Constraint Programming and Practical Optimization

Martin Fabian
Chalmers

2020

This document serves as the lecture notes for the CoPPo course. The document collects my own view and ideas about the interesting topic on constraint programming and practical optimization.

CoPPo is built upon selected chapters from the following references. These books overlap so that there are available different expositions of the same topic, which I think is good for learning.

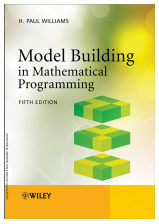
- [Williams, 2013], chapters 1.2, 3.3, 4.1, 5.2, 5.3, 8.3, 9.2, 9.3, 10.1. These parts form the basis of our treatment of modeling optimization problems as (mixed integer) linear programming problems.

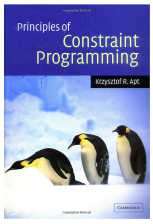
- [Apt, 2003], chapters 1, 3, 4, 5, 6, 7, 8. This is our main text on constraint programming.

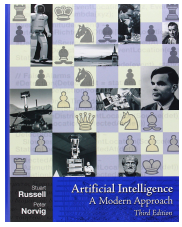
- [Russell and Norvig, 2009], Chapter 7 is a well-written and succinct description of constraint programming (a.k.a. constraint satisfaction problems). Of special importance is Chapter 7.5 that describes how structural properties of CSPs can be exploited to ease solving. Also of relevance is Chapter 3, "Solving Problems by Searching", and especially so Chapter 3.5, which covers informed searching; the authors Russell and Norvig invented A*, and the proof of its optimality is given there. In general, this is *the* book on artificial intelligence.

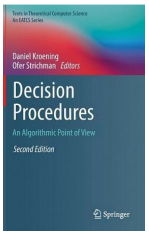
- [Kroening and Strichman, 2008], describes many of the algorithms that underlie constraint programming, and in general the procedure to decide whether a formulated problem is valid or not. In CoPPo we mainly use Chapter 1, but examples from the book will appear in the lectures. Chapter 5 gives a slightly different view, compared to [Williams, 2013], on "deciding" linear programming problems.

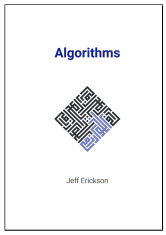
- [Erickson, 2020], Chapter 12 is the backbone for the lecture on computational complexity. This book also contains chapters on basic graph algorithms, depth-first search, and Dijkstra's and other shortest path algorithms (but not A*). The treatment is a bit computer science geeky, but not incomprehensible and well worth reading.











References

[Apt, 2003] Apt, K. (2003). *Principles of Constraint Programming*. Cambridge University Press.

[Erickson, 2020] Erickson, J. (2020). *Algorithms*. http://algorithms.wtf/.

[Hoeve, 2001] Hoeve, W.-J. v. (2001). The alldifferent constraint: A survey. Technical report, Cornell University, Department of Computer Science.

[Kroening and Strichman, 2008] Kroening, D. and Strichman, O. (2008). *Decision Procedures: An Algorithmic Point of View*. Springer Link, 1 edition.

[Russell and Norvig, 2009] Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Pearson Education, USA, 3rd edition.

[Strang, 2009] Strang, G. (2009). *Introduction to Linear Algebra*. Wellesley-Cambridge Press.

[Williams, 1978] Williams, H. (1978). The reformulation of two mixed integer programming problems. *Mathematical Programming*, 14:325–331.

[Williams, 2013] Williams, H. P. (2013). *Model Building in Mathematical Programming*. John Wiley & Sons, Incorporated.

Contents

EEN025: Constraint Programming and Applied Optimization **Fall 2020**

Lecture 1: Linear Programming I

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

Linear Programming
A mathematical model of solving practical problems (as the allocation of resources) by means of linear functions where the variables involved are subject to constraints.

www.meniam-webster.com

"Programming" as in "planning, scheduling or performing a program" (*not* as in computer programming)

"Linear" as in "constant-weighted sum of variables"

$$y = kx + m \quad \text{linear } (k, m \text{ are constants})$$
$$y = \frac{k}{x} + m \quad \underline{\text{not}} \text{ linear}$$

What does $y = kx + m$ look like? This is shown by Fig 1.1.

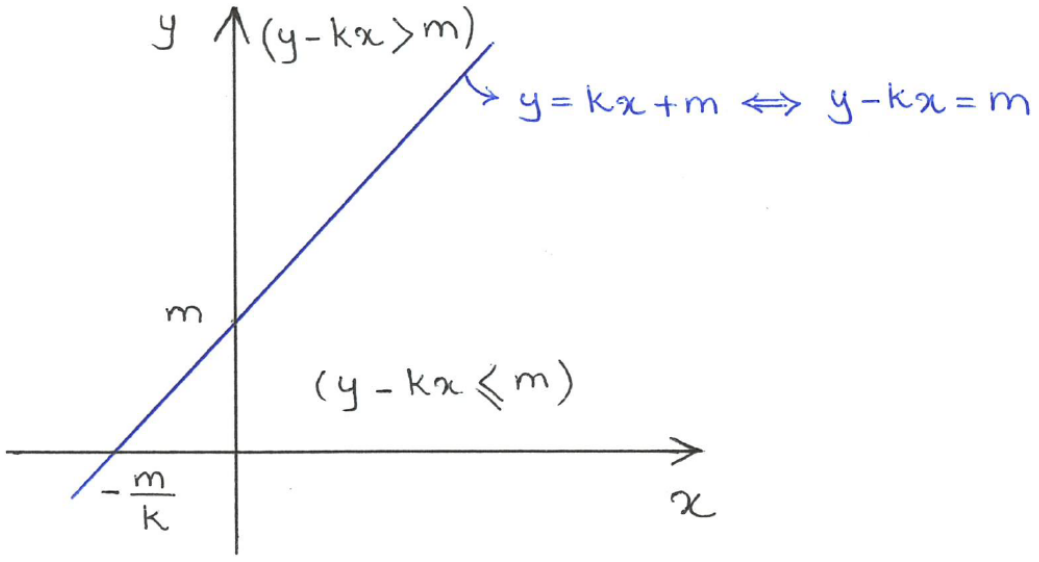


Figure 1.1: Linear function $y = kx + m$.

$$\begin{aligned} x = 0 \quad &\rightarrow \quad y = m \\ x = -\frac{m}{k} \quad &\rightarrow \quad y = 0 \end{aligned}$$

k gives the gradient of the line

m determines where the line crosses the y axis

1.1 LP - problem

Find min/max of linear objective function subject to linear variable constraints.

Example: Can assemble two types of products, A and B. Advances on respective product:

$$\begin{aligned} A &: €12 \\ B &: €9 \end{aligned}$$

Requirements for assembly:

$$\begin{aligned} A: 1 \quad &\text{of piece} \quad A_1 \\ 1 \quad &\text{of piece} \quad C \\ 4 \quad &\text{of piece} \quad D \\ B: 1 \quad &\text{of piece} \quad B_1 \\ 1 \quad &\text{of piece} \quad C \\ 2 \quad &\text{of piece} \quad D \end{aligned}$$

Problem: Maximize profit, given that the current stock is:

$$\begin{aligned} 1000 \quad &A_1 \\ 1500 \quad &B_1 \\ 1750 \quad &C \\ 4800 \quad &D \end{aligned}$$

Problem formulation

(x_1 is number of A, x_2 is number of B produced)

$$\text{objective function} \begin{cases} \max \quad 12x_1 + 9x_2 \end{cases}$$

$$\text{constraints} \begin{cases} \text{given} \quad x_1 \leq 1000 \quad \text{number of } A_1 \text{ available} \\ \qquad\quad x_2 \leq 1500 \quad \text{number of } B_1 \text{ available} \\ \qquad\quad x_1 + x_2 \leq 1750 \quad \text{number of } C \text{ available} \\ \qquad\quad 4x_1 + 2x_2 \leq 4800 \quad \text{number of } D \text{ available} \\ \qquad\quad x_1, x_2 \geq 0 \quad \text{cannot produce negative numbers} \end{cases}$$

How do we solve this?

For two-dimensional problems we can do it graphically. This is shown in Fig 1.2.

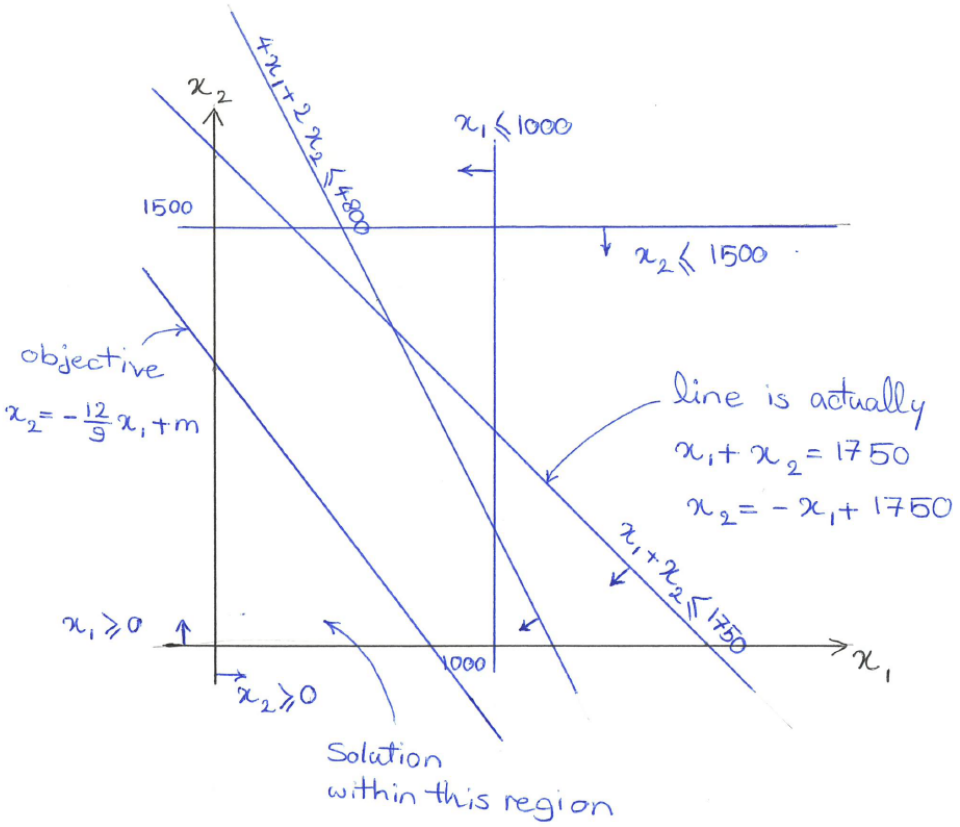


Figure 1.2: The constraints and the objective function displayed graphically.

We are looking for (x_1^*, x_2^*) such that:

$$12x_1^* + 9x_2^* = \max \quad \text{given the constraints}$$

We have

$$ax + by = m' \quad \text{where } m' \text{ is unknown}$$

Then

$$y = -\frac{a}{b}x + m \quad k \text{ is known, } k = -\frac{a}{b}$$

we are only searching for m.

Lets look at Fig 1.3.

$$m = y + kx$$

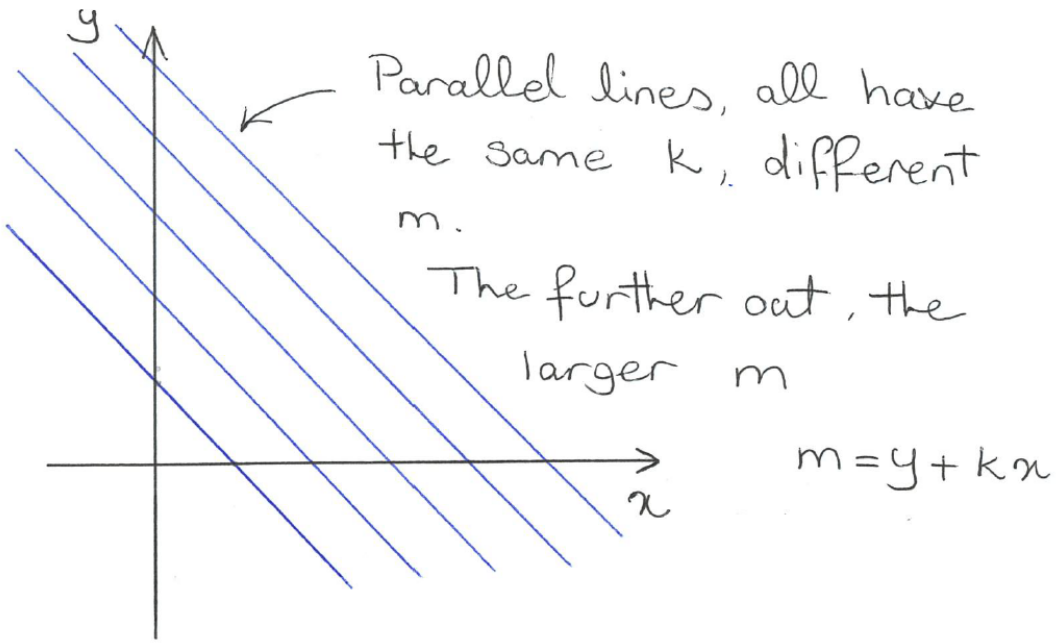


Figure 1.3: Parallel lines all have the same k, but different m.

