, min, max, and sum.

# Picat - Constraint Modeling

Arithmetic Constraints

*cond(BoolConstr,ThenExp,ElseExp)*

*count(V ,DVars)*: The number of times V occurs in DVars, where DVars is a list of domain variables.

*max(DVars)*: The maximum of DVars, where DVars is a list of domain variables.

*max(Exp1,Exp2)*: The maximum of Exp1 and Exp2.

# Picat - Constraint Modeling

Arithmetic Constraints

*min(DVars)*: The minimum of DVars, where DVars is a list of domain variables.

*min(Exp1,Exp2)*: The minimum of Exp1 and Exp2.

*prod(DVars)*: The product of DVars, where DVars is a list of domain variables.

*sum(DVars)*: The sum of DVars, where DVars is a list of domain variables.

When a constraint occurs in an arithmetic expression, it is evaluated to 1 if it is satisfied and 0 if it is not satisfied.

# Picat - Sudoku Example

```
import cp.
sudoku(Board) =>
        N = Board.length,
        N1 = ceiling(sqrt(N)),
        Board :: 1..N,
        foreach(R in 1..N)
                all_different([Board[R,C] :
                C in 1..N])
        end,
        foreach(C in 1..N)
                all_different([Board[R,C] : R in 1..N])
        end,
        foreach(R in 1..N1..N, C in 1..N1..N)
                all_different([Board[R+I,C+J] :
                I in 0..N1-1, J in 0..N1-1])
        end,
        solve(Board)
```

```
board(Board) =>
  Board = {{_, 6, _, 1, _, 4, _, 5, _},
           {_, _, 8, 3, _, 5, 6, _, _},
           {2, _, _, _, _, _, _, _, 1},
           {8, _, _, 4, _, 7, _, _, 6},
           {_, _, 6, _, _, _, 3, _, _},
           {7, _, _, 9, _, 1, _, _, 4},
           {5, _, _, _, _, _, _, _, 2},
           {_, _, 7, 2, _, 6, 9, _, _},
           {_, 4, _, 5, _, 8, _, 7, _}}.
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 9 | 6 | 3 | 1 | 7 | 4 | 2 | 5 | 8 |
| 1 | 7 | 8 | 3 | 2 | 5 | 6 | 4 | 9 |
| 2 | 5 | 4 | 6 | 8 | 9 | 7 | 3 | 1 |
| 8 | 2 | 1 | 4 | 3 | 7 | 5 | 9 | 6 |
| 4 | 9 | 6 | 8 | 5 | 2 | 3 | 1 | 7 |
| 7 | 3 | 5 | 9 | 6 | 1 | 8 | 2 | 4 |
| 5 | 8 | 9 | 7 | 1 | 3 | 4 | 6 | 2 |
| 3 | 1 | 7 | 2 | 4 | 6 | 9 | 8 | 5 |
| 6 | 4 | 2 | 5 | 9 | 8 | 1 | 7 | 3 |

# Picat - Seesaw Example

The problem: Adam (36 kg), Boris (32 kg) and Cecil (16 kg) want to sit on a seesaw with the length 10 mts such that the minimal distances between them are more than 2 mts and the seesaw is balanced.

