A brief overview of mathematical optimization

Paul Bouman

February 21, 2022

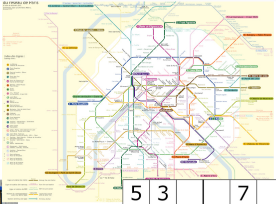
Winter Workshop on Complex Systems 2022, Arc-et-Senans, France

Table of contents

Introduction

Examples

What do these pictures have in common?



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

Attribution

Metro picture taken from Wikipedia

Timetable picture taken from Wikipedia

Examples

Three important ingredients:

- Coordinated/Centralized but **variable** decisions are needed
- Local or global rules **constrain** what combinations of decision are possible
- Optionally: **objective** it to make some measure of quality as *good as possible*

Optimization

Optimization

1. Try to find a **feasible solution** that is **as good as possible**

Optimization

Optimization

1. Try to find a **feasible solution** that is **as good as possible**
2. Provide reasoning why **no better solution can exist**

Optimization

Optimization

1. Try to find a **feasible solution** that is **as good as possible**
2. Provide reasoning why **no better solution can exist**

Pendantic people could say that there is *search* and that there is *optimization*

Optimization

Optimization

1. Try to find a **feasible solution** that is **as good as possible**
2. Provide reasoning why **no better solution can exist**

Pendantic people could say that there is *search* and that there is *optimization*

The distance between the best solution found and the proof of the best possible is called an **optimization gap**

Optimization

Many concepts and approaches exists

Concepts

- Logic Programming: variables are true/false, try to satisfy a formula
- Heuristics / Genetic Programming: start somewhere, iterate between small changes and accepting improvements
- Constraint Satisfaction: guess, then propagate over constraints to reduce variable domains, backtrack when stuck
- **Mathematical Programming**:

Flexibility

- Non-linear programming: all kinds of equations, local or global optimum?
- Convex Programming: allow equations such that local and global optimum coincide
- **Linear Programming**: only allow linear equations

Linear Programming

In my opinion *Linear Programming* is a bit of a *sweet spot*:

- good to decent interpretability of decision models
- surprisingly versatile decision modelling power
- decent opportunities for analytical and theoretical results
- high quality (commercial) software available

Models and Examples

(Integer) Linear Programming

- **Decision variables** with optional lower bound and upper bound, either **continuous** or **integer**. Variables that can be either 0 or 1 can model yes/no decisions and are called **binary** variables (special case of integer variables).
- **Objective function** to be minimize or maximized that is linear in terms of the variables
- **Constraints** that must be satisfied expressed using $\leq$, $=$ or $\geq$ where left and right hand side are linear in terms of the variables (or constant).

Basics

With decision variables x, y and z.

Allowed

$$\begin{aligned}
&\text{Minimize } 4x - 2y \\
&3.14x + 18y \geq 4 \\
&6(4x + 6y) \leq 2z \\
&4x = 6y
\end{aligned}$$

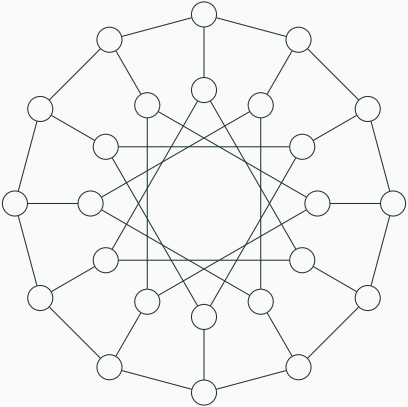
Not allowed

$$\begin{aligned}
&y(2x - 3z) \geq 2 \\
&y + 2x \neq z
\end{aligned}$$

However, there are many rewriting tricks. For example absolute values or the product of binary variables can often be dealt with.

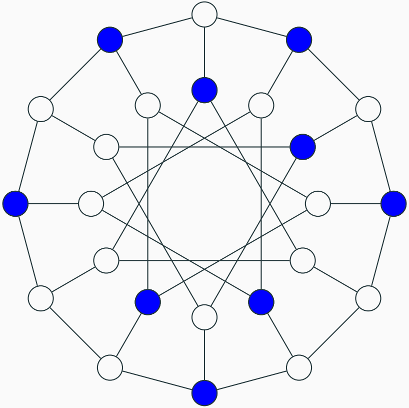
Example: Maximum Independent Set

Given a network, what is the maximum number of nodes we can choose such that we do not select two neighbours?