Want to find the (largest) m that gives us a single point still within the constraints.

How?

This is shown in Fig 1.4.

(x_1^*, x_2^*) is the intersection between

$$4x_1 + 2x_2 = 4800 \quad \text{and} \quad x_1 + x_2 = 1750$$

The corresponding graph is shown in Fig 1.5.

Intersection occurs when $k_1 x_1^* + m_1 = k_2 x_2^* + m_2$, that is, when $x_1^* = x_2^*$:

$$x_1^* = \frac{m_1 - m_2}{k_2 - k_1}.$$

In the example:

$$\begin{aligned} x_1^* &= \frac{1750 - 2400}{(-2) - (-1)} = 650 \\ x_2^* &= 1100 \end{aligned}$$

Thus, producing 650 A and 1100 B will maximize profits to €17700.

GREAT!!

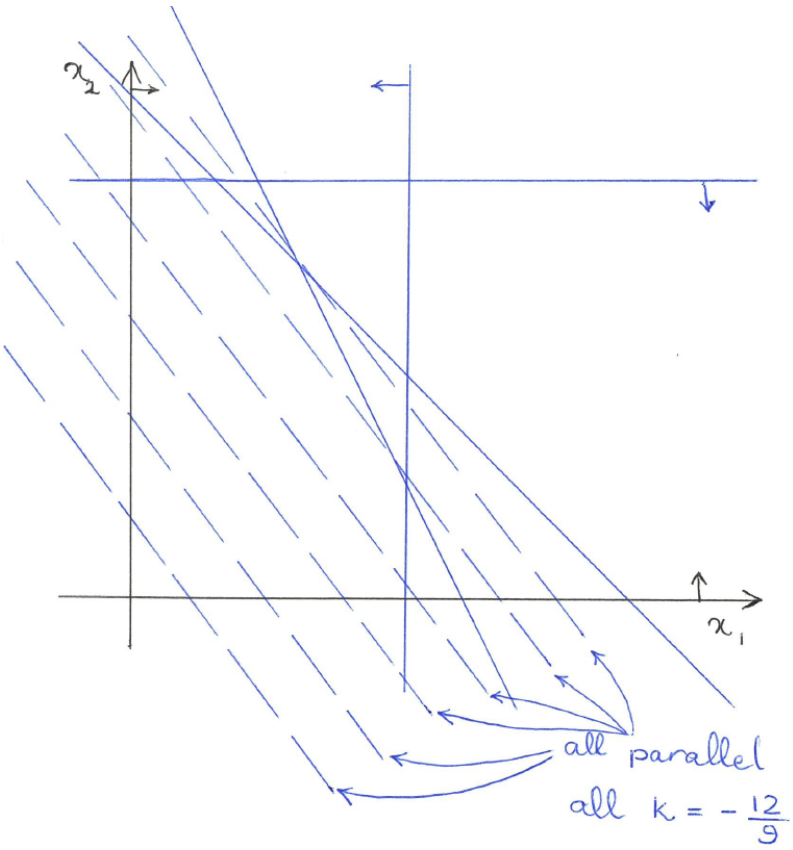


Figure 1.4: Parallel lines for the objective function with increasing m.

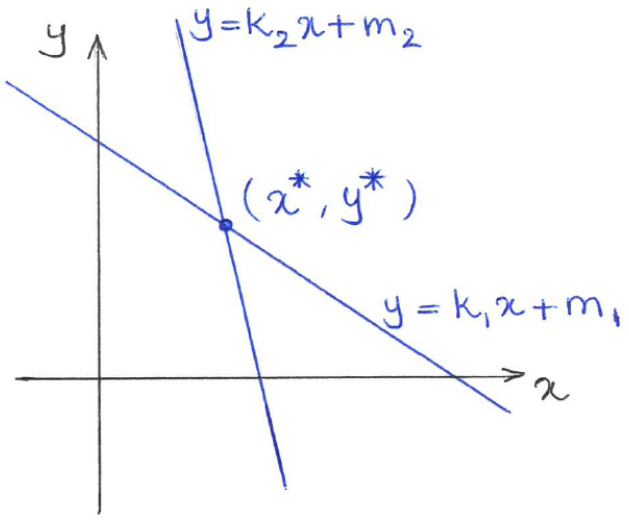


Figure 1.5: Intersection between the two constraints that give the optimum.

EEN025: Constraint Programming and Applied Optimization **Fall 2020**

Lecture 2: Linear programming II

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

2.1 LP general

$$\max \text{(or min)} \quad \sum_{i=1}^{n} c_i x_i \quad (c_i \text{ constants})$$

Subject to:

$$\begin{aligned}
(\sum_{i=1}^{n} a_i^1 x_i) &\leq b_1 \quad (a_i^j,\, b_j \text{ constants}) \\
(\sum_{i=1}^{n} a_i^2 x_i) &\leq b_2 \\
&\vdots \\
(\sum_{i=1}^{n} a_i^m x_i) &\leq b_m \\
x_i &\geq 0
\end{aligned}$$

where m is the number of constraints, and n is the number of decision variables.

This is more conveniently expressed in matrix form:

$$\begin{aligned}
\max \quad & c^T x \\
\text{subject to} \quad & Ax \leq b \\
& x \geq 0
\end{aligned}$$

where A is an $m \times n$ matrix, b is $1 \times m$, and c and x are $1 \times n$ vectors.

For the example:

$$\max \begin{bmatrix} 12 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 4 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 1000 \\ 1500 \\ 1750 \\ 4800 \end{bmatrix}$$

By convention, the so-called *standard form*:

$$\begin{aligned}
\min \quad & c^T x \\
\text{subject to} \quad & Ax = b \\
& x \geq 0
\end{aligned}$$

We will soon see how we can transform any LP problem into this form.

2.2 LP, what is it we want to do?

Let us call the objective function p, it is a function $p : X \to \mathcal{R}$, where X is the set of decision variables x_i. Let D be the domains for the decision variables, and let C be set of constraints. Then we can define a problems as:

$$\langle X, D, C, p \rangle.$$

For this problem, we want to find an *assignment* to the decision variables X, call it x^*, such that:

- they are all within their respective domain, $D(x^*)$ is true;
- all constraints are satisfied, $C(x^*)$ is true; and
- $p(x^*)$ is optimal.

We call x^* an *optimal satisfiable assignment*, and this is then a *satisfiability* problem.

This problem can be:

Infeasible – no satisfiable assignment exists, the *feasible set* spanned by the constraints and domains is empty. For instance, conflicting constraints such as $C = \{x_1 + x_2 \leq 2, -x_1 - x_2 \leq 3\}$.

Unbounded – no *bounded* solution exists, we can find arbitrary good values for the objective function. This happens when the constraints are not closed so the solution is not "locked in". The example:

$$\begin{aligned} \max \quad & x_1 + 3x_2 \\ \text{s.t.} \quad & x_1 + x_2 \geq 4 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{2.1}$$

is depicted in Fig 2.1. Note if we were to minimize instead of maximize, then a bounded solution would exist.

Feasible – a bounded solution does exist. In that case, the solution can be either:

Unique – and then it appears at an *extreme point*, an intersection between constraints; or

Non-unique – and there are infinitely many equally good solutions on a constraint boundary. Then the objective function is "parallel" to some constraint. Note though that this includes at least one solution at an extreme point. See Fig 2.2.

A note about constraints. Suppose we have problem with a single constraint $C = \{x_1 + x_2 \leq 4\}$ and $D = \{x_1 \geq 0, x_2 \geq 0\}$. The feasible set then looks like Fig 2.3. Adding another constraint, say $5x_1 + 3x_2 \leq 15$, what happens to the feasible set? See Fig 2.4.

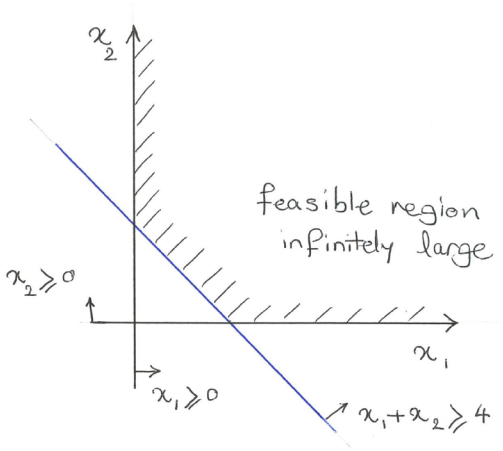


Figure 2.1: Unbounded problem.

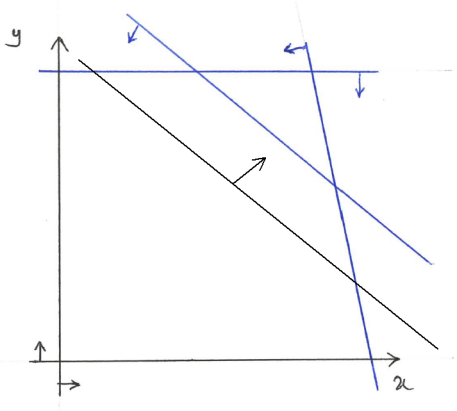


Figure 2.2: Non-unique solution, the objective function is parallel to the diagonal constraint.

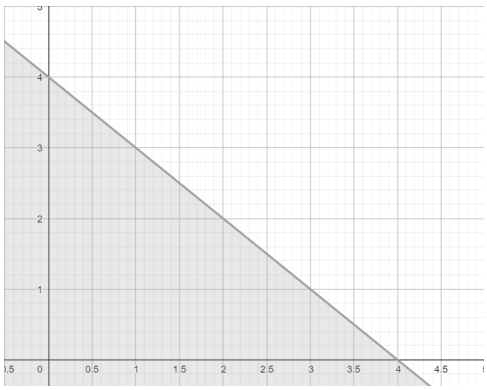


Figure 2.3: Feasible set for $C = \{x_1 + x_2 \leq 4\}$ and $D = \{x_1 \geq 0,\, x_2 \geq 0\}$.

Figure 2.4: Adding extra constraint shrinks the feasible set.

2.3 The Fundamental Theorem of LP

Convex Set: A set is *convex* if line segments between any pair of points within the set fall entirely within the set.

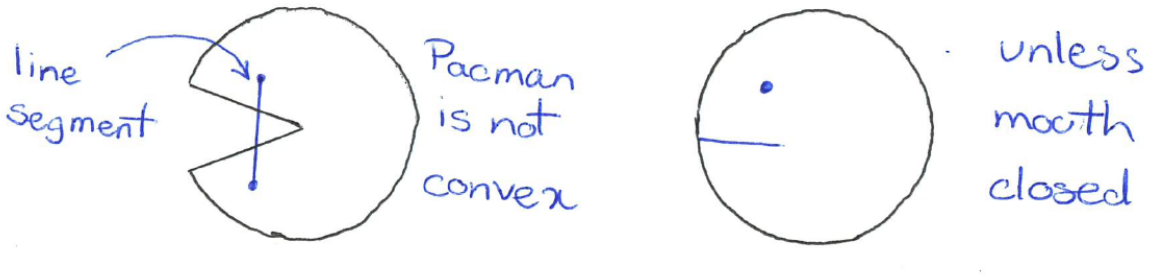


Figure 2.5: Non-convex and convex set.