```
import cp.
seesaw(Sol) =>
      Sol = [A,B,C],
      Sol :: -5..5,
      A #=< 0,
      36*A+32*B+16*C #= 0,
      abs(A-B)#>2, abs(A-C)#>2, abs(B-C)#>2,
      solve(Sol).
```

# Picat - Example (Maximum Flow Problem)

Some directed node have a clear beginning (called the source) and a clear end (called the sink). Such graphs are flow graphs. Each node has an amount of inflow capacity (total weight of all edges going into the node) and an outflow capacity (total weight of all edges leaving the node).

The **maximum flow problem** seeks the maximum possible flow in a graph from a specified source node *Source* to a specified *Sink* node t without exceeding the capacity of any arc.

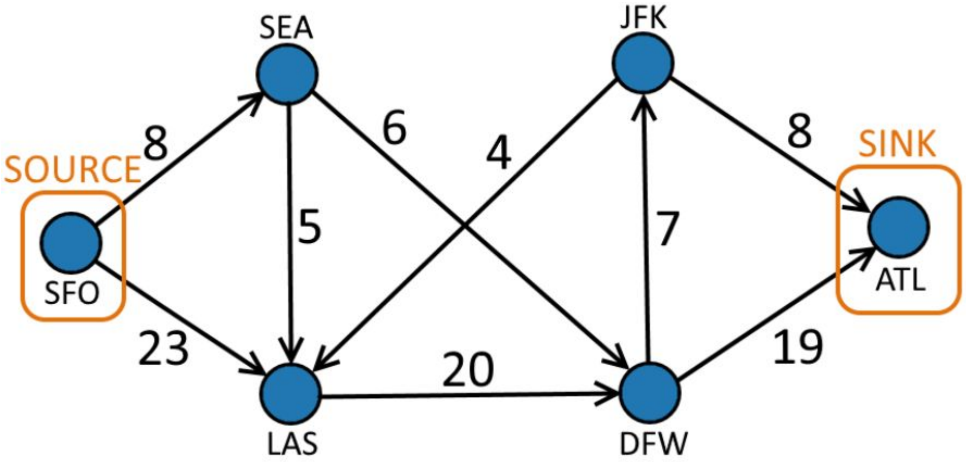
There are two principles to keep in mind when thinking about flow graphs:

- The actual outflow from a node cannot be larger than the inflow capacity.
- The actual outflow from a node cannot be larger than the outflow capacity.

# Picat - Example (Maximum Flow Problem)

Network of international airports, where the weights represent the maximum number of flights that can be scheduled between the two airports in a single day.

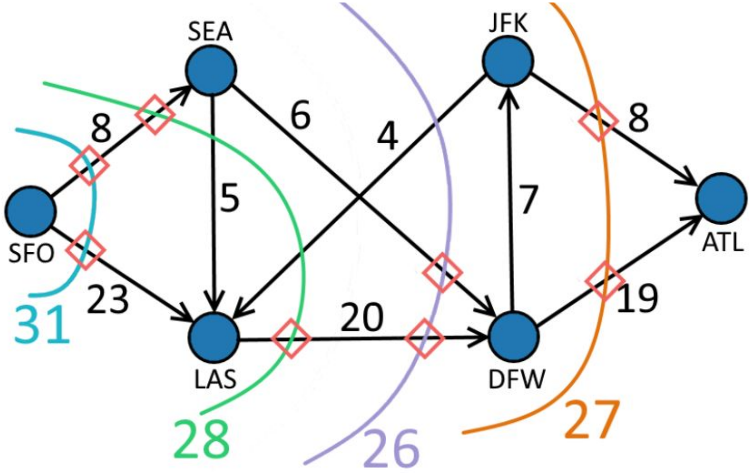
Suppose 23 flights leave from SFO to LAS, and 8 flights leave from SFO to SEA. How many flights will make it all the way through the network to ATL?



# Picat - Example (Maximum Flow Problem)

Cutting the network with a line, drawing through the edges and separating the network into two parts - one containing the source, and one containing the sink.

All possible cuts must be checked.



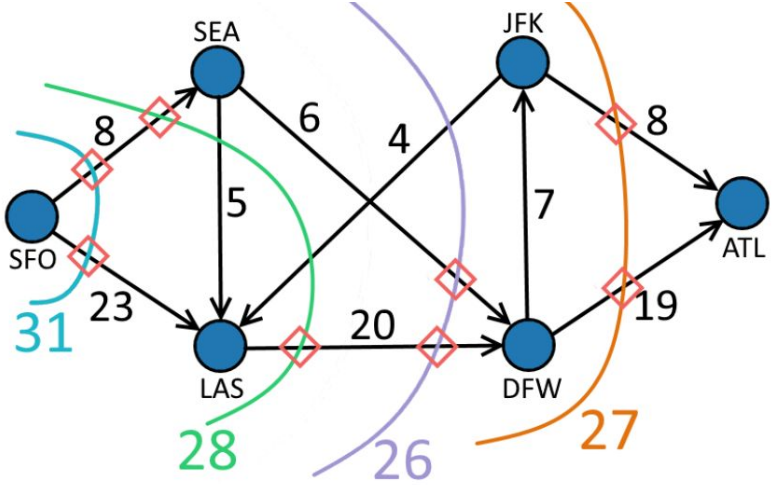
# Picat - Example (Maximum Flow Problem)

The cuts numbers give us the total weights of the edges flowing across the cut from the source to the sink.

The least value obtained after considering each cut, in this case 26.

It is the maximum flow through the network.

At most 26 flights can c
in a single day, and all flig
this line on their way from source to sink.



# Picat - Constraint Modeling

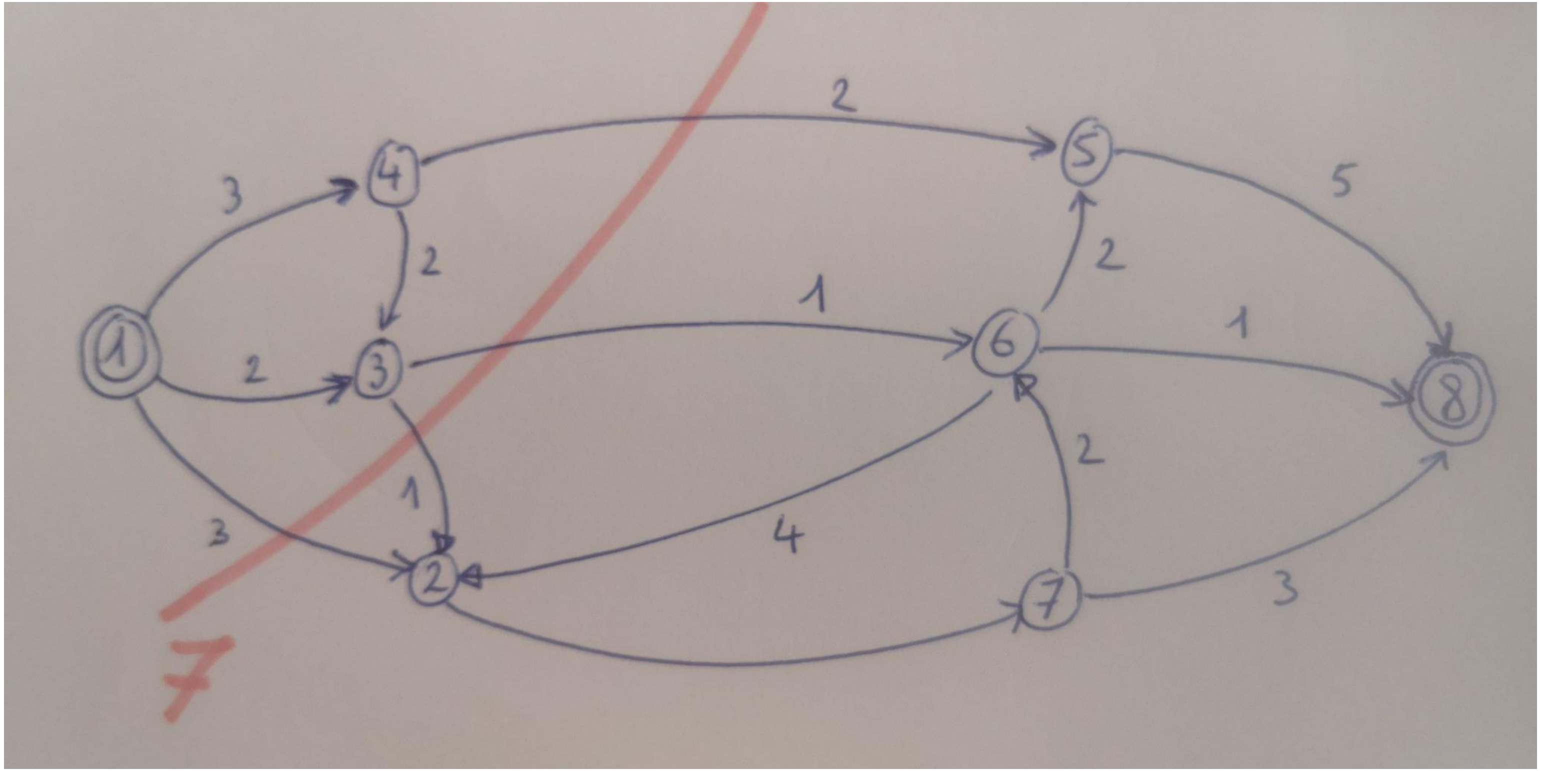
Example

```
import mip.

go =>
    M = {{0,3,2,3,0,0,0,0},
         {0,0,0,0,0,0,5,0},
         {0,1,0,0,0,1,0,0},
         {0,0,2,0,2,0,0,0},
         {0,0,0,0,0,0,0,5},
         {0,4,0,0,2,0,0,1},
         {0,0,0,0,0,2,0,3},
         {0,0,0,0,0,0,0,0}},
    maxflow(M,1,8).
```