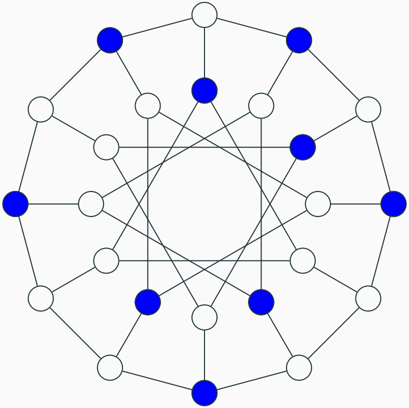
Example: Maximum Independent Set

Given a network, what is the maximum number of nodes we can choose such that we do not select two neighbours?



Example: Maximum Independent Set

Given a network, what is the maximum number of nodes we can choose such that we do not select two neighbours?



Possible application: where to plant trees that require distance

(nodes are locations, edges indicate conflicts)

Example: Maximum Independent Set

Notation

A graph $G = (V, E)$ where V contains nodes and E contains edges.

Variables: For each node v in V, introduce a binary variable x_v and let us say that

$$x_v = \begin{cases} 1 & \text{if v is selected} \\ 0 & \text{otherwise} \end{cases}$$

Objective:

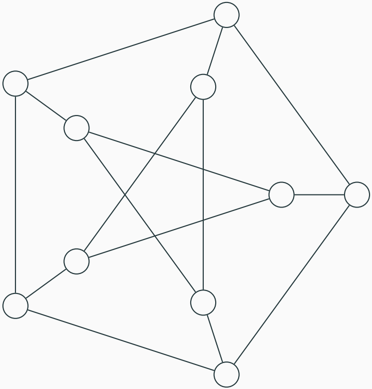
$$\text{Maximize} \quad \sum_{v \in V} x_v$$

Constraints: for each edge $(v, w) \in E$:

$$x_v + x_w \leq 1$$

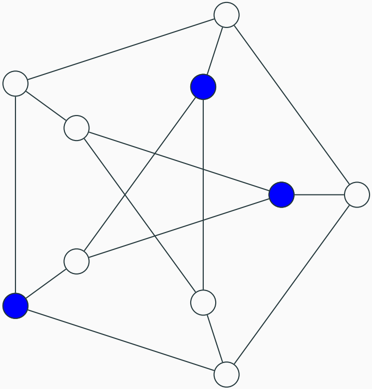
Exercise: Minimum Dominating Set

Choose the minimum number of nodes such that each node is either selected or has a selected neighbour



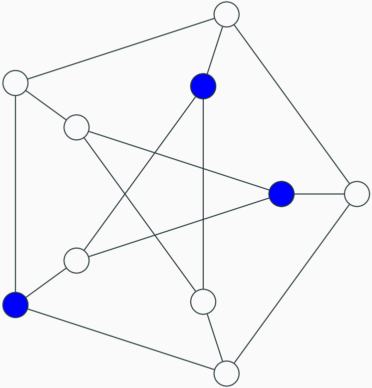
Exercise: Minimum Dominating Set

Choose the minimum number of nodes such that each node is either selected or has a selected neighbour



Exercise: Minimum Dominating Set

Choose the minimum number of nodes such that each node is either selected or has a selected neighbour

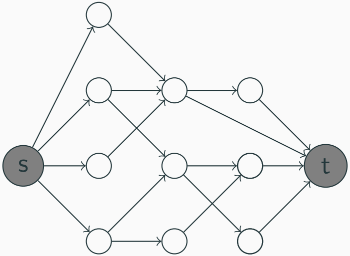


Possible application: where to locate ambulance depots

(how would you model this?)

Example: Minimum Cut

Remove minimum number of edges such that two given nodes are disconnected



Example: Minimum Cut

Notation

A directed graph $G = (V, A)$ where V contains nodes and A contains arcs.