Mathematically: A set X is convex if for any $x_1, x_2 \in X$ and any $0 \leq \delta \leq 1$, $x_1\delta + x_2(1-\delta) \in X$.

Known fact: If all constraints C of an optimization problem are linear, then the feasible set is convex.

This is good!

A function $f(x)$ is *unimodal* (over an interval) if there exists an m (the mode) such that $f(x)$ strictly (monotonically) increases for $x < m$ and strictly decreases for $x > m$ (over the interval).

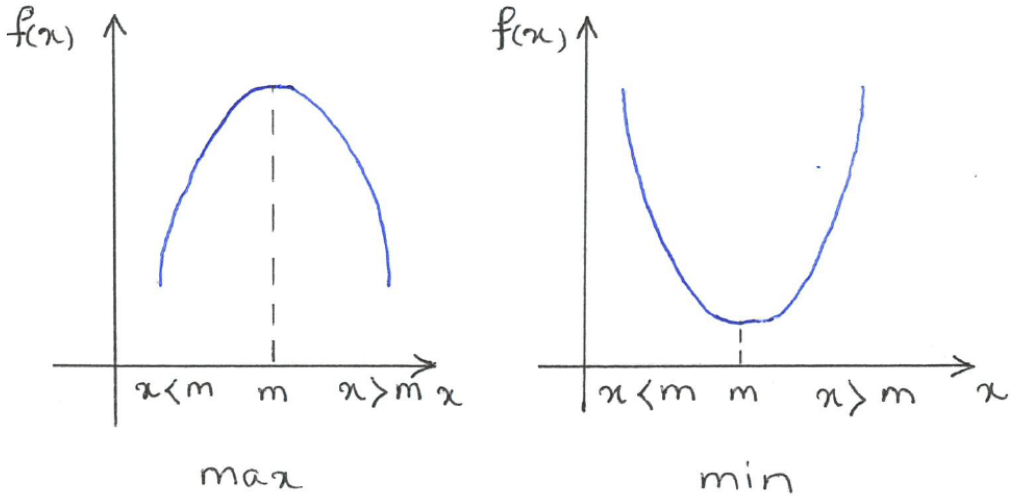


Figure 2.6: Unimodal functions.

Then m is the max of $f(x)$ and no other local maximum exist. Similarly (depending on context) for min.

Known fact: Linear objective functions are unimodal both in *max* and *min*.

This is good!

If for an optimization problem

- the objective function is unimodal, and
- the constraints define a convex feasible set, then

every local optimum is also a global optimum. Thus, for LP problems a local optimum is also *the* global optimum.

This is *very* good!!

Theorem 2.1 *If an optimal solution exists, then it lies at an extreme point.*

The extremal values of a linear function over a convex feasible set are attained at the corners of the set. If an extremum is attached to two corners, then it is attained everywhere on the "line" that connects them.

For a real proof of this, see for instance https://en.wikipedia.org/wiki/Fundamental_theorem_of_linear_programming; here we just give a sketch.

Lets us consider a minimization problem, with the optimal solution $c^T x^*$. Assume that x^* is not on the boundary of the feasible set, but inside it. Then there exists an $\epsilon > 0$ such that $x^* - \epsilon$ is still inside the feasible set. Then, $c^T(x^* - \epsilon) = c^T x^* - c^T \epsilon$, which is smaller than $c^T x^*$, and thus x^* was not the optimal solution. Thus, x^* must lie on the boundary of the feasible set.

Assume that x^* is not on a corner. Then it can be written as a linear combination of adjacent corners, $x^* = \sum \lambda_i x_i$, with $\lambda_i > 0$ and $\sum \lambda_i = 1$. Then of course, $(\sum \lambda_i x_i) - x^* = 0$. Since the λ_i sum to 1, we can rewrite this as $\sum \lambda_i (x_i - x^*)$, and then we have that $c^T \sum \lambda_i (x_i - x^*) = 0$. Now we can multiply in the c^T, so we get $\sum \lambda_i (c^T x_i - c^T x^*)$ which is 0 as earlier. Since $c^T x^*$ is the minimum, $(c^T x_i - c^T x^*)$ must be non-negative. Then, for the whole expression to be 0, all of the $(c^T x_i - c^T x^*)$ must be 0, hence all x_i and all points on the "line" between x_i and x^* are optimal solutions.

Lecture 3: Linear programming III

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

3.1 LP, standard form

$$\begin{aligned} \max \quad & 12x_1 + 9x_2 \\ \text{subject to} \quad & x_1 \leq 1000 \\ & x_2 \leq 1500 \\ & x_1 + x_2 \leq 1750 \\ & 4x_1 + 2x_2 \leq 4800 \\ & x_1 x_2 \geq 0 \end{aligned} \tag{3.1}$$

This we have seen and understood.

The standard form looks like this:

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{3.2}$$

where A is an $m \times n$ matrix, b is $1 \times m$ and $b \geq 0$, and c and x are $1 \times n$ vectors.

How do we get to the standard form from the other?

- To make $b \geq 0$ is trivial
- Then we need to change max to min. Observe: if $x \geq 0$ then $-x \leq 0$.

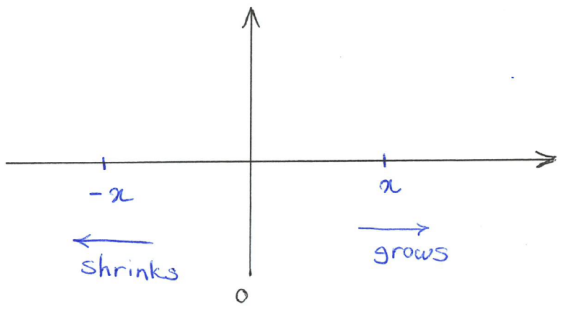


Figure 3.1: The interplay between x and $-x$.

$$\max\ p(x) \Longleftrightarrow \min\ -p(x)$$

So, for the example: $\max\ 12\,x_1 + 9\,x_2 \Longleftrightarrow \min\ -12\,x_1 - 9\,x_2$.

• And then we need to change the inequalities ($\leq$ or $\geq$) to equality ($=$).

Lets look at the constraints of (3.1):

$$\begin{aligned} x_1 &\leq 1000 \\ x_2 &\leq 1500 \\ x_1 + x_2 &\leq 1750 \\ 4x_1 + 2x_2 &\leq 4800 \\ x_1, x_2 &\geq 0 \end{aligned}$$

In the four constraints (the domain constraint remains as it is), we want $=$ where we now have $\leq$.

How? Introduce extra variables!

$$\begin{aligned} x_1 + x_3 &= 1000 \\ x_2 + x_4 &= 1500 \\ x_1 + x_2 + x_5 &= 1750 \\ 4x_1 + 2x_2 + x_6 &= 4800 \end{aligned}$$

$x_3, \ldots, x_6$ are *slack* variables.

Note: slack variables represent:

$$\begin{aligned} x_3 &= 1000 - x_1 \\ x_4 &= 1500 - x_2 \\ x_5 &= 1750 - x_1 - x_2 \\ x_6 &= 4800 - 4x_1 - 2x_2 \end{aligned} \tag{3.3}$$

so, for instance,

with $x_3 = 0$, $x_1 = 1000$, we are on the constraint boundary;
with $x_3 > 0$, $x_1 < 1000$, we are inside the boundary;
with $x_3 < 0$, $x_1 > 1000$, we are outside the boundary.

Thus, the non-negative requirement must hold also for the slack variables.

But what if the constraints are $\geq$, as in $x + 2y \geq 4$?

Then, we add *surplus* variables, like $x + 2y - w = 4$. We can rewrite this as $w = x + 2y - 4$. Again:

with $w = 0$, $x + 2y = 4$, we are on the constraint boundary;

with $w > 0$, $x + 2y > 4$, we are inside the boundary;

with $w < 0$, $x + 2y < 4$, we are outside the boundary.

Thus, the non-negative requirement must hold also for the surplus variables.

For the optimal solution x^* of our running problem (3.1):

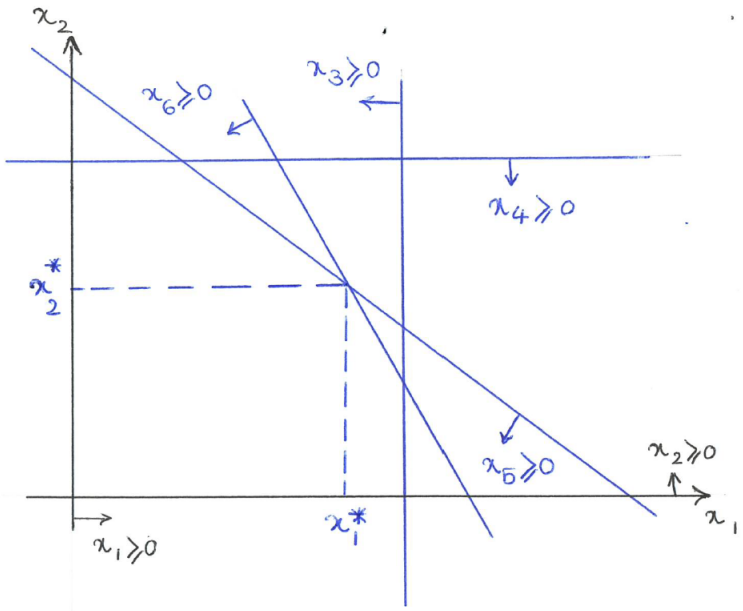


Figure 3.2: x^* and slack variables for (3.1).

$$\begin{aligned} x_3^* &= -x_1^* + 1000 \quad &= 350 \\ x_4^* &= -x_2^* + 1500 \quad &= 400 \\ x_5^* &= -x_1^* - x_2^* + 1750 \quad &= 0 \\ x_6^* &= -4x_1^* - 2x_2^* + 4800 \quad &= 0 \end{aligned} \tag{3.4}$$

Why are x_5^* and x_6^* in (3.4) equal to zero? See Fig 3.2 and Fig 3.3.

(3.1) in matrix form:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 4 & 2 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 1000 \\ 1500 \\ 1750 \\ 4800 \end{bmatrix}$$

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{bmatrix}^T \geq 0$$

With the objective function:

$$\begin{bmatrix} 12 & 9 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{bmatrix}^T$$

Note: A is *not square*, the solution is not simply $x = A^{-1}b$.

Note: Matrix A contains the identity matrix for the slacks x_3, x_4, x_5, x_6.

3.2 The Dual Problem

For the problem in standard form, there exists its *dual* problem:

$$\begin{aligned} \max \quad & b^T y \\ \text{subject to} \quad & A^T y = c \\ & y \geq 0 \end{aligned} \tag{3.5}$$

For our example (3.1), its dual problem looks like:

$$\begin{array}{lrcrcrcl} \min & 1000\,y_1 & + & 1500\,y_2 & + & 1750\,y_3 & + & 4800\,y_4 \\ \text{s.t.} & y_1 & + & & & y_3 & + & 4\,y_4 \geq 12 \\ & & & y_2 & + & y_3 & + & 2\,y_4 \geq 9 \\ & & & & & y_1, y_2, y_3, y_4 & \geq & 0 \end{array} \tag{3.6}$$

Solving it we get $y^* = \langle 0, 0, 6, 1.5 \rangle$, $p^* = 17700$.

We can note a few things.

- The dual of the dual problem is the original problem, that is obvious, not very exciting.
- The dual problem has m decision variables, and n constraints. This means that the dual problem may be easier to solve than the original problem (probably not the case here, since the number of decision variables is often an indicator of how hard an LP problem is to solve).
- The optimal value is the same for the dual problem as for the original problem. This is always the case. In fact, for any feasible vectors x and y, if $c^T x = b^T y$ then x and y are optimal!
- If either the original problem or the dual problem has an optimal solution, then so has the other, and the values are the same. If one problem does not have an optimal solution, then either both feasible sets are empty, or one feasible set is empty and the other problem is unbounded.
- In the dual problem the decision variables relate to the constraints of the original problem. If a constraint is not "binding", meaning it is not at its limit, in the optimal solution for the original problem, then the decision variable representing that constraint in the dual problem will be zero.

In economics the dual problem is very important, as it contains a considerable amount of extra economic information, things like "shadow prices" (see Chapter 6.2 of [Williams, 2013]).