```
maxflow(M,Source,Sink) =>
    N = M.length,
    X = new_array(N,N),
    foreach(I in 1..N, J in 1..N)
        X[I,J] :: 0..M[I,J]
    end,
    foreach(I in 1..N, I!=Source, I!=Sink)
        sum([X[J,I] : J in 1..N]) #= sum([X[I,J] : J in 1..N])
    end,
    Total #= sum([X[Source,I] : I in 1..N]),
    Total #= sum([X[I,Sink] : I in 1..N]),
    solve([$max(Total)],X),
    writeln(Total),
    writeln(X).
```

# Picat - Example (Maximum Flow Problem)



# Picat - Example (Maximum Flow Problem)

Gurobi output

```
Explored 1 nodes (1 simplex iterations) in 0.02 seconds (0.00 work units)
Thread count was 8 (of 8 available processors)

Solution count 4: 7 5 4 -0

Optimal solution found (tolerance 1.00e-04)
Best objective 7.000000000000e+00, best bound 7.000000000000e+00, gap 0.0
000%

Wrote result file '__tmp.sol'

7
{{0,3,2,2,0,0,0,0},{0,0,0,0,0,0,4,0},{0,1,0,0,0,1,0,0},{0,0,0,0,2,0,0,0},
{0,0,0,0,0,0,0,4},{0,0,0,0,2,0,0,1},{0,0,0,0,0,2,0,2},{0,0,0,0,0,0,0,0}}
```

# Picat - Constraint Modeling

Boolean Constraints

#~ *BoolExp* This constraint is 1 iff BoolExp is equal to 0.

*BoolExp* #/\ *BoolExp* This constraint is 1 iff both BoolExp1 and BoolExp2 are 1.

*BoolExp* #^ *BoolExp* BoolExp2: This constraint is 1 iff exactly one of BoolExp1 and BoolExp2 is 1.

*BoolExp* #\/ *BoolExp* This constraint is 1 iff BoolExp1 or BoolExp2 is 1.

*BoolExp* #=> *BoolExp* This constraint is 1 iff BoolExp1 implies BoolExp2.

*BoolExp* #<=> *BoolExp* This constraint is 1 iff BoolExp1 and BoolExp2 are equivalent.

BoolExp is either a Boolean constant (0 or 1), a Boolean variable (an integer-domain variable with the domain [0,1]), an arithmetic constraint, a domain constraint (in the form of V ar :: Domain or Var notin Domain), or a Boolean constraint.

# Picat - Constraint Modeling

Global Constraints

A global constraint is a constraint over multiple variables.

*acyclic(Vs,Es) (only available in sat)*: This constraint ensures that the undirected graph represented by Vs and Es contains no cycles.

*acyclic_d(Vs,Es) (only available in sat)*: This constraint ensures that the directed graph represented by Vs and Es contains no cycles.

*all_different(FDVars)*: This constraint ensures that each pair of variables in the list or array FDVars is different.

*all_distinct(FDVars)*: This constraint ensures that each pair of variables in the list or array FDVars is different

*all_different_except_0(F DV ars)*: This constraint is true if all non-zero values in FDV ars are different.

# Picat - Constraint Modeling

Global Constraints

*assignment(F DV ars1,F DV ars2)*: This constraint ensures that F DV ars2 is a dual assignment of F DV ars1, i.e., if the ith element of F DV ars1 is j, then the jth element of F DV ars2 is i.

*at_least(N,L,V)*: This constraint succeeds if there are at least N elements in L that are equal to V , where N and V must be integer-domain variables, and L must be a list of integer-domain variables.

*at_most(N ,L,V )*: This constraint succeeds if there are at most N elements in L that are equal to V , where N and V must be integer-domain variables, and L must be a list of integer-domain variables.

# Picat - Constraint Modeling

Global Constraints

*circuit(FDVars)*: Let FDVars be a list of variables [X1 , X2 , . . . , XN ], where each Xi has the domain 1..N . A valuation X1 = v1 , X2 = v2 , . . ., Xn = vn satisfies the constraint if 1->v1 , 2->v2 , ..., n->vn forms a Hamiltonian cycle.

This constraint ensures that each variable has a different value, and that the graph that is formed by the assignment does not contain any sub-cycles. For example, for the constraint circuit([X1,X2,X3,X4]) [3,4,2,1] is a solution, but [2,1,4,3] is not, because the graph 1->2, 2->1, 3->4, 4->3 contains two sub-cycles.

# Picat - Constraint Modeling

Global Constraints

*count(V ,FDVars,Rel,N)*: In this constraint, V and N are integer-domain variables, FDVars is a list of integer-domain variables, and Rel is an arithmetic constraint operator *(#=, #!=, #>, #>=, #<, #=<, or #<=).*

*count(V ,FDVars,N)*: This constraint is the same as count(V ,FDVars,#=,N ).

# Picat - Constraint Modeling

## Global Constraints

*cumulative(Starts,Durations,Resources,Limit):* This constraint is useful for describing and solving scheduling problems. The arguments Starts, Durations, and Resources are lists of integer-domain variables of the same length, and Limit is an integer-domain variable. Let Starts be [S1 , S2 , . . ., Sn ], Durations be [D1 , D2 , . . ., Dn ], and Resources be [R1 , R2 , . . ., Rn ]. For each job i, Si represents the start time, Di represents the duration, and Ri represents the units of resources needed. Limit is the limit on the units of resources available at any time. This constraint ensures that the limit cannot be exceeded at any time.

# Picat - Constraint Modeling

Global Constraints

*decreasing(L)*: The sequence (an array or a list) L is in (non-strictly) decreasing order.

*decreasing_strict(L)*: The sequence (an array or a list) L is in strictly decreasing order.

increasing(L)

increasing_strict(L)

*diffn(RectangleList)*: This constraint ensures that no two rectangles in RectangleList overlap with each other. A rectangle in an n-dimensional space is represented by a list of 2 × n elements [X1 , X2 , . . ., Xn , S1 , S2 , . . ., Sn ], where Xi is the starting coordinate of the edge in the ith dimension, and Si is the size of the edge.

# Picat - Constraint Modeling

Global Constraints

*element(I,List,V)*: This constraint is true if the Ith element of List is V , where I and V are integer-domain variables, and List is a list of integer-domain variables.

*exactly(N ,L,V )*: This constraint succeeds if there are exactly N elements in L that are equal to V , where N and V must be integer-domain variables, and L must be a list of integer-domain variables.

*global_cardinality(List,P airs)*

*disjunctive_tasks(Tasks)*

*hcp(V s,Es) (only available in sat)*

*hcp_grid(A) (only available in sat*

# Picat - Constraint Modeling

Global Constraints

*lex_le(L1 ,L2 )*: The sequence (an array or a list) L1 is lexicographically less than or equal to L2.

*lex_lt(L1 ,L2 )*: The sequence (an array or a list) L1 is lexicographically less than L2.

*matrix_element(Matrix,I,J,V )*: This constraint is true if the entry at <I,J> in Matrix is V, where I, J, and V are integer-domain variables, and Matrix is an two-dimensional array of integer-domain variables.

# Picat - Constraint Modeling

Global Constraints