Variables: for each arc $(v, w) \in A$ a variable x_{vw} and for each node $u \in V$ a variable z_u:

$$x_{vw} = \begin{cases} 1 & \text{if } (v, w) \text{ is cut} \\ 0 & \text{otherwise} \end{cases} \quad , \; z_u = \begin{cases} 1 & \text{if } v \text{ is connected to } s \\ 0 & \text{otherwise} \end{cases}$$

Objective:

$$\text{Minimize} \quad \sum_{(v,w)\in A} x_{vw}$$

Constraints: for each arc $(v, w) \in A$ for propagation of connectivity

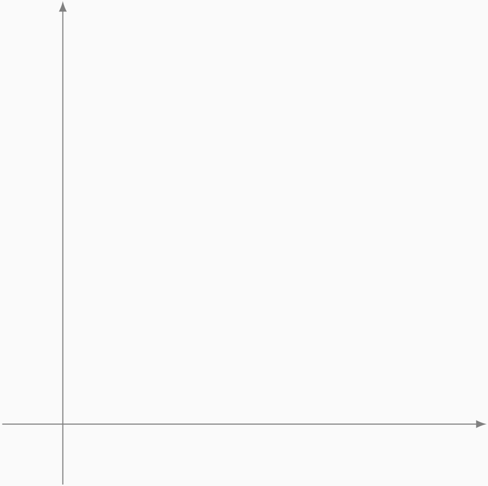
$$z_w \geq z_v - x_{vw}$$

and also fix the connectivity state of s and t:

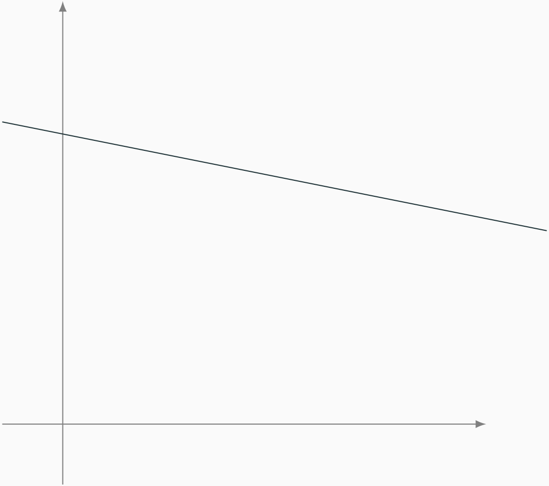
$$x_s = 1 \; , \; x_t = 0$$

How is it solved

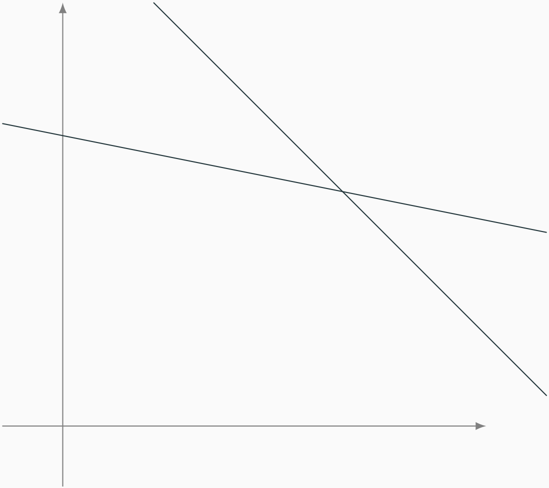
How to solve it



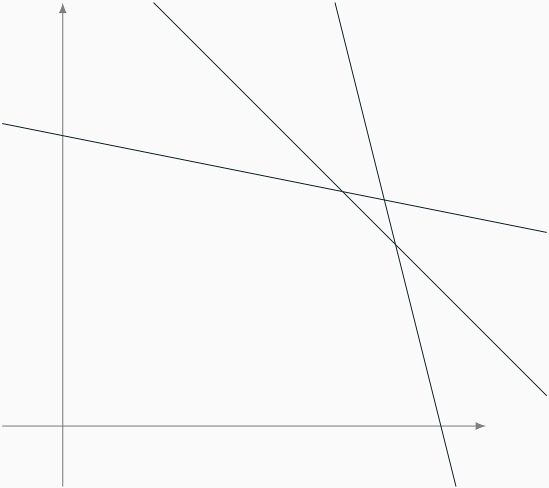
How to solve it



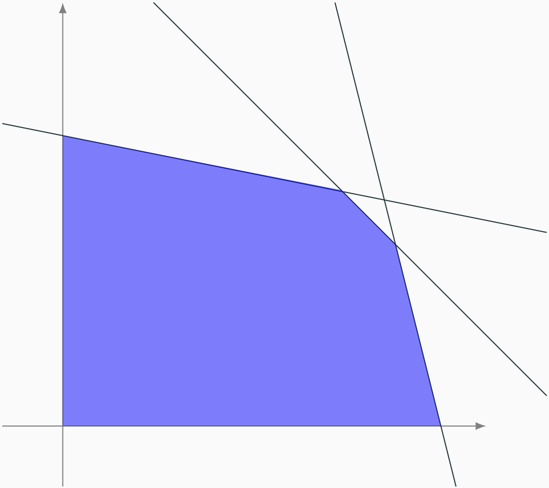
How to solve it



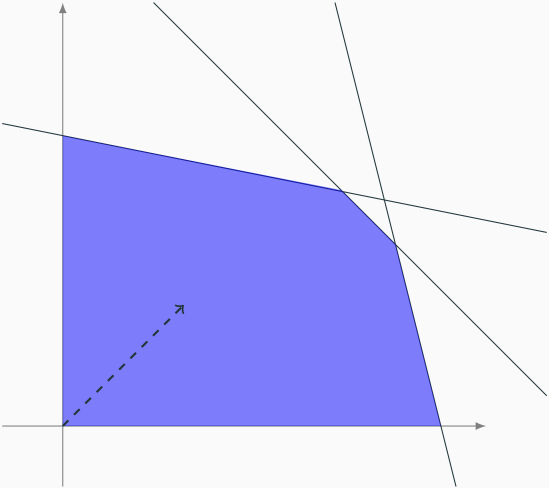
How to solve it



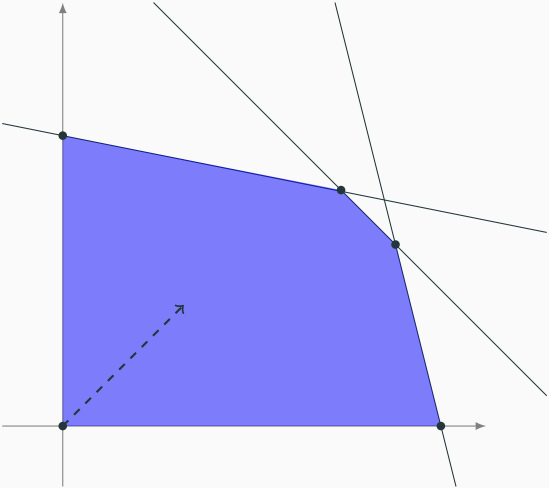
How to solve it



How to solve it



How to solve it



How to solve it

The **Simplex Algorithm** walks from the intersection points on the boundary of the feasible region in the direction of the objective function.

There are four cases

- The optimal solution is at a intersection point.
- The objective vector is orthogonal to a line segment (hyperplane): all the points on this line segment (hyperplane) are optimal.
- The feasible region is empty and you have no solution
- The feasible region is unbounded and the optimal solution is a *ray* rather than a point.

How to solve it

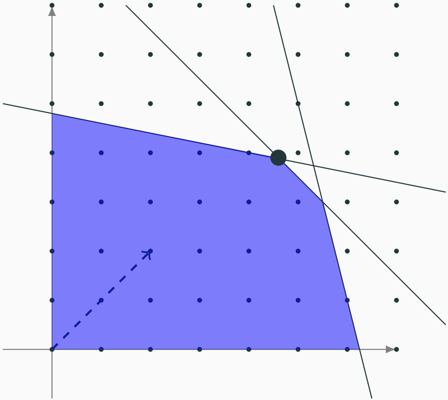
The **Simplex Algorithm** walks from the intersection points on the boundary of the feasible region in the direction of the objective function.

There are four cases

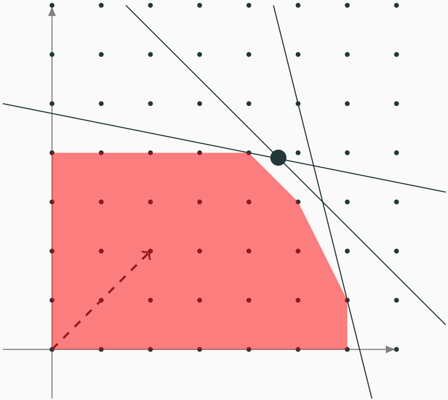
- The optimal solution is at a intersection point.
- The objective vector is orthogonal to a line segment (hyperplane): all the points on this line segment (hyperplane) are optimal.
- The feasible region is empty and you have no solution
- The feasible region is unbounded and the optimal solution is a *ray* rather than a point.

But what about integer solutions?