3.3 Some notes on $Ax = b$

We have in the standard form the expression $Ax = b$, with $b \geq 0$, and A being $m \times n$ with $n \geq m$.

We want to find values of x for which this expression holds.

If A was invertible, we could solve this as $x = A^{-1}b$, but A is not even square.

From matrix algebra, we know that if we can find in A an $m \times m$ invertible submatrix, call it B, then $x = B^{-1}b$ is a solution to $Ax = b$.

There may be many such B.

Let us write A as $\begin{bmatrix} B & N \end{bmatrix}$, then:

$$Ax = \begin{bmatrix} B & N \end{bmatrix} \begin{bmatrix} x_B \\ x_N \end{bmatrix} = Bx_B + Nx_N = b$$

Assume now that $x_B = B^{-1}b$, then:

$$Ax = \begin{bmatrix} B & N \end{bmatrix} \begin{bmatrix} B^{-1}b \\ x_N \end{bmatrix} = BB^{-1}b + Nx_N = b + Nx_N = b$$

Thus, when $x_B = B^{-1}b$, then $N\,x_N = 0$.

The variables in x_B are called *basic*, and B is a *basic solution*.

Each basic solution corresponds to an extreme point.

Not all basic solutions are feasible. But...

If an LP problem has a bounded optimal solution, then it has an optimal basic feasible solution

We will soon see how we can exploit this fact to efficiently find an optimal solution

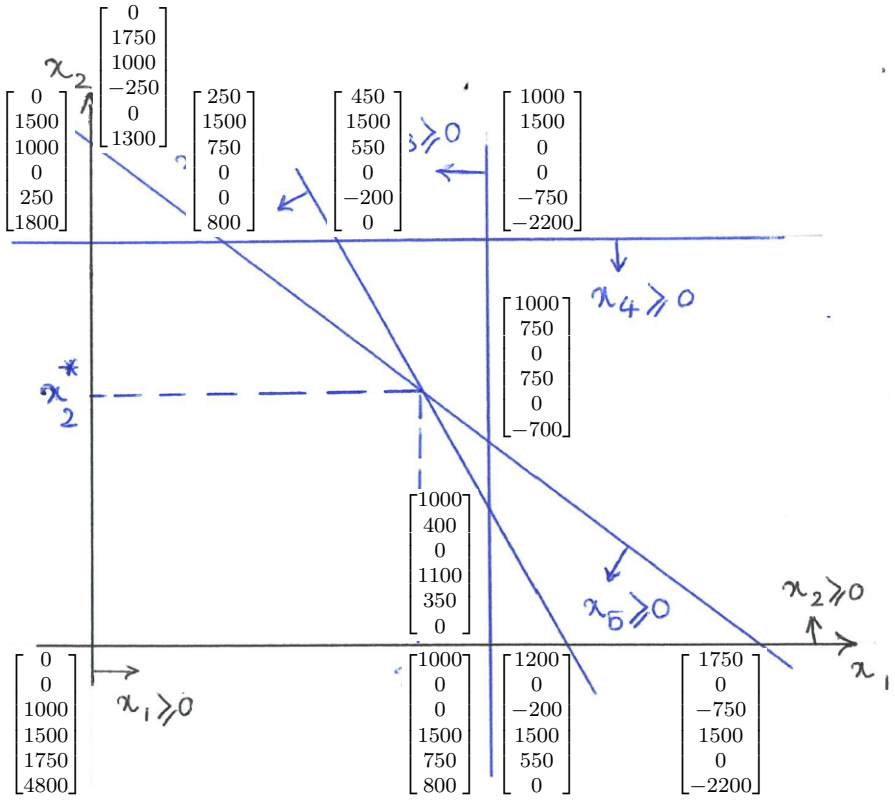


Figure 3.3: x values at extreme points of (3.3).

Lecture 4: Linear Programming IV

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

4.1 Simplex algorithm

We know the solution (if it exists) lies on a extreme point. We could calculate all extreme points and check which one is optimal. In principle...

In practice there may be *billions* of extreme points.

The number of extreme points is:

$$\frac{(n+m)!}{n!\,m!},$$

for n variables and m constraints.

Start at one extreme point, for instance (0,0).

We know the objective function is unimodal, so if we move in a direction that improves the objective function, we will not find a worse value for the objective function at the extreme point in that direction. If no such direction exist, we are done!

The algorithm turns this into matrix calculations to find B, but it can also be done by hand.

4.2 Simplex by hand

Consider the running problem (3.1), with slacks:

$$\begin{array}{lrcrcrcrcrcl} \min & -12\,x_1 & - & 9\,x_2 & + & 0\,s_1 & + & 0\,s_2 & + & 0\,s_3 & + & 0\,s_4 & \\ \text{s.t.} & x_1 & & & + & s_1 & & & & & & = 1000 \\ & & & x_2 & + & & + & s_2 & & & & = 1500 \\ & x_1 & + & x_2 & & & & & + & s_3 & & = 1750 \\ & 4\,x_1 & + & 2\,x_2 & + & & & & & & + & s_4 = 4800 \\ & & & & & & & & & & & x_i, s_i \geq 0 \end{array} \tag{4.1}$$

Rewrite the constraints in terms of the slacks:

$$\begin{aligned} s_1 &= 1000 - x_1 \\ s_2 &= 1500 - x_2 \\ s_3 &= 1750 - x_1 - x_2 \\ s_4 &= 4800 - 4\,x_1 - 2\,x_2 \end{aligned} \tag{4.2}$$

With $x_1 = x_2 = 0$, we get $p = 0$, is this optimal?

No! Increasing either x_1 or x_2 will drive p down. Let us examine what happens if we increase x_1.

Why x_1? Because it affects p the most (its factor is -12, instead of -9 for x_2).

Examine (4.2) with $x_2 = 0$:

$$\begin{aligned} s_1 &= 1000 - x_1 && \Rightarrow x_1 \le 1000 \\ s_2 &= 1500 \\ s_3 &= 1750 - x_1 && \Rightarrow x_1 \le 1750 \\ s_4 &= 4800 - 4\,x_1 && \Rightarrow x_1 \le \tfrac{4800}{4} = 1200 \end{aligned}$$

x_1 is most constrained by $s_1 = 1000 - x_1$. So we "take out" s_1 (make it non-basic), and we "put in" x_1 (make it basic):

$$x_1 = 1000 - s_1.$$

Now we rewrite the whole problem in terms of x_1, s_2, s_3, s_4, from (4.2):

$$\begin{aligned} x_1 &= 1000 - s_1 \\ s_2 &= 1500 - x_2 \\ s_3 &= 1750 - (1000 - s_1) - x_2 && = 750 + s_1 - x_2 \\ s_4 &= 4800 - 4\,(1000 - s_1) - 2\,x_2 && = 800 + 4\,s_1 - 2\,x_2 \\ p &= -12\,(1000 - s_1) - 9\,x_2 && = -12000 + 12\,s_1 - 9\,x_2 \end{aligned} \tag{4.3}$$

So, currently with $x_2 = s_1 = 0$, $p = -12000$, is this the optimal.

No! Increasing x_2 will drive p further down.

Examine (4.3) with $s_1 = 0$:

$$\begin{aligned} x_1 &= 1000 \\ s_2 &= 1500 - x_2 && \Rightarrow x_2 \le 1500 \\ s_3 &= 1750 - x_2 && \Rightarrow x_2 \le 1750 \\ s_4 &= 800 - 2\,x_2 && \Rightarrow x_2 \le 400 \end{aligned}$$

x_2 is most constrained by $s_4 = 800 - 2\,x_2$. So we take out s_4, put in x_2:

$$x_2 = 400 + 2\,s_1 - \tfrac{1}{2}s_4$$

Rewrite the problem in terms of x_1, x_2, s_2, s_3, from (4.3):

$$\begin{aligned} x_1 &= 1000 - s_1 \\ x_2 &= 400 + 2\,s_1 - \tfrac{1}{2}s_4 \\ s_2 &= 1500 - (400 + 2\,s_1 - \tfrac{1}{2}s_4) && = 1100 - 2\,s_1 + \tfrac{1}{2}s_4 \\ s_3 &= 750 + s_1 - (400 + 2\,s_1 - \tfrac{1}{2}s_4) && = 350 - s_1 + \tfrac{1}{2}s_4 \\ p &= -12000 - 9\,(400 + 2\,s_1 - \tfrac{1}{2}s_4) + 12\,s_1 && = -15600 - 6\,s_1 + \tfrac{9}{2}s_4 \end{aligned} \tag{4.4}$$

So, currently with $s_1 = s_4 = 0$, $p = -15600$. Is this the optimal?

No! Increasing s_1 will drive p further down.

Examine (4.4) with $s_4 = 0$:

$$\begin{aligned} x_1 &= 1000 - s_1 && \Rightarrow s_1 \le 1000 \\ x_2 &= 400 + 2\,s_1 \\ s_2 &= 1100 - 2\,s_1 && \Rightarrow s_1 \le 550 \\ s_3 &= 350 - s_1 && \Rightarrow s_1 \le 350 \end{aligned}$$

s_1 is most constrained by s_3. So we take out s_3, and put in $s_1 = 350 - s_3 + \frac{1}{2}s_4$.

Rewrite the problem (4.4) in terms of x_1, x_2, s_1, s_2, we get:

$$\begin{aligned} x_1 &= 650 + s_3 - \tfrac{1}{2}s_4 \\ x_2 &= 1100 - 2\,s_3 + \tfrac{1}{2}s_4 \\ s_1 &= 350 - s_3 + \tfrac{1}{2}s_4 \\ s_2 &= 450 + 2\,s_3 - \tfrac{1}{2}s_4 \\ p &= -17700 + 6\,s_3 + \tfrac{3}{2}s_4 \end{aligned} \tag{4.5}$$

So, currently with $s_3 = s_4 = 0$, $p = -17700$, is this the optimum?

YES! Increasing s_3 or s_4 will not drive p further down, so we are at the optimum.

Thus, $x^* = \langle 650, 1100, 350, 400, 0, 0 \rangle$, with $p^* = -17700$.

4.3 More Simplex by hand

Here are more examples of doing the Simplex algorithm by hand. These were taken out from the 2019 lecture notes, and the running example was put in instead. These example could still be of interest, though.

$$\begin{aligned} \min \quad & -x_1 - 2x_2 \\ \text{subject to} \quad & -2x_1 + x_2 + x_3 = 2 \\ & -x_1 + 2x_2 + x_4 = 7 \\ & x_1 + x_5 = 3 \\ & x_i \geq 0 \end{aligned} \tag{4.6}$$

Rewrite the constraints in terms of the slacks:

$$\begin{aligned} x_3 &= 2 + 2x_1 - x_2 \\ x_4 &= 7 + x_1 - 2x_2 \\ x_5 &= 3 - x_1 \end{aligned}$$

1. x_2 decreases the objective function the most, examine it:

$$\begin{aligned} &\text{Let } x_1 = 0 \text{ then} \\ &x_3 = 2 - x_2 \quad \geq 0 \rightarrow \quad x_2 \leq 2 \quad (= \frac{2}{1}) \\ &x_4 = 7 - 2x_2 \quad \geq 0 \rightarrow \quad x_2 \leq 3.5 \quad (= \frac{7}{2}) \\ &x_5 = 3 \end{aligned}$$

When $x_2 = 2$ then $\underline{x_3 = 0}$, $x_4 = 3$, $x_5 = 3$

$$\begin{aligned} &\text{replace } x_3 \text{ by } x_2 \\ &x_2 = 2 + 2x_1 - x_3 \end{aligned}$$

"Take out x_2, put in x_3"

Rewrite problem with $x_2 = 2 + 2x_1 - x_3$

$$\begin{aligned}
\min\ & -x_1 - 2(2 + 2x_1 - x_3) = -4 - 5x_1 + 2x_3 \\
\text{s.t. } & x_2 = 2 + 2x_1 - x_3 \\
& x_4 = 7 + x_1 - 2(2 + 2x_1 - x_3) = 3 - 3x_1 + 2x_3 \\
& x_5 = 3 - x_1
\end{aligned}$$