*neqs(N eqList)*: N eqList is a list of inequality constraints of the form X #!= Y , where X and Y are integer-domain variables. This constraint is equivalent to the conjunction of the inequality constraints in N eqList, but it extracts all_distinct constraints from the inequality constraints.

*nvalue(N ,List)*: The number of distinct values in List is N , where List is a list of integer-domain variables.

*path(V s,Es,Src,Dest) (only available in sat)*: This constraint ensures that the undirected graph represented by V s and Es is a path from Src to Dest.

path_d(V s,Es,Src,Dest) (only available in sat).

# Picat - Constraint Modeling

Global Constraints

*regular(L, Q, S, M, Q0, F )*: Given a finite automaton (DFA or NFA) of Q states numbered 1, 2, . . ., Q with input 1..S, transition matrix M , initial state Q0 (1 ≤ Q0 ≤ Q), and a list of accepting states F , this constraint is true if the list L is accepted by the automaton. The transition matrix M represents a mapping from 1..Q × 1..S to 0..Q, where 0 denotes the error state. For a DFA, every entry in M is an integer, and for an NFA, entries can be a list of integers.

*scalar_product(A,X,Product)*: The scalar product of A and X is Product, where A and X are lists or arrays of integer-domain variables, and Product is an integer-domain variable. A and X must have the same length.

*scc(V s,Es) (only available in sat)*: This constraint ensures that the undirected graph represented by V s and Es is strongly connected.

scc(V s,Es,K) (only available in sat)

scc_grid(A) (only available in sat) scc_grid(A,K) (only available in sat) scc_d(V s,Es) (only available in sat) scc_d(V s,Es,K) (only available in sat)

# Picat - Constraint Modeling

Global Constraints

*serialized(Starts,Durations)*: This constraint describes a set of non-overlapping tasks, where Starts and Durations are lists of integer-domain variables, and the lists have the same length. Let Os be a list of 1s that has the same length as Starts. This constraint is equivalent to cumulative(Starts,Durations,Os,1).

*subcircuit(FDV ars)*: This constraint is the same as circuit(FDVars), except that not all of the vertices are required to be in the circuit. If the ith element of FDVars is i, then the vertex i is not part of the circuit.

subcircuit_grid(A) (only available in sat) subcircuit_grid(A,K) (only available in sat) tree(V s,Es) (only available in sat) tree(V s,Es,K) (only available in sat)

# Picat - Constraint Modeling

**Solver Invocation**

*solve(Opts,Vars)*: This predicate calls the imported solver to label the variables Vars with values, where Opts is a list of options for the solver.

*solve_all(Opts,Vars) = Solutions*: This function returns all the solutions that satisfy the constraints.

# Picat - Constraint Modeling

Common Solving Options

*$limit(N )*: Search up to N solutions.

*$max(Var)*: Maximize the variable Var.

*$min(Var)*: Minimize the variable Var.

$report(Call): Execute Call each time a better answer is found while searching for an optimal answer. This option cannot be used if the mip module is used.

# Picat - Constraint Modeling

**Solving Options for cp**

*backward*: The list of variables is reversed first.

*constr*: Variables are first ordered by the number of attached constraints.

*degree*: Variables are first ordered by degree, i.e., the number of connected variables.

*down*: Values are assigned to variables from the largest to the smallest.

*ff*: The first-fail principle is used: the leftmost variable with the smallest domain is selected.

*ffc*: The same as with the two options: ff and constr.

# Picat - Constraint Modeling

**Solving Options for cp**

*ffd*: The same as with the two options: ff and degree.

*forward*: Choose variables in the given order, from left to right.

*inout*: The variables are reordered in an inside-out fashion. For example, the variable list [X1,X2,X3,X4,X5] is rearranged into the list [X3,X2,X4,X1,X5].

*label(CallName)*: This option informs the CP solver that once a variable V is selected, the user-defined call CallName(V ) is used to label V , where CallName must be defined in the same module, an imported module, or the global module.

*leftmost*: The same as forward.

*max*: First, select a variable whose domain has the largest upper bound, breaking ties by selecting a variable with the smallest domain.

# Picat - Constraint Modeling

**Solving Options for cp**

*min*: First, select a variable whose domain has the smallest lower bound, breaking ties by selecting a variable with the smallest domain.

*rand*: Both variables and values are randomly selected when labeling.

*rand_var*: Variables are randomly selected when labeling.

*rand_val*: Values are randomly selected when labeling.

*reverse_split*: Bisect the variable’s domain, excluding the lower half first.

*split*: Bisect the variable’s domain, excluding the upper half first.

*updown*: Values are assigned to variables from the values that are nearest to the middle of the domain.

# Picat - Constraint Modeling

**Solving Options for sat**

*dump*: Dump the CNF code to stdout.

*dump(File)*: Dump the CNF code to File.

*seq: Use sequential search to find an optimal answer.*

*split: Use binary search to find an optimal answer (default).*

*$nvars(NVars): The number of variables in the CNF code is NVars.*

*$ncls(NCls): The number of clauses in the CNF code is N Cls.*

# Picat - Constraint Modeling

**Solving Options for mip**

*cbc*: Instruct Picat to use the Cbc MIP solver.

*dump*: Dump the constraints in CPLEX format to stdout.

*dump(File)*: Dump the CPLEX format to File.

*glpk*: Instruct Picat to use the GLPK MIP solver.

*gurobi*: Instruct Picat to use the Gurobi MIP solver.

*tmp(File)*: Dump the CPLEX format to File rather than the default file "__tmp.lp"

Internally: gurobi_cl ResultFile=res.sol __tmp.lp

# Picat - Constraint Modeling

**Solving Options for smt**

*cvc4*: Instruct Picat to use the CVC4 SMT solver.

*dump*: Dump the constraints in SMT-LIB2 format to stdout.

*dump(File)*: Dump the SMT-LIB2 format to File.

*logic(Logic)*: Instruct the SMT solver to use Logic in the solving.

*tmp(F ile)*: Dump the SMT-LIB2 format to File rather than the default file "__tmp.smt2"

*z3*: Instruct Picat to use the z3 SMT solver

# Picat - Example using CP Module SEND+MORE=MONEY

The problem is to substitute each letter (SENDMORY) with a distinct digit in the range of 0..9, such that the equation SEND + MORE = MONEY is satisfied