How to solve it - integer variables

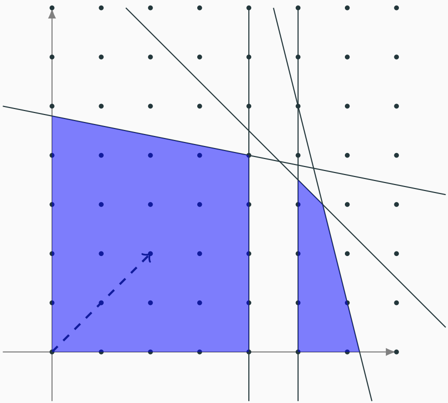


Simplex algorithm may give use fractional solutions, even if we want (some) variables to be integer.



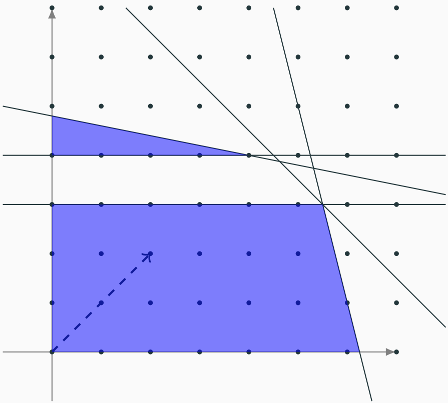
We do not know a good way to convert the fractional feasible space to an integer feasible space.

How to solve it - integer variables



We can start with a fractional solution, and then split up the feasible spaces into two subspaces, and take the best solution in the two smaller spaces. In this example we can either split on the x or the y variable

How to solve it - integer variables



We can start with a fractional solution, and then split up the feasible spaces into two subspaces, and take the best solution in the two smaller spaces. In this example we can either split on the x or the y variable

How to solve it - integer variables

Try to solve the non-integer version as explained before, and split the problem in two if your solution is fractional.

For every variable you have to split the number of subproblems doubles, so this could lead to an exponential amount of work in the number of variables.

However, you can eliminate subproblems where the optimal fractional solution is worse than the best solution *so far*, and only focus on sub-problems where there is actually a potential gap to close. This is called **branch-and-bound**.

The decision on which variable to split can have a huge impact on the amount of work that is needed. Commercial solvers have invested insane amounts of effort in developing tricks to do this in a clever way (among many other things).

Software

Solvers

There are many linear programming and integer linear programming solvers, but the quality varies wildly.

To create a good solver, there are many things to deal with:

- Exploit that many constraint matrices are very sparse (i.e. many constraints deal with a small number of variables)
- Numerical precision
- Strategies for branching/splitting to obtain integer solutions
- Strategies to obtain intermediate integer solutions that allow you to eliminate irrelevant parts of the search space
- Preprocessing of the model to make it more compact/easier to solve

Solvers

Since better optimization software can save large enterprises a lot of money, developers of commercial solvers invested a lot of effort to learn all kinds of mathematical and emperical tricks to make better solvers to tackle larger problems. Open source efforts were typically developed by volunteers working in Academia.

For integer programming, the 2015 version of Guribo is in benchmarks 800000 ($8 \cdot 10^5$) times faster[1] than the 1991 version of CPLEX, **on the same computer hardware**.

This means that a problem that solves in an hour with a state-of-the-art solver, could take more than 90 years to solve on a simple one.

There are cases where researchers incorrectly conclude linear programming is *intractable* based only on experiments with a simple solver.

[1] according to a presentation by one of it's creators

Solvers

Of course, it is good to be aware there is tension between *open science* and *closed source commercial packages* and *black box methods*. Whether this is a problem or not depends on what you try to achieve. Some points to consider

- The mathematical model that is solved is known (you define it) and it can be validated that a solution is actually correct according to the model independently.
- Generally, the overarching approach of these solvers is known, but the clever tricks they use to find solutions faster is not
- Good solvers provide a lot of ways to track and influence what they are doing while they are solving, which provides some level of transparency.

However, if you want to make your computational process easily reproducible for anyone, even companies and curious people not affiliated with a research institute, an open source solver may be a better choice.

Solvers

An overview of some interesting solvers for Linear Programming:

Solver	Models	Speed	Open Source	Academic	Commercial
COIN-OR Clp	LP	good	Yes	Free	Free
COIN-OR Cbc	ILP	ok	Yes	Free	Free
SCIP	ILP+	good	Yes	Free	Paid
CPLEX	ILP+	great	No	Free	Paid
Gurobi	ILP+	great	No	Free	Paid

LP: linear programming, ILP: integer linear programming,
ILP+: also support for some non-linear models.