2. x_1 decreases objective function, examine

$$\begin{aligned}
&\text{Let } x_3 = 0, \text{ then} \\
&\quad x_2 = 2 + 2x_1 \\
&\quad x_4 = 3 - 3x_1 \geq 0 \rightarrow x_1 \leq 1 \\
&\quad x_5 = 3 - x_1 \geq 0 \rightarrow x_1 \leq 3
\end{aligned}$$

When $x_1 = 1$, then $x_2 = 4$, $\underline{x_4 = 0}$, $x_5 = 2$

$$\begin{aligned}
&\text{replace } x_4 \text{ by } x_1 \\
&x_1 = 1 + \frac{2}{3}x_3 - \frac{1}{3}x_4
\end{aligned}$$

"Take out x_4, put in x_1"

Rewrite problem with $x_1 = 1 + \frac{2}{3}x_3 - \frac{1}{3}x_4$

$$\begin{aligned}
\min\ & -4 - 5(1 + \frac{2}{3}x_3 - \frac{1}{3}x_4) + 2x_3 = -9 - \frac{4}{3}x_3 + \frac{5}{3}x_4 \\
\text{s.t. } & x_1 = 1 + \frac{2}{3}x_3 - \frac{1}{3}x_4 \\
& x_2 = 2 + 2(1 + \frac{2}{3}x_3 - \frac{1}{3}x_4) - x_3 = 4 + \frac{1}{3}x_3 - \frac{2}{3}x_4 \\
& x_5 = 3 - (1 + \frac{2}{3}x_3 - \frac{1}{3}x_4) = 2 - \frac{2}{3}x_3 + \frac{1}{3}x_4
\end{aligned}$$

3. x_3 decreases objective function, examine

$$\begin{aligned}
&\text{Let } x_4 = 0, \text{ then} \\
&\quad x_1 = 1 + \frac{2}{3}x_3 \\
&\quad x_2 = 4 + \frac{1}{3}x_3 \\
&\quad x_5 = 2 - \frac{2}{3}x_3 \geq 0 \rightarrow x_3 \leq 3
\end{aligned}$$

When $x_3 = 3$, then $x_1 = 3$, $x_2 = 5$, $\underline{x_5 = 0}$

$$\text{replace } x_5 \text{ by } x_3$$
$$x_3 = 3 + \frac{1}{2}x_4 - \frac{2}{3}x_5$$

"Take out x_3, put in x_5"

Rewrite problem with $x_3 = 3 + \frac{1}{2}x_4 - \frac{3}{2}x_5$

$$\begin{aligned} \min \ & -9 - \frac{4}{3}(3 + \frac{1}{2}x_4 - \frac{3}{2}x_5) + \frac{5}{3}x_4 = -13 + x_4 + 2x_5 \\ \text{s.t. } & x_1 = 3 - x_5 \\ & x_2 = 5 - \frac{1}{2}x_4 - \frac{1}{2}x_5 \\ & x_3 = 3 + \frac{1}{2}x_4 - \frac{3}{2}x_5 \end{aligned}$$

4. No way to further decrease objective function!

Minimum, $p^* = -13$ when $x_1 = 3$, $x_2 = 5$, $x_3 = 3$, $x_4 = 0$, $x_5 = 0$.

What does $x_3 = 3$ mean?

x_3 is the slack for

$$-2x_1 + x_2 + x_3 = 2$$

x_3 tells us how far $(-2x_1 + x_2)$ is away from 2

For a singleton requirement

$$x_1 \leq b_1$$

the slack variable s_1 (when optimum found)

$$x_1 + s_1 = b$$

will give the exact distance of x_1 from b.

Note: The algorithm started by going in the x_2 direction from $(0, 0)$. What if it instead had gone in the x_1 direction? Try it!

Begin by walking x_1-way instead

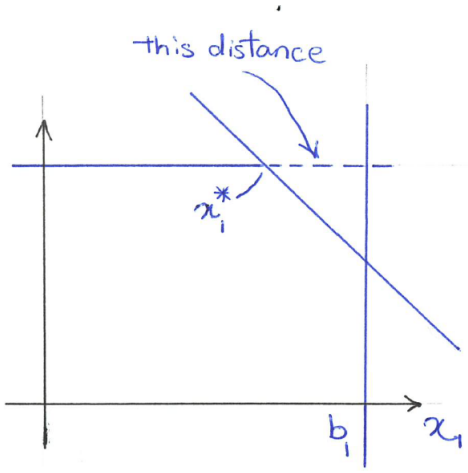


Figure 4.1: Graph 1

1. $\min\ -x_1 - 2x_2$
   s.t. $x_3 = 2 + 2x_1 - x_2$
   $x_4 = 7 + x_1 - 2x_2$
   $x_5 = 3 - x_1$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0 \rightarrow \begin{bmatrix} x_4 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 2 \\ 7 \\ 3 \end{bmatrix}$$

Optimal? No

Let $\underline{x_2} = 0$

$$x_3 = 2 + 2x_1$$
$$x_4 = 7 + x_1$$
$$x_5 = 3 - x_1 \geq 0 \rightarrow x_1 \leq 3$$

when $x_1 = 3$ then $x_3 = 8$, $x_4 = 10$, $\underline{x_5 = 0}$

replace x_5 by x_1
$x_1 = 3 - x_5$

why choose x_5 here? It is the only one that puts constraints on x_1

"Take out x_5, put in x_1"

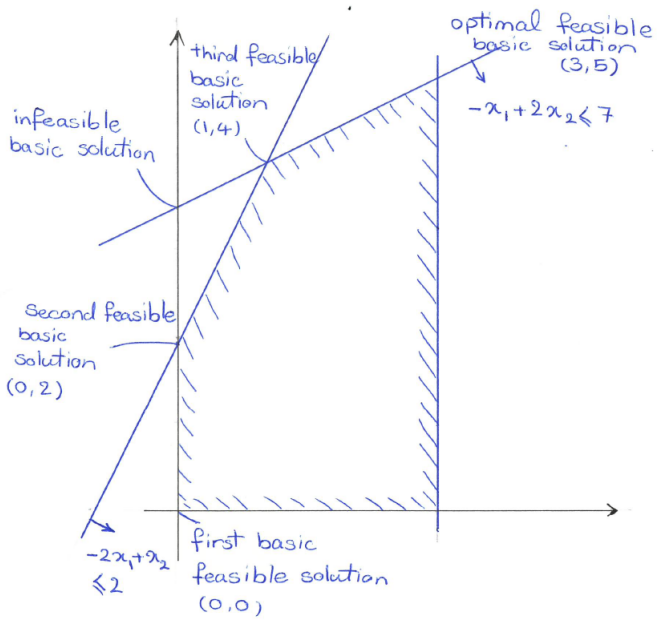


Figure 4.2: Graph 2

2.
$$\begin{aligned} \min \quad & -(3-x_5)-2x_2 \\ & = -3+x_5-2x_2 \\ \text{s.t.} \quad & x_3 = 2+2(3-x_5)-x_2 = 8-2x_5-x_2 \\ & x_4 = 7+(3-x_5)-2x_2 = 10-x_5-2x_2 \\ & x_1 = 3-x_5 \end{aligned}$$

$$\begin{bmatrix} x_2 \\ x_5 \end{bmatrix} = 0 \rightarrow \begin{bmatrix} x_1 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \\ 10 \end{bmatrix}$$

Optimal? No

Let $\underline{x_5} = 0$

$$\begin{aligned} & x_1 = 3 \\ & x_3 = 8-x_2 \geq 0 \rightarrow x_2 \leq 8 \\ & x_4 = 10-2x_2 \geq 0 \rightarrow x_2 \leq 5 \end{aligned}$$

when $x_2 = 5$ then $x_1 = 3$, $x_3 = 8$, $x_4 = 0$

replace x_4 by x_2

$$x_2 = 5 - \frac{1}{2}x_4 - \frac{1}{2}x_5$$

$$
\begin{aligned}
3. \qquad \min \; & -3 + x_5 - 2(5 - \frac{1}{2}x_4 - \frac{1}{2}x_5) \\
& = -13 + x_4 + 2x_5 \\
\text{s.t.} \quad & x_1 = 3 - x_5 \\
& x_2 = 5 - \frac{1}{2}x_4 - \frac{1}{2}x_5 \\
& x_3 = 3 - \frac{3}{2}x_5 + \frac{1}{2}x_4
\end{aligned}
$$

$$
\begin{bmatrix} x_4 \\ x_5 \end{bmatrix} = 0 \rightarrow \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ 3 \end{bmatrix}
$$

Optimal? YES!

No increase in x_4 or x_5 can decrease min!

$$
\begin{bmatrix} x_1^* \\ x_2^* \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}
$$

Optimal value -13

Another example

$$
\begin{aligned}
& -20x_1 - 10x_2 - 15x_3 \\
& 3x_1 + 2x_2 + 5x_3 + S_1 = 55 \\
& 2x_1 + x_2 + x_3 + S_2 = 26 \\
& x_1 + x_2 + 3x_3 + S_3 = 30 \\
& 5x_1 + 2x_2 + 4x_3 + S_4 = 57
\end{aligned}
$$

$$
\begin{aligned}
S_1 &= 55 - 3x_1 - 2x_2 - 5x_3 \\
S_2 &= 26 - 2x_1 - x_2 - x_3 \\
S_3 &= 30 - x_1 - x_2 - 3x_3 \\
S_4 &= 57 - 5x_1 - 2x_2 - 4x_3
\end{aligned}
$$

x_1 decreases objective, examine

Let $x_2 = 0, x_3 = 0$ then

$$S_1 = 55 - 3x_1 \geq 0 \rightarrow x_1 \leq \frac{55}{3} = 18.3$$
$$S_2 = 26 - 2x_1 \geq 0 \rightarrow x_1 \leq \frac{26}{2} = 13$$
$$S_3 = 30 - x_1 \geq 0 \rightarrow x_1 \leq \frac{1}{30}$$
$$S_4 = 57 - 5x_1 \geq 0 \rightarrow x_1 \leq \frac{57}{5} = 11.4 \qquad \text{most constraining}$$

$$x_1 = \frac{57}{5} \text{ then }, S_4 = 0, S_3 = 18.6, S_2 = 3.2, S_1 = 20.8$$

replace S_4 by x_1

$$\begin{aligned} x_1 &= \frac{1}{5}(57 - 2x_2 - 4x_3 - S_4) \\ &= 11.4 - 0.4x_2 - 0.8x_3 - 0.2S_4 \end{aligned}$$

Rewrite objective function in terms of x_2, x_3, S_4

$$\begin{aligned} &- 20(11.4 - 0.4x_2 - 0.8x_3 - 0.2S_4) - 10x_2 - 15x_3 \\ &= -228 - 2x_2 + x_3 + 4S_4 \end{aligned}$$

$$\begin{aligned} x_1 &= 11.4 - 0.4x_2 - 0.8x_3 - 0.2S_4 \\ S_1 &= 20.8 - 0.8x_2 - 2.6x_3 + 0.6S_4 \\ S_2 &= 3.2 - 0.2x_2 + 0.6x_3 + 0.4S_4 \\ S_3 &= 18.6 - 0.6x_2 - 2.2x_3 + 0.2S_4 \end{aligned}$$

Now only x_2 decreases the objective function, examine

Let $x_3 = 0, S_4 = 0$ then

$$x_1 = 11.4 - 0.4x_2 \geq 0 \rightarrow x_2 \leq \frac{11.4}{0.4} = 28.5$$
$$S_1 = 20.8 - 0.8x_2 \geq 0 \rightarrow x_2 \leq \frac{20.8}{0.8} = 26$$
$$S_2 = 3.2 - 0.2x_2 \geq 0 \rightarrow x_2 \leq \frac{3.2}{0.2} = 16 \quad \text{most constraining}$$
$$S_3 = 18.6 - 0.6x_2 \geq 0 \rightarrow x_2 \leq \frac{18.6}{0.6} = 31$$

replace S_2 by x_2

$$x_2 = 16 + 3x_3 - 5S_2 + 2S_4$$

rewrite objective function in terms of x_1, x_2, s_1, s_4, s_3

$$\begin{aligned} &-228 - 2(16 + 3x_3 - 5S_2 + 2S_4) + x_3 + 4S_4 = \\ &-260 - 5x_3 + 10S_2 \end{aligned}$$

$$\begin{aligned} x_1 &= 5 - 2x_3 + 2S_2 - S_4 \\ x_2 &= 16 + 3x_3 - 5S_2 + 2S_4 \\ S_1 &= 8 - 5x_3 + 4S_2 - S_4 \\ S_3 &= 9 - 4x_3 + 3S_2 - S_4 \end{aligned}$$