```
import cp.

main =>
  Digits = [S,E,N,D,M,O,R,Y],
  Digits :: 0..9,
  all_different(Digits),
  S #> 0,
```

```
M #> 0,
                 1000*S + 100*E + 10*N + D
+                1000*M + 100*O + 10*R + E
#= 10000*M + 1000*O + 100*N + 10*E + Y,
solve(Digits),
println(Digits).
```

# Picat - Example using CP Module SEND+MORE=MONEY

Digits :: 0..9: This defines the domains of the decision variables in the list Digits, and thus the single variables S, E, N, D, M, O, R, and Y.

#> and #=: These are arithmetic constraint operators. All of the arithmetic constraint operators begin with #. This special notation distinguishes between the constraint operators and the normal relational operators in Picat.

Changing #= in the program to = would result in an evaluation error, since the expressions involve uninstantiated variables, meaning that the functions cannot be evaluated.

```
import cp.

main =>
  Digits = [S,E,N,D,M,O,R,Y],
  Digits :: 0..9,
  all_different(Digits),
  S #> 0,
```

```
M #> 0,
                 1000*S + 100*E + 10*N + D
+                1000*M + 100*O + 10*R + E
#= 10000*M + 1000*O + 100*N + 10*E + Y,
solve(Digits),
println(Digits).
```

# Picat - Example using CP Module SEND+MORE=MONEY

all_different(Digits): all different is a global constraint which states that all of the decision variables in Digits must be distinct. Global constraints are quite unique to CP.

solve(Digits): The solve predicate finds an assignment of values to the variables that satisfies all of the accumulated constraints

```
import cp.

main =>
  Digits = [S,E,N,D,M,O,R,Y],
  Digits :: 0..9,
  all_different(Digits),
  S #> 0,
```

```
M #> 0,
               1000*S + 100*E + 10*N + D
+              1000*M + 100*O + 10*R + E
#= 10000*M + 1000*O + 100*N + 10*E + Y,
solve(Digits),
println(Digits).
```

# Picat - Minesweeper

The goal is to identify the positions of all the mines in a given matrix, with hints that state how many mines there are in the neighboring cells, including diagonal neighbors. If there is a hint in a cell, then it cannot be a mine.

For this instance, the third cell in the first row has the value 2, which indicates that it has two adjacent mines.

The cells marked with “.” are unknowns, meaning that the cell may or may not have a mine.

```
. . 2 . 3 .
2 . . . . .
. . 2 4 . 3
1 . 3 4 . .
. . . . . 3
. 3 . 3 . .
```

# Picat - Minesweeper using SAT Module

```
import sat.

main =>
  % define the problem instance
  problem(Matrix),
  NRows = Matrix.length,
  NCols = Matrix[1].length,

  % decision variables: where are the mines?
  Mines = new_array(NRows,NCols),
  Mines :: 0..1,
```

```
problem(P) =>
  P = {{_,_,2,_,3,_},
       {2,_,_,_,_,_},
       {_,_,2,4,_,3},
       {1,_,3,4,_,_},
       {_,_,_,_,_,3},
       {_,3,_,3,_,_}}.
```

# Picat - Minesweeper using SAT Module

```
foreach (I in 1..NRows, J in 1..NCols)

  % only check those cells that have hints
  if ground(Matrix[I,J]) then

    % The number of neighboring mines must equal Matrix[I,J].
    Matrix[I,J] #= sum([Mines[I+A,J+B] :
                             A in -1..1, B in -1..1,
                             I+A >  0, J+B >  0,
                             I+A =< NRows, J+B =< NCols]),

    % If there is a hint in a cell, then it cannot be a mine.
    Mines[I,J] #= 0
  end
end,
solve(Mines),
println(Mines).
```

# Picat - Minesweeper - CP vs SAT Solvers

For large instances, the sat module tends to be faster than other solver modules. In general, SAT solvers tends to outperform CP solvers on 0/1 integer programming modules.

The timings for selected N random hints values with at least one solvable instance are shown in following table:

| N | solutions | CP (s) | SAT (s) | Winner |
|---|---|---|---|---|
| 50 | 1 | 0.016 | 0.072 | CP |
| 107 | 1 | 0.068 | 0.208 | CP |
| 202 | 1 | 0.284 | 0.548 | CP |
| 430 | 1 | 1.96 | 2.19 | CP |
| 440 | 2 | 3.6 | 3.08 | SAT |
| 450 | 5 | 10.1 | 6.46 | SAT |
| 500 | 3 | 7.5 | 5.03 | SAT |
| 601 | 1 | 4.99 | 3.65 | SAT |
| 1000 | 1 | 21.69 | 9.47 | SAT |
| 1500 | 3 | 804.6 | 61.27 | SAT |
| 1601 | 1 | 143.25 | 40.79 | SAT |

# Picat - Diet Problem using MIP Module

Given a set of foods, each of which has given nutrient values, a cost per serving, and a minimum limit for each nutrient, the objective of the diet problem is to select the number of servings of each food to consume so as to minimize the cost of the food while meeting the nutritional constraints;

A diet is required to contain at least 500 calories, 6 ounces of chocolate, 10 ounces of sugar, and 8 ounces of fat.

| Type of Food | Calories | Chocolate (oz.) | Sugar (oz.) | Fat (oz.) | Price (cents) |
|---|---|---|---|---|---|
| Chocolate Cake (1 slice) | 400 | 3 | 2 | 2 | 50 |
| Chocolate ice cream (1 scoop) | 200 | 2 | 2 | 4 | 20 |
| Cola (1 bottle) | 150 | 0 | 4 | 1 | 30 |
| Pineapple cheesecake (1 piece) | 500 | 0 | 4 | 5 | 80 |
| Limits | 500 | 6 | 10 | 8 | – |

# Picat - Diet Problem using MIP Module