CPLEX used to be state of the art, now Gurobi has the most active development. An academic license for Gurobi requires authentication from a campus network, for CPLEX you can just install and use it after creating an academic account.

Using a Solver

A good approach can be to use a package that is *solver-agnostic*. That means you build your optimization model with the package, and then can select which solver can be used. This way, you can easily switch between a free and a commercial solver (assuming both are available on your computer).

In Python, two popular options are Pyomo and PuLP. Pyomo is more abstract in modelling and providers abstract tools to transform your data into a model. PuLP is more low level and only allows you to define variables, constraints and the objective in a straightforward manner. For Julia, JuMP is a popular option.

The downside of using a solver-agnostic package is that you lose some access to the internals of a particular solver, but that is typically only a concern if you want to implement very advanced techniques.

Pyomo code example

```
from pyomo.environ import *

model = ConcreteModel()

# declare decision variables
x = Var(domain=Reals,bounds=(0,4))
y = Var(domain=Reals,bounds=(-1,1))
z = Var(domain=NonNegativeReals)

# declare objective
model.profit = Objective(expr = x + 4*y + 9*z, sense = minimize)

# declare constraints
model.c1 = Constraint(expr = x + y <= 5)
model.c2 = Constraint(expr = x + z >= 10)
model.c3 = Constraint(expr = -y + z == 7)

# solve
SolverFactory('cplex').solve(model)
print("objective=", model.profit())
```

PuLP code example

```
from pulp import *

prob = LpProblem("test1", LpMinimize)

# Variables
x = LpVariable("x", 0, 4) # 0 <= x <= 4
y = LpVariable("y", -1, 1) # -1 <= y <= 1
z = LpVariable("z", 0) # 0 <= z

# Objective (the name at the end is facultative)
prob += x + 4 * y + 9 * z, "obj"

# Constraints (the names at the end are facultative)
prob += x + y <= 5, "c1"
prob += x + z >= 10, "c2"
prob += -y + z == 7, "c3"

# Solve the problem using the default solver
prob.solve()  # use prob.solve(CPLEX()) instead to use CPLEX

# Print the value of the objective
print("objective=", value(prob.objective))
```

(Based on https://github.com/coin-or/pulp/blob/master/examples/test1.py)

Implementing Models

To construct models based on graphs, you typically want to create a list or a dictionary to hold variables and fill these based on data about nodes/edges.

For more examples and explanation, it makes sense to read the documentation:

- Pyomo documentation
- Pyomo cookbook
- PuLP documentation

Advanced Topics

Advanced topics

Some advanced topics not discussed in this tutorial:

- *Duality*: for an optimal solution to a continuous linear programming model there is an associated and closely connected *dual* solution, that provides interesting information about the sensitivity of your currect solution to changes in the constraints.
- *Column generation/Branch-and-Prize*: approaches where we consider a very large number of variables (e.g. one variable for every subset of nodes of a graph), but add only relevant variables iteratively based on information from the dual problem. Typically you do not need to enumerate all variables this way, but only consider a small number of them.
- *Constraint generation/Branch-and-Cut*: approaches where we consider a very large number of constraints (e.g. one constraint for every subset of nodes of a graph). When a solution is found, we try to find a violated constraint. If one is found, we add it. If no violated constraint can be found for the current solution, we know the solution is valid.

Conclusion

Conclusion

This tutorial gave a quick overview of (integer) linear programming.

- Provides an interesting mix between modelling versatility and high quality tools to tackle the models
- Simplicity of equations does provide opportunities to have some analytical results and helps with interpretability
- Commercial solvers are a lot faster, but free for academics (in particular if you have integer variables)
- Many packages that allow you to build models and swap which solver you use to solve them.

Thanks for your attention!

Solutions

Solution: Maximum Dominating Set

Notation

- A graph $G = (V, E)$ where V contains nodes and E contains edges.
- For a node v, $\delta(v)$ gives us the neigbours

Variables: For each node v in V, introduce a binary variable x_v and let us say that

$$x_v = \begin{cases} 1 & \text{if v is selected} \\ 0 & \text{otherwise} \end{cases}$$

Objective:

$$\text{Minimize} \quad \sum_{v \in V} x_v$$

Constraints: for each node $v \in V$:

$$x_v + \sum_{w \in \delta(v)} x_w \geq 1$$