Now only x_3 decreases the objective function, examine

Let $S_2 = 0$, $S_4 = 0$ then

$$\begin{aligned} &x_1 = 5 - 2x_3 \geq 0 \Rightarrow x_3 \leq \frac{5}{2} = 2.5 \\ &x_2 = 16 + 3x_3 \\ &S_1 = 8 - 5x_3 \geq 0 \Rightarrow x_3 \leq \frac{8}{5} = 1.6 \quad \text{most constraining} \\ &S_3 = 9 - 4x_3 \geq 0 \Rightarrow x_3 \leq \frac{9}{4} = 2.25 \end{aligned}$$

replace S_1 by x_3

$$x_3 = 1.6 - 0.2S_1 + 0.8S_2 - 0.2S_4$$

Rewrite the objective function

$$\begin{aligned} &-260 - 5(1.6 - 0.2S_1 + 0.8S_2 - 0.2S_4) + 10S_2 = \\ &-268 + S_1 + 6S_2 + S_4 \qquad \text{No negative coefficient, DONE!} \end{aligned}$$

Optimal value: -268

$$\begin{aligned} \text{Found at } x_1^* &= 1.8 \\ x_2^* &= 20.8 \\ x_3^* &= 1.6 \end{aligned}$$

Slack values $S_1 = 0$, $S_2 = 0$, $S_3 = 2.6$, $S_4 = 0$

Lecture 5: Linear Programming V

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

5.1 Some more matrix algebra

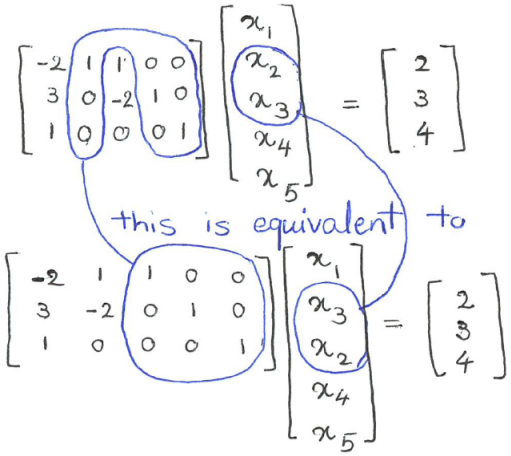


Figure 5.1: Moving columns around

It is OK to move columns around.

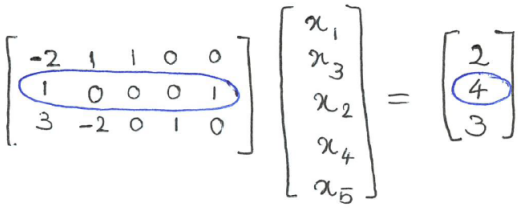


Figure 5.2: Moving rows around.

It is OK to move rows around.

In both cases we may multiply rows/columns with scalars and add other rows/columns.

5.2 Simplex in matrix form

Let us look again at the running example (3.1), this time in matrix form:

$$\begin{aligned} \min \quad & \begin{bmatrix} -12 & -9 & 0 & 0 & 0 & 0 \end{bmatrix} x \\ \text{s.t.} \quad & \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 4 & 2 & 0 & 0 & 0 & 1 \end{bmatrix} x = \begin{bmatrix} 1000 \\ 1500 \\ 1750 \\ 4800 \end{bmatrix} \\ & x \geq 0 \end{aligned} \tag{5.1}$$

A is not square, so we cannot simply invert it to get the solution $x = A^{-1}b$.

From linear algebra (see for example [Strang, 2009]):

If A has rank m, there will always exist at least one submatrix of A that has rank m.

If there exists in A an $m \times m$ submatrix B, such that B has full rank (linearily independent), then $x = B^{-1}b$ is a solution to $Ax = b$.

There can be *many* such B, in fact $\frac{(m+n)!}{m!n!}$, and each represents an extreme point. We want to explore these different Bs, called *basic solutions* to find the one that minimizes the objective function.

We need to do this cleverly:

- avoid exploring non-feasible basic solutions, and
- avoid exploring too many feasible basic solutions.

This is what the Simplex algorithm does.

Looking carefully a A of (5.1) we see that there *does* exist an $m \times m$ submatrix of rank m, namely the identity matrix I_4:

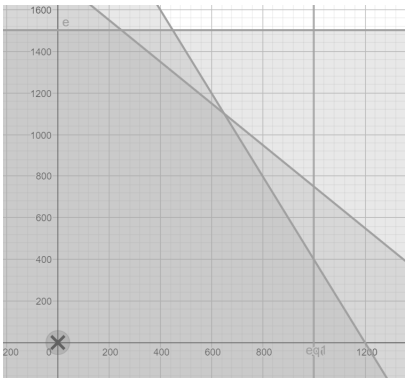
$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 4 & 2 \end{bmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

If we let $x = I_4^{-1}\, b$ then the slack variables become:

$$\begin{cases} s_1 &= 1000 \\ s_2 &= 1500 \\ s_3 &= 1750 \\ s_4 &= 4800 \end{cases} \qquad \begin{cases} x_1 &= 0 \\ x_2 &= 0 \end{cases}$$

These are currently the basic variables. These are currently the non-basic variables.

So, $(0, 0)$ is a solution.
Is it the optimal one?
How would we know?



Consider the objective function:

$$\begin{bmatrix} -12 & -9 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1000 \\ 1500 \\ 1750 \\ 4800 \end{bmatrix}$$

Currently, its value is 0.

If x_1 or x_2 were raised from 0, the objective function value would go down. And since we are minimizing, this is good!

So this is not the optimum. Look for a new B.

One of x_2 or x_2 should become a basic variable, and one of the s_i $(i = 1..4)$ should become non-basic.

Have to decide:

- which one of the slacks to "take out" (make non-basic)
- which one of x_1 or x_2 to "put in" (make basic)

and we have to be careful so we:

- avoid non-feasible basic solutions (extreme points)
- avoid exploring too many basic feasible solutions.

Rule of thumb:

- put in the one that decreases the objective function the most.

This decides our "pivot column".

Which one to take out?

Calculate ratios between the elements of b the elements of the pivot column. Smallest positive ratio gives the "pivot row".

Here, pivot column is given by x_1, then:

$$\frac{1000}{1} = 1000, \quad \cancel{\frac{1500}{0}}, \quad \frac{1750}{1} = 1750, \quad \frac{4800}{4} = 1200$$

Thus, s_1 decides our pivot row:

$$\left[\begin{array}{cccccc:c} 1 & 0 & 1 & 0 & 0 & 0 & 1000 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1500 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1750 \\ 4 & 2 & 0 & 0 & 0 & 1 & 4800 \\ \hdashline -12 & -9 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \tag{5.2}$$

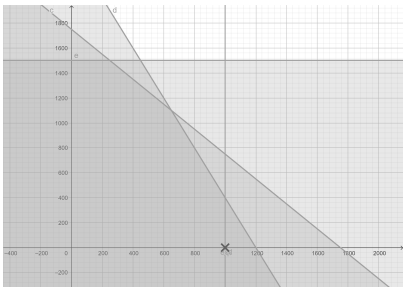
Now we use the pivot row to make the pivot column look like:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

This results in:

$$\begin{cases} x_1 &= 1000 \\ s_2 &= 1500 \\ s_3 &= 750 \\ s_4 &= 800 \end{cases} \qquad \begin{cases} x_2 &= 0 \\ s_1 &= 0 \end{cases}$$

Basic variables. Non-basic variables.



The objective function value is:

$$\begin{bmatrix} -12 & -9 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1000 \\ 0 \\ 0 \\ 1500 \\ 750 \\ 800 \end{bmatrix} = -12000$$

Is this the optimum? No, because increasing x_2 will further drive the objective function down.

So repeat, now for x_2 deciding the pivot column.

From here you can compare to the Simplex by hand that we did in Section 4.2, starting from (4.1). Comparing the numbers to see where which number turns up where, reveals how the whole procedure works. Please try it yourself.

The Simplex "recipe":

1. Determine if the objective function can be further decreased
   - Negative coefficient in front of some non-basic variable means that the objective function can be decreased.
2. Determine which non-basic variable to take out (make basic), and which basic variable to put in (make non-basic)

- Examine which non-basic variable that decrease the objective function the most, this is the one to take out.
- Examine which basic variable that constrains the non-basic variable the most, this is the one to put in.

3. Rewrite the problem in terms of the new basic variables
   - If there are negative coefficients in front of one non-basic variable, do it all again from the top.
   - Else, we are done and the values of the objective function and the basic variables are given when the non-basic variables are set to zero.

Silently we performed a little trick at (5.2). We assembled the matrices into a single matrix, like so:

$$\left[\begin{array}{c|c} A & b \\ \hline c^T & 0 \end{array}\right]$$

This is called the *tableaux*, and it allows a convenient way to perform the manipulations described in Section 4.2 in matrix form.

So, here we go again... The tableux:

$$\left[\begin{array}{cccccc|c} 1 & 0 & 1 & 0 & 0 & 0 & 1000 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1500 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1750 \\ 4 & 2 & 0 & 0 & 0 & 1 & 4800 \\ \hline -12 & -9 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \tag{5.3}$$

We observe the I_4 identity matrix, and ask: are we at the optimum?

No! Because of the negative coefficients in the bottom row we know that we are not.

Select the pivot column. -12 is the largest negative coefficient, so it seems like the best choice (but we do not know, we only guess!). So the first column is the pivot column.

Now we find the pivot row by determining the for x_1 most constraining constraint, which is the constraint related to s_1. This is our pivot row, see (5.2).

Now, doing row manipulations using the pivot row (row 1), we turn the pivot column into $\begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}^T$.

- The pivot element is already 1 so no need to do anything with the pivot row.
- The pivot column element for row 2 is already zero, so no need to do anything there.
- Subtract the pivot row from row 3.
- Multiply the pivot row by -4 and add to row 4.
- Multiply the pivot row by 12 and add to the bottom row.

This results in the following new tableaux:

$$\left[\begin{array}{cccccc|c} 1 & 0 & 1 & 0 & 0 & 0 & 1000 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1500 \\ 0 & 1 & -1 & 0 & 1 & 0 & 750 \\ 0 & 2 & -4 & 0 & 0 & 1 & 800 \\ \hline 0 & -9 & 12 & 0 & 0 & 0 & 12000 \end{array}\right] \tag{5.4}$$

Comparing with (4.3) we recognize the numbers.

The I_4 submatrix given by columns 1 and $4-6$ mean that x_1, s_2, s_3, and s_4 are our basic variables, and that x_2 and s_1 are non-basic. And the current value of the objective function is -12000 as we see in the lower right corner.

Now we ask: are we at the optimum?

The answer is no, since increasing the value of either x_2 (which is currently 0) we can decrease the objective function further. So, column 2 becomes our pivot column.

Then we select the pivot row by looking at which constraint constrains x_2 the most. This is the constraint representing s_4, see also (4.3), and so row 4 is our pivot row.

$$\left[\begin{array}{cc|cccc:c} 1 & 0 & 1 & 0 & 0 & 0 & 1000 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1500 \\ 0 & 1 & -1 & 0 & 1 & 0 & 750 \\ \hline 0 & 2 & -4 & 0 & 0 & 1 & 800 \\ \hdashline 0 & -9 & 12 & 0 & 0 & 0 & 12000 \end{array}\right]$$

Row manipulations to make the pivot column $\begin{bmatrix}0 & 0 & 0 & 1 & 0\end{bmatrix}^T$:

- Replace the pivot row by itself multiplied by $\frac{1}{2}$.
- Then, subtract the pivot row from rows 2 and 3.
- Add 9 times the pivot row to the bottom row.