```
diet(Calories,Chocolate,Sugar,Fat,Price,Limits, Xs,XSum) =>
    Len = length(Price),
    Xs = new_list(Len),
    Xs :: 0..10,
    scalar_product(Calories,  Xs, #>=, Limits[1]), % 500,
    scalar_product(Chocolate, Xs, #>=, Limits[2]), %   6,
    scalar_product(Sugar,     Xs, #>=, Limits[3]), %  10,
    scalar_product(Fat,       Xs, #>=, Limits[4]), %   8,
    scalar_product(Price, Xs, #=, XSum), % to minimize
    % optimize or find all (optimal) solutions
    if var(XSum) then
        solve([$min(XSum)], Xs)
    else   % here XSum is bound so we just label the vars
        solve(Xs)
    end.
```

# Picat - Traveling Salesman Problem TSP

Given a set of cities, the objective is to find a tour of all of the cities such that the total traveling cost is minimized.

TSP is a graph problem. For a given graph, which can be directed or undirected, the goal of TSP is to find a tour that connects all of the vertices of the graph such that the total travel cost is minimized.

A combinatorial problem can normally be modeled in different ways and solved by different solvers:

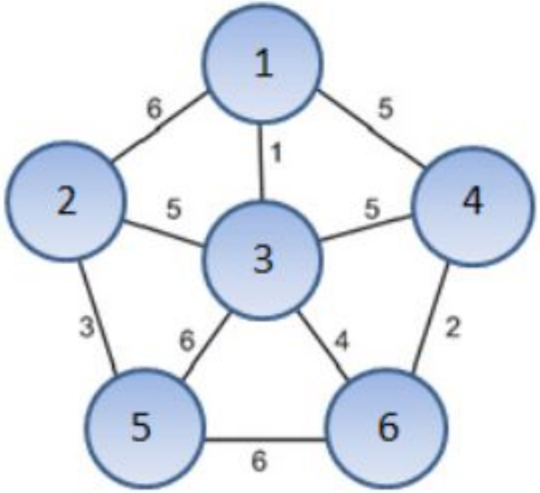
- Several models for solving the Traveling Salesman Problem;
- Resolution through different solvers including CP, SAT, MIP, and tabled planning.

# Picat - Traveling Salesman Problem TSP

An Encoding for CP

```
import cp.

main =>
    M = {{0,6,1,5,0,0},
         {6,0,5,0,3,0},
         {1,5,0,5,6,4},
         {5,0,5,0,0,2},
         {0,3,6,0,0,6},
         {0,0,4,2,6,0}},
    tsp(M).
```



# Picat - Traveling Salesman Problem TSP

```
tsp(M) =>
    N = length(M),
    NextArr = new_array(N),      % visit NextArr[I] after I
    NextArr :: 1..N,
    CostArr = new_array(N),
    circuit(NextArr),
    foreach (I in 1..N)
       CostArr[I] #> 0,
       element(NextArr[I],M[I],CostArr[I])
    end,
    TotalCost #= sum(CostArr),
    solve($[min(TotalCost), report(println(cost=TotalCost))],
          NextArr),
    foreach (I in 1..N)
       printf("%w -> %w\n",I,NextArr[I])
    end.
```

# Picat - Flexible JobShop Problem

- Manufacturing enterprises
  - Efficient planning
  - Competitiveness
- Heuristics
  - Fast to reach a solution
  - Solution close to optimal
- Recent advances in mathematical optimization solvers
  - Improved their performance
  - Optimum scheduling problems solving
  - Computation time feasibility

# Picat - Flexible JobShop Problem

- The flexible Job-Shop problem is a combinatorial optimization problem
  - Minimize makespan
- Solution
  - Find an optimal schedule for a set of jobs on a set of machines
  - Take the smallest possible amount of time to conclude all operations

# Flexible Job-Shop Problem Formulation

- Combinatorial optimization problem
  - All operations for a job must be made in order
  - A machine can only process one operation at a time
  - An operation can only be processed in one machine from the multiple machines possibility
- Formulation based on a set of variables that represent the start times of each operation on each machine
- No two jobs are scheduled for the same time on the same machine

# Experiment

- Dataset consists in 8 jobs, each with up to 7 operations, on a setup of 8 different machines.

# Picat - Flexible Job-Shop Scheduling

| Process Plan | Operation | | | | | | |
|---|---|---|---|---|---|---|---|
| | O 1 | O 2 | O 3 | O 4 | O 5 | O 6 | O 7 |
| $pr_{1,2}$ | (1,3) [4,5] | (2,4) [4,5] | (3,5) [5,6] | (4,5,6,7,8) [5,5,4,5,9] | | | |
| $pr_{2,2}$ | (1,3,5) [1,5,7] | (4,8) [5,4] | (4,6) [1,6] | (4,7,8) [4,4,7] | (4,6) [1,2] | (1,6,8) [5,6,4] | (4) [4] |
| $pr_{3,3}$ | (2,3,8) [7,6,8] | (4,8) [7,7] | (3,5,7) [7,8,7] | (4,6) [7,8] | (1,2) [1,4] | | |
| $pr_{4,2}$ | (1,3,5) [4,3,7] | (2,8) [4,4] | (3,4,6,7) [4,5,6,7] | (5,6,8) [3,5,5] | | | |
| $pr_{5,1}$ | (1) [3] | (2,4) [4,5] | (3,8) [4,4] | (5,6,8) [3,3,3] | (4,6) [5,4] | | |
| $pr_{6,3}$ | (1,2,3) [3,5,6] | (4,5) [7,8] | (3,6) [9,8] | | | | |
| $pr_{7,2}$ | (3,5,6) [4,5,4] | (4,7,8) [4,6,4] | (1,3,4,5) [3,3,4,5] | (4,6,8) [4,6,5] | (1,3) [3,3] | | |
| $pr_{8,1}$ | (1,2,6) [3,4,4] | (4,5,8) [6,5,4] | (3,7) [4,5] | (4,6) [4,6] | (7,8) [1,2] | | |

# Results

- Mathematical optimization solution based on solvers
  - Mathematical optimization can find the optimum solution: 28 time units
  - Small amount of processing time for this problem size

| Solver | Gurobi | Z3 | Game-theoretic |
|---|---|---|---|
| Processing time (seconds) | 0,83 | 1,51 | n.a. |
| Makespan | 28 | 28 | 41 |

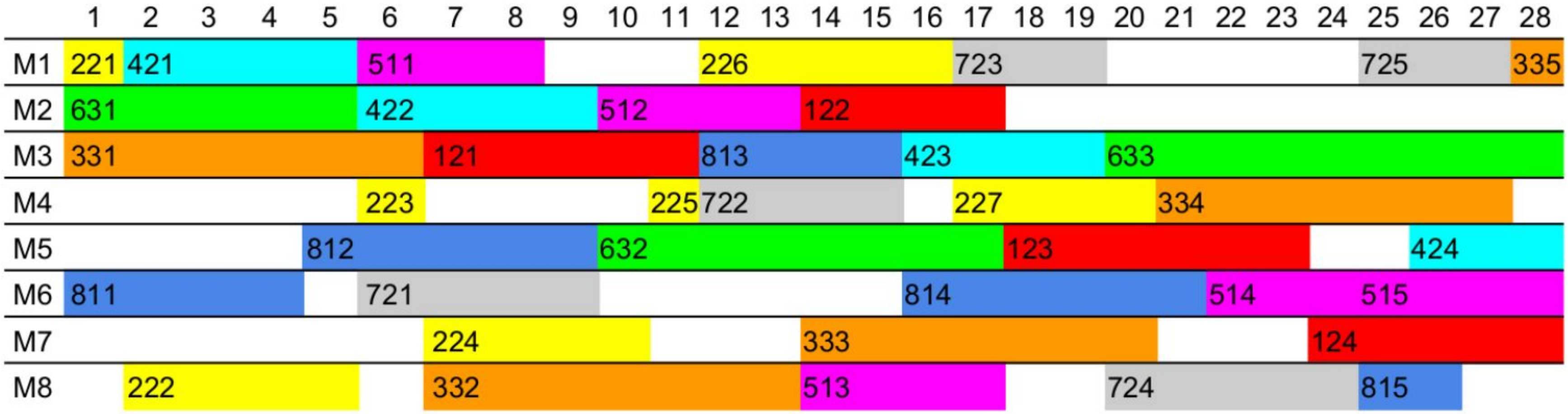
# Flexible Job-Shop Picat Formulation

- Picat: a rule-based programming language

```
foreach (I in 1..N)
    ...
    Machine[I] #!= 0,    Units[I] #!= 0,
    ...
    TimeEnd[I] #= Timeinit[I]+Units[I]-1,
    foreach (J in (I+1)..N) (Machine[I] #= Machine[J]) #=>
            ((TimeEnd[I] #< TimeInit[J]) #\/ (TimeEnd[J] #< Timeinit[I])) end
end,
foreach (I in 2..N) (M[I-1,1] #= M[I,1]) #=>  (TimeEnd[I-1] #< TimeInit[I]) end,
solve([...], [Machine,Units,TimeInit,TimeEnd])
```

# Optimum Solution

- Mathematical optimization solution based on solvers
  - Automatic Gantt chart generation for the visualization
  - Optimum solution with twenty-eight units of time for the makespan



# Picat - Flexible Job-shop Conclusions

- Flexible Job-Shop Scheduling Problem
  - Solution needed by the manufacturing enterprises
  - Proposed solution to generate a optimal scheduling solution
  - Game theoretic approach comparison
  - Mathematical optimization formulation
  - Fast computation time
  - Viable approach to achieve a valid solution

# Picat - Planner Module

Module for declarative solving planning problems;

To solve a planning problem in Picat a programmer needs to define an initial state, a final predicate for the final state, and an action predicate for possible actions.

Picat tries to find a sequence of actions;

The final predicate in its simplest form has only one parameter – the current state – and succeeds if the state is final;

The action predicate usually has several clauses – one for each possible action. The predicate has 4 parameters: current state, new state, action name, and action cost.

# Picat - Planner Module

Picat's predicate for finding an optimal plan – best_plan – has 2 input parameters: the initial state and the resource limit, and 2 output parameters: the best plan and its cost;

To find an optimal plan the system uses iterative deepening depth-first search-like algorithm. If no plan was found and the maximum resource limit was reached, the predicate fails.

Picat's planning module uses tabling (a form of memoization) to convert a tree search through state space to a graph search.

# Picat - Planner Module Predicates

***final(S)***: This predicate succeeds if S is a final state.

***action(S,NextS,Action,ACost)***: This predicate encodes the state transition diagram of a planning problem. State S can be transformed to NextS by performing Action. The cost of Action is ACost, which must be non-negative. If the plan's length is the only interest, then ACost=1. Note that the assignment operator := cannot be used to update variable S to produce NextS.

The final and action predicates are called by the planner.

The action predicate specifies the precondition, effect, and cost of each of the actions.

# Picat - Planner Module Predicates

resource-unbounded search

***best_plan_unbounded(S,Limit,Plan,PlanCost)***: This predicate, if it succeeds, binds Plan to a plan that can transform state S to a final state. PlanCost is the cost of Plan. PlanCost cannot exceed Limit, which is a given non-negative integer. The argument PlanCost is optional. If it is omitted, then the predicate does not return the plan's cost. The Limit can also be omitted. In this case, the predicate assumes the cost limit to be 268435455;

***plan_unbounded(S,Limit,Plan,PlanCost)***: This predicate is the same as the best plan unbounded predicate, except that it terminates the search once it finds a plan whose cost does not exceed Limit.

# Picat - Planner Module Predicates

resource-bounded search

***best_plan(S,Limit,Plan,PlanCost)***: This predicate finds an optimal plan by using iterative-deepening. The best plan predicate calls the plan/4 predicate to find a plan, using 0 as the initial cost limit and gradually relaxing the cost limit until a plan is found.

***best_plan_bb(S,Limit,Plan,PlanCost)***: This predicate finds an optimal plan by using branch-and-bound. First, the best plan bb predicate calls plan/4 to find a plan. Then, it tries to find a better plan by imposing a stricter limit. This step is repeated until no better plan can be found. Then, this predicate returns the best plan that was found.

***plan(S,Limit,Plan,PlanCost)***: This predicate searches for a plan by performing resource-bounded search. The predicate binds Plan to a plan that can transform state S to a final state that satisfies the condition given by final/1. PlanCost is the cost of Plan. PlanCost cannot exceed Limit, which is a given non-negative integer. The arguments Limit and PlanCost are optional.

# Picat - Planner Module Predicates

resource-bounded search

***current resource() = Limit***: This function returns the resource limit argument of the latest call to plan/4. In order to retrieve the Limit argument, the implementation has to traverse the call-stack until it reaches a call to plan/4. The current resource function can be used to check against a heuristic value. If the heuristic estimate of the cost to travel from the current state to a final state is greater than the resource limit, then the current state should fail. If the estimated cost never exceeds the real cost, meaning that the heuristic function is admissible, then the optimality of solutions is guaranteed.

# Picat - Planner Module Predicates

**Resource-unbounded** search does not work for problem if the search space is infinite. The path in the search space can go infinitely deep.

**Resource-bounded** search avoids unfruitful exploration of paths that are deemed to fail.

# Picat - Planner Module - Deadfish Example

The esoteric programming language Deadfish has one accumulator (which starts at 0) and 4 commands: i to increment the accumulator, s to square the accumulator, d to decrement the accumulator, and o to output the accumulator's value and a new line character.

The problem asks to find the shortest possible sequence of Deadfish commands to output an integer from the range [0, 255] given as the program's parameter.

- Resource-bounded search

# Picat - Planner Module - Deadfish Example