This results in the following new tableaux:

$$\left[\begin{array}{cccccc:c} 1 & 0 & 1 & 0 & 0 & 0 & 1000 \\ 0 & 0 & 2 & 1 & 0 & -\frac{1}{2} & 1100 \\ 0 & 0 & 1 & 0 & 1 & -\frac{1}{2} & 350 \\ 0 & 1 & -2 & 0 & 0 & \frac{1}{2} & 400 \\ \hdashline 0 & 0 & -6 & 0 & 0 & \frac{9}{2} & 15600 \end{array}\right]$$

Are we done? No, increasing s_1 will further decrease the objective function.

Column 3 is our pivot column. The pivot row is row 3 (why?). Doing the same kind of manipulation as earlier, we arrive at:

$$\left[\begin{array}{ccccccc} 1 & 0 & 0 & 0 & -1 & \frac{1}{2} & 650 \\ 0 & 0 & 0 & 1 & -2 & \frac{1}{2} & 400 \\ 0 & 0 & 1 & 0 & 1 & -\frac{1}{2} & 350 \\ 0 & 1 & 0 & 0 & 2 & -\frac{1}{2} & 1100 \\ \hdashline 0 & 0 & 0 & 0 & 6 & \frac{3}{2} & 17700 \end{array}\right] \tag{5.5}$$

Are we done? YES!

As earlier, $x^* = \langle 650, 1100, 350, 400, 0, 0\rangle$, with $p^* = -17700$.

Lecture 6: Linear Programming VI

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

6.1 LP/Simplex revisited

We have been considering the problem

$$\begin{aligned} &\min c^T x \\ &\text{s.t. } Ax \leq b \\ &\quad x \geq 0 \end{aligned}$$

This is a special case of all possible LP problems.

This we can rewrite into the "standard form".

$$\begin{aligned} &\min c^T x \\ &\text{s.t. } \begin{bmatrix} A & I \end{bmatrix} \begin{bmatrix} x \\ s \end{bmatrix} = b \\ &\quad x \geq 0, s \geq 0 \end{aligned}$$

Not so very "standard" after all, but anyway...

Then we asked the question

Is $x = 0$, $s = b$, $\min c^T x = 0$ the optimal solution?

If yes, we're done!

If no, we rewrite the problem to

$$\begin{aligned} &\min\ c'^T x' \\ &\text{s.t. } A'x' = b' \\ &\quad x' \geq 0 \end{aligned}$$

And examine this new formulation.

How did we know we are done?

We looked at ${c'}^T$ and its coefficient ${c'}_i$

If any ${c'}_i < 0$ then increasing its corresponding ${x'}_i$ will drive min ${c'}^T x'$ further down to its min.

A relevant question now is:

What if the original c^T has only non-negative c_i, are we then immediately done with the optimum at $x = 0$, with $\min c^T x = 0$?

The answer is: Yes ... and no!

If $x \geq 0$ (which we require) and $c^T \geq 0$, then a lower bound on min $c^T x$ is of ocurse zero.

We can *never* get lower, no way.

Thus, if zero is not the solution, then the original problem must include a constraint:

$$a_{ij}x_j + a_{ik}x_k + ... \geq b_i$$

Example:

$$\begin{aligned} \min\ & x + y \\ \text{s.t. } & x + 2y \geq 2 \\ & x, y \geq 0 \end{aligned}$$

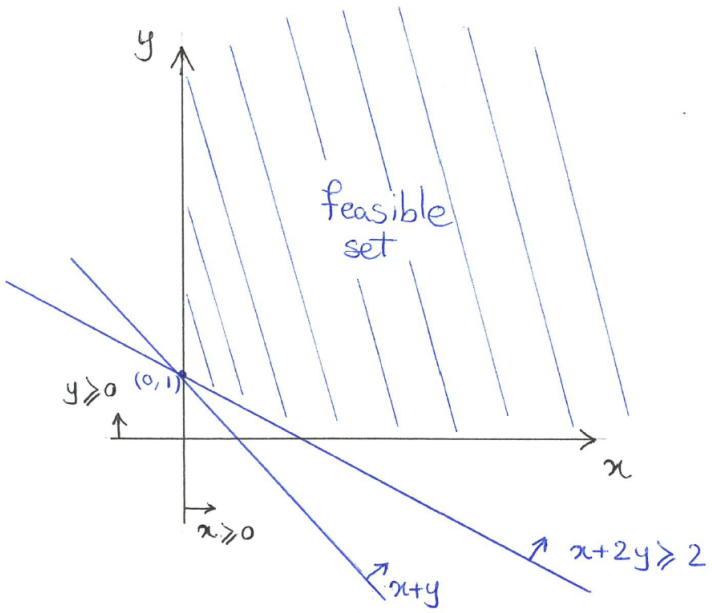


Figure 6.1: Graph 1

Then origin is not in the feasible set! (assuming $b_i \geq 0$, which we always can assume)

The Simplex algorithm as has been described (implicitly) assumes that the origin is in the feasible set.

If it is not, we need to find a point in the feasible set to start from, prior to get the Simplex algorithm going.

This is called "Phase I".

An example:

$$\begin{aligned} \min\ & p = x + y + z \\ \text{s.t. } & x + y + z \leq 40 \\ & 2x + y - z \geq 10 \\ & -y + z \geq 10 \\ & x, y, z \geq 0 \end{aligned}$$

Add slacks and surplus variables:

$$\begin{aligned} x + y + z + s &= 40 \\ 2x + y - z - t &= 10 \\ -y + z - u &= 10 \\ x, y, z, t, u, s &\geq 0 \end{aligned}$$

Rewrite the constraints:

$$\begin{aligned} s &= \ \ 40 - x - y - z \\ t &= -10 + 2x + y - z \\ u &= -10 - y + z \end{aligned}$$

Note: at $(0, 0, 0)$, t and u are negative! Not good! It means $(0, 0, 0)$ not in the feasible set.

Phase I: Get into feasible set.

For t, x *raises* it the most, rewrite

$$x = 5 - \frac{1}{2}y + \frac{1}{2}z + \frac{1}{2}t$$

Rewrite the system in terms of x, s, and u:

$$\begin{aligned} \min \quad p &= (5 - \tfrac{1}{2}y + \tfrac{1}{2}z + \tfrac{1}{2}t) + y + z && = 5 + \tfrac{1}{2}y + \tfrac{3}{2}z + \tfrac{1}{2}t \\ x &= 5 - \tfrac{1}{2}y + \tfrac{1}{2}z + \tfrac{1}{2}t \\ s &= 40 - (5 - \tfrac{1}{2}y + \tfrac{1}{2}z + \tfrac{1}{2}t) - y - z && = 35 - \tfrac{1}{2}y - \tfrac{3}{2}z - \tfrac{1}{2}t \\ u &= -10 - y + z \end{aligned}$$

Still $u < 0$ at $y = z = 0$

Rewrite for u:

$$z = 10 + y + u$$

Rewrite the system in terms of x, s, z:

$$\begin{array}{lrll} \min & p & = 5 + \frac{1}{2}y + \frac{3}{2}(10 + y + u) + \frac{1}{2}t & = 20 + 2y + \frac{1}{2}t + \frac{3}{2}u \\ & x & = 5 - \frac{1}{2}y + \frac{1}{2}(10 + y + u) + \frac{1}{2}t & = 10 + \frac{1}{2}t + \frac{1}{2}u \\ & s & = 35 - \frac{1}{2}y - \frac{3}{2}(10 + y + u) - \frac{1}{2}t & = 20 - 2y - \frac{1}{2}t - \frac{3}{2}u \\ & z & = 10 + y + u & \end{array}$$

Now, at $y = t = u = 0$, x, s, $z \geq 0$

OK!

Phase II: Simplex

Are we done? YES! No negative coefficients in p and all variables ≥ 0

Solution $p^* = 20$, occurs at $x^* = 10$, $y^* = 0$, $z^* = 10$, ($s^* = 20$, $t^* = 0$, $u^* = 0$).

Note: As you recognized, getting into the feasible set as was done in "Phase I" above can be done in tableux form. Please try it.

Let us look at what happens with an ill posed example

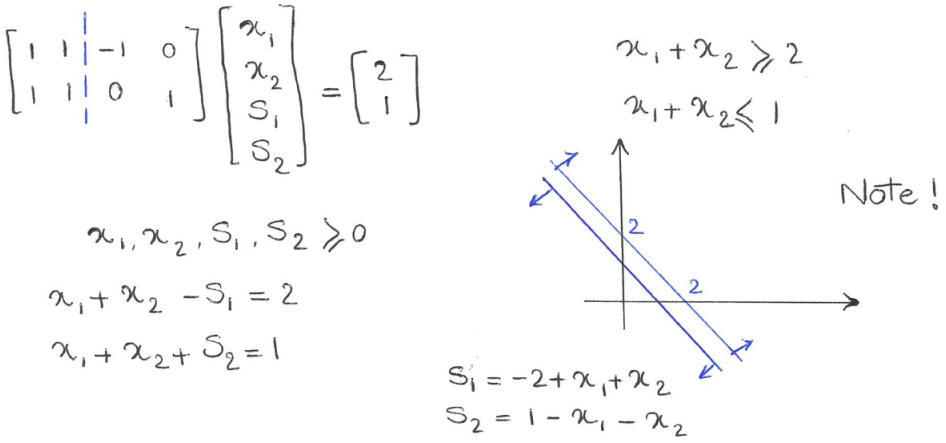


Figure 6.2: Ill posed example, conflicting constraints.

At $(x_1, x_2) = (0, 0)$, $s_1 \leq 0$, so we need to rewrite s_1 in terms of x_1 (or x_2):

$$x_1 = 2 - x_2 + s_1.$$

Then for s_2 we get:

$$s_2 = 1 - x_1 - x_2 = 1 - (2 - x_2 + s_1) - x_2 = -1 - s_1.$$

Now at $(x_2, s_1) = (0, 0)$ s_2 is negative, rewrite:

$$\begin{aligned} s_1 &= -1 - s_2 \\ x_1 &= 2 - x_2 - 1 - s_2 = 1 - x_2 - s_2. \end{aligned}$$

But now we see that at $(x_2, s_2) = (0, 0)$, $s_1 < 0$.

Thus, either s_1 or $s_2 < 0$, so no feasible solution exists

Solving it with

$$B = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

will give the solution

$$B^{-1}b = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

That is, that $s_1 = -2$, $s_2 = 1$ and $x = y = 0$. This is of course infeasible due to $s_1 = -2$.

6.2 Why does the optimum end up negated in the tableux?

We have

$$\begin{aligned} &\min\ c^T x \\ &\text{s.t. } Ax = b \\ &(\text{with } x \geq 0) \end{aligned}$$

We can rewrite this as:

$$\begin{aligned} &\min\ p \\ &\text{s.t. } Ax = b \\ &\quad c^T x = p \end{aligned}$$

Which can again be written as:

$$\begin{aligned} &\min\ p \\ &\text{s.t. } Ax - b = 0 \\ &\quad c^T x - p = 0 \end{aligned}$$

Matrix formulation:

$$\left[\begin{array}{c|c} A & b \\ \hline c^T & p \end{array}\right] \left[\begin{array}{c} x \\ \hline -1 \end{array}\right] = 0$$

The -1 is why p ends up negated.

6.3 Why only positive pivot?

The tableaux method requires you to seek out pivot row elements. You do this by looking at the ratio between the elements of b and the pivot column. But you only look at *positive* coefficients.

The reason is that a negative coefficient in the pivot column means an *increase* in the pivot row variables as the corresponding pivot column variables increases.

Compare the example (4.6) of Section 4.3:

$$\begin{aligned} \min \quad & -x_1 - 2x_2 \\ \text{s.t.} \quad & -2x_1 + x_2 + x_3 = 2 \\ & -x_1 + 2x_2 + x_4 = 7 \\ & x_1 + x_5 = 3 \end{aligned}$$

In the second step we have the equation system (for $x_3 = 0$, x_1 entering):

$$\begin{aligned} x_2 &= 2 + 2x_1 \\ x_4 &= 3 - 3x_1 \geq 0 \quad \Rightarrow x_1 \leq 1 \\ x_5 &= 3 - x_1 \geq 0 \quad \Rightarrow x_1 \leq 3 \end{aligned}$$

The result of line 2 is: x_4 exits, Which corresponds to the tableaux:

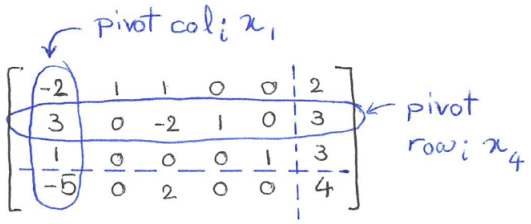


Figure 6.3: Shape 3

For the top row (pivot variable x_2) the -2 corresponds to the $+2x_1$ in the eq system.