```
import planner.

final((N, N)) => true.

action((N, A), NewState, Action, Cost) ?=>
    NewState = (N, A + 1),
    Action = i,
    Cost = 1.
action((N, A), NewState, Action, Cost) ?=>
    NewState = (N, A * A),
    Action = s,
    Cost = 1.
action((N, A), NewState, Action, Cost) ?=>
    A > 0,
    NewState = (N, A - 1),
    Action = d,
    Cost = 1.

main([X]) =>
    N = X.to_integer(),
    best_plan((N, 0), Plan),
    printf("%w\n", Plan ++ [o]).
```

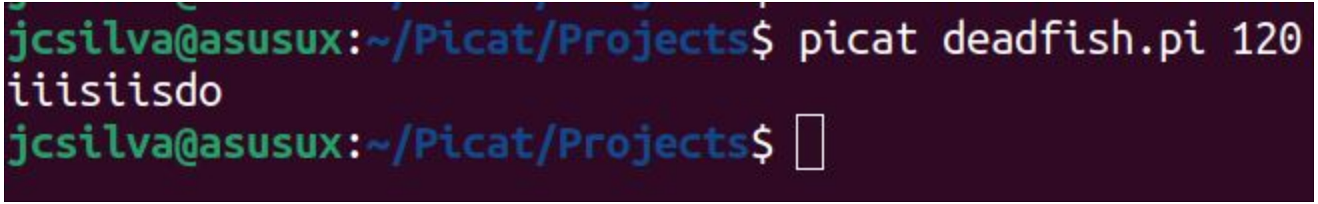
# Picat - Planner Module - Deadfish Example

The state representation for this problem is a pair (goal value, current value);

The state is final when the current value equals to the goal value;

The actions – i, s, and d – correspond to the Deadfish commands;

The main predicate gets the goal number from the command-line parameters, calls a two-parameter version of the best_plan (which assumes a very high resource limit – 268435455 – and doesn't return the best plan's cost), and prints the best plan plus the o command.



# Picat - Planner Module - Cannibals Example

Define an initial state, a final state, and a set of actions, and it tries to find a sequence of actions;

Example: The missionaries and cannibals problem is a classic AI planning problem.
Three missionaries and three cannibals come to the southern bank of a river and find a boat that holds up to two people. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. Find a plan to move them to the other bank of the river.

- Resource-unbounded search

# Picat - Cannibals Example

```
import planner.

main =>
    best_plan_unbounded([3,3,south],Plan),
    foreach (Step in Plan)
       println(Step)
    end.
```

```
final([3,3,north]) => true.

action([M,C,Bank],NextS,Action,Cost) =>
    member(BM,0..M),
    member(BC,0..C),
    BM+BC > 0, BM+BC =< 2,
    OppBank = opposite(Bank),
    Action = $cross(BM,BC,OppBank),
    Cost = 1,
    NewM1 = M-BM,
    NewC1 = C-BC,
    NewM2 = 3-NewM1,
    NewC2 = 3-NewC1,
    if NewM1 !== 0 then    % missionaries are safe
        NewM1 >= NewC1
    end,
    if NewM2 !== 0 then
       NewM2 >= NewC2
    end,
    NextS = [NewM2,NewC2,OppBank].

opposite(south) = north.
opposite(north) = south.
```

# Picat - Planner Module - Logistics Planning

Considering a weighted directed graph, a set of trucks, each of which has a capacity that indicates the maximum number of packages that the truck can carry, and a set of packages, each of which has an initial location and a destination. There are three types of actions: load a package onto a truck, unload a package from a truck, and move a truck from one location to a different location.

Each action has an associated cost. The objective of the problem is to find an optimal, minimum-cost plan to transport the packages from their initial locations to their destinations. The weighted directed graph is given by the predicate road(From,To,Cost), which succeeds if there is an edge from node From to node To that has a cost of Cost. An optimal plan for this problem normally requires trucks to cooperate. This problem degenerates into the shortest path problem if there is only one truck and only one package.

# Z3 - An efficient SMT solver

https://www.microsoft.com/en-us/research/project/z3-3/
https://github.com/z3prover/z3

Default input format is SMTLIB2

de Moura, L., Bjørner, N. (2008). Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008. Lecture Notes in Computer Science, vol 4963. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-78800-3_24

# Additional Resources - Pyomo

https://jckantor.github.io/ND-Pyomo-Cookbook

https://www.pyomo.org/

https://pyomo.readthedocs.io/en/stable/index.html

https://github.com/Pyomo

https://neos-guide.org/users-guide/third-party-interfaces/#pyomo

https://www.gams.com/blog/2023/07/performance-in-optimization-models-a-comparative-analysis-of-gams-pyomo-gurobipy-and-jump/

https://ebin.pub/qdownload/pyomo-optimization-modeling-in-python-3nbsped-3030689271-9783030689278-9783030689285.html

# Additional Resources - Gurobi

https://www.gurobi.com/

https://www.gurobi.com/resource-center/

https://www.gurobi.com/documentation/

# Additional Resources - Z3

https://www.microsoft.com/en-us/research/project/z3-3/

https://github.com/z3prover/z3

# Additional Resources - Picat

http://picat-lang.org/

http://www.hakank.org/picat/

https://buttondown.email/hillelwayne/archive/picat-is-my-favorite-new-toolbox-language/

http://sdymchenko.com/blog/2015/01/31/ai-planning-picat/

# Additional Resources - NEOS

https://neos-guide.org/users-guide/third-party-interfaces/

https://neos-guide.org/users-guide/third-party-interfaces/#pyomo

https://neos-server.org/neos/solvers/lp:Gurobi/AMPL.html