Put differently, the top row represents:

$$-2x_1 + x_2 + x_3 = 2$$

When $x_3 = 0$ we have $x_2 = 2 + 2x_1$ (see above)

6.4 What if no positive pivots exist?

Assume we have a pivot column variable, x_i, that is, there is a column i for which $c_i^T < 0$.

If we rewrite the tableaux with the basics on the left and consider all other non-basic to be 0, then we have an equation system.

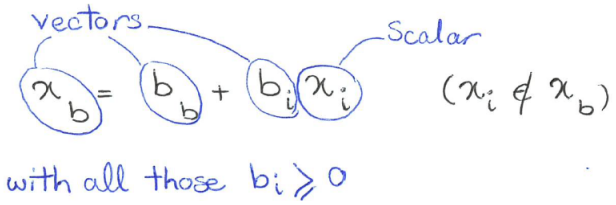


Figure 6.4: Shape 4

This means we could decrease the objective function by increasing x_i, but this could increase the basics and hence increase the objective function.

Either we have to look for another pivot column variable, or the problem is ill posed:

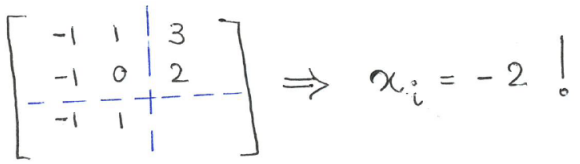


Figure 6.5: Ill posed problem.

Lecture 7: Integer Programming I

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

LP assumes all variables are continuous. If however the variables are required to be integer problem then we have an *integer programming* problem, IP.

Much harder to solve.

7.1 Exponential growth

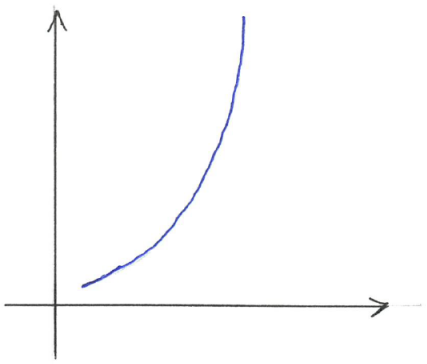


Figure 7.1: Exponential growth.

Assume computer program determines one solution per millisecond for a program with Boolean variables:

10 variables:	$2^{10} = 1024$ ms	$\approx$ 1 second
20 variables:	$2^{20} = 1048576$ ms	$\approx$ 17 minutes
30 variables:	$2^{30} = \ldots$	$\approx$ 12 days
40 variables:	$2^{40} = \ldots$	$\approx$ 35 years

Example, batch production

$$\begin{aligned} \max \quad & 20x_1 + 6x_2 + 8x_3 \\ & 0.8x_1 + 0.2x_2 + 0.3x_3 \leq 4 \\ & 0.4x_1 + 0.3x_2 \qquad\quad \leq 2 \\ & 0.2x_1 \qquad\quad + 0.1x_3 \leq 1 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

The x_i are batches of different products

LP solution

$$\left.\begin{array}{l} x_1^* = 0 \\ x_2^* = 6.667 \\ x_3^* = 8.889 \end{array}\right\} \max = 111.111\ldots$$

However, batches must really be whole integers, not only positive numbers.

$x_1 = 0$ for all cases

Rounding up ($x_2 = 7$, $x_3 = 9$, max $= 114$) or

rounding down ($x_2 = 6$, $x_3 = 8$, max $= 100$) does *not* give the optimal value ($x_2 = 5$, $x_3 = 10$, max $= 110$)

For IP problems, the variable's domain constraints restrict them to be non-negative integers.

Note that the optimum for the IP, 110 is smaller than the optimum for the LP, $11.111\ldots$

7.2 LP relaxation of IP problems

The discrete domain constraint means that the feasible set is no longer dense, only certain "points" are valid as solutions. Relaxation disregards the discrete domain constraint, and treat discrete variables as real valued.

Thus, relaxation grows the feasible set.

The feasible set of the relaxed problem contains the feasible set of the original problem. All solutions of the original problem (with discrete variables) are solutions of the relaxed problem (with continuous variables)

Note: It's not as simple as rounding up or down to the next discrete value

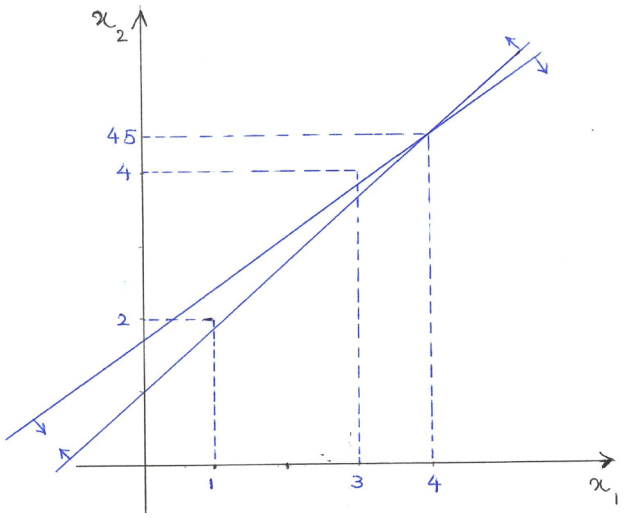


Figure 7.2: The integer solution is $(1, 2)$, while the relaxed problem has the solution $(4, 4.5)$

Relaxation alters the problem so that all feasible solutions to the original problem, are also solutions to the altered "relaxed" problem

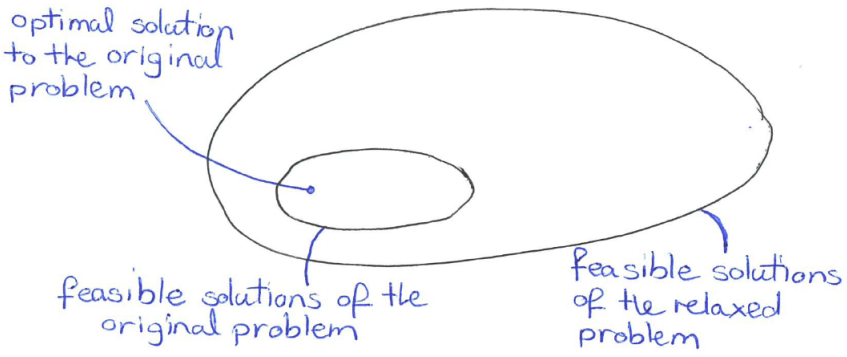


Figure 7.3: LP relaxation grows the feasible set.

What good is relaxation?

- show infeasibility
- get bounds on the true optimal solutions (lower bounds for min, upper bound for max)
- rounded solutions may be good enough
- if the relaxed optimum is also feasible for the original problem, then it is the optimum for the original solution

Thus, if all discrete variables happen to take discrete values in the relaxed solution, then the real optimum has been found

Compare LP, IP

$$z_1^* = \max c^T x \quad \text{s.t. } Ax \le b, \; 0 \le x \qquad z_3^* = \max c^T x \quad \text{s.t. } Ax \le b, \; 0 \le x, \text{ integer}$$

$$z_2^* = \max c^T x \quad \text{s.t. } Ax \le b, \; 0 \le x \le 1 \qquad z_4^* = \max c^T x \quad \text{s.t. } Ax \le b, \; 0 \le x \le 1, \text{ integer}$$

What can we say about the relationships between the optimal solutions, z_1^*, z_2^*, z_3^*, z_4^*?

Which one has the (potentially) largest solution? z_1^*

Which one has the (potentially) smallest solution? z_4^*

What about z_2^* and z_3^*? We cannot say.

Thus:

$$z_4^* \le \begin{Bmatrix} z_2^* \\ z_3^* \end{Bmatrix} \le z_1^*$$

7.3 When is the relaxed solution the true optimum?

Regard again our running example (5.1). We solved this as an ordinary LP problem, and we got an integer solution! What luck!

The reason is that the A and b matrices have certain properties that make the basic solutions that we encountere integer. It is known that given an IP problem A, b, c with b integer, if the optimal basic solution B^* has $\det(B^*) = \pm 1$, then the optimal solution is integer. So, when does the optimal basic solution B^* have $\det(B^*) = \pm 1$? One special case for which when we are guaranteed this, is when *all* basic solutions have $\det(\cdot) = \pm 1$. And we can strengthen this to arrive at an important special case: A is *totally unimodular*.

Unimodular matrix

A *unimodular* matrix is:

- Square
- Integer
- Determinant ± 1

Note: Square and non-zero determinant means that the matrix is invertible

Note: Integer and determinant ± 1 means that the inverse matrix is also integer

Totally unimodular

- not necessarily square, but
- every square, non-singular sub-matrix is unimodular

A 1×1 sub-matrix is square, thus every element of a totally unimodular matrix is –1, 0, or 1.

Being unimodular, every non-singular sub-matrix has integer inverse.

Unimodular matrix, example:

$$A = \begin{bmatrix} 2 & 3 & 2 \\ 4 & 2 & 3 \\ 9 & 6 & 7 \end{bmatrix}, \quad \det(A) = 1, \quad A^{-1} = \begin{bmatrix} -4 & -9 & 5 \\ -1 & -4 & 2 \\ 6 & 15 & -8 \end{bmatrix}$$

Not necessarily -1, 0, 1 elements.

Necessarily square

Totally unimodular matrix, example:

$$\begin{bmatrix} -1 & -1 & 0 & 0 & 0 & 1 \\ 1 & 0 & -1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 1 & -1 \end{bmatrix}$$

Not necessarily square

Necessarily only -1, 0, 1 elements

Known fact

For the IP problem

$$\begin{aligned} \min \quad & c^T x \\ & Ax = b \\ & x \in \mathbb{N} \\ & b \in \mathbb{Z} \end{aligned}$$

If A is totally unimodular, then the optimal solution is integer, since then for every basic solution B, $x = B^{-1}\, b$ is integer (whenever b is integer).

When is A totally unimodular?

A sufficient (but *not necessary*) condition ([Williams, 2013], p.213):

1. Each element of A either $-1, 0$, or 1
2. At most two non-zero elements in each column (row)
3. The columns (rows) can be partitioned into M_1, M_2, such that:
   (a) for two non-zero elements of *equal* sign in same column (row), one is in M_1, the other in M_2
   (b) for two non-zero elements of *different* signs in same column (row), both are in M_1, or both in M_2

Since if A is totally unimodular, then so is also A^T, and the other way around, we can exchange "column" for "row", and vice versa.

Condition 3 above means that for each column c, it holds that

$$\sum_{r \in M_1} a_{r,c} - \sum_{r \in M_2} a_{r,c} = 0,$$

where r counts the rows.

Either we have elements of different signs in either M_1 or M_2, and then they cancel "within" M_1 (or M_2). Or, we have elements of the same sign, one in M_1 the other in M_2 and then they cancel "between" M_1 and M_2.

Special case:

M_1 empty, M_2 contains all columns (rows), then all columns (rows) have either:

- at most one non-zero element ± 1, or
- two non-zero elements +1 and -1

Example:

$$\begin{aligned} x_1 &= 42 \\ x_2 &= 11 \\ -x_1 - x_2 &= -66 \end{aligned} \qquad A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{bmatrix} = M_2 \qquad M_1 = \begin{bmatrix} \ \end{bmatrix}$$

But note that this is equivalent to:

$$\begin{aligned} x_1 &= 42 \\ x_2 &= 11 \\ x_1 + x_2 &= 66 \end{aligned} \qquad A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \hline 1 & 1 \end{bmatrix} = \begin{bmatrix} M_1 \\ M_2 \end{bmatrix}$$

Another example:

In one of the assignments almost this A turns up

$$A = \begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix}$$

Is A totally unimodular?

Each element of A is -1, 0, or 1.

Each row has two non-zero elements +1 and -1. Thus, no more than two non-zero elements in each row.

We can partition A into M_1 empty, and $M_2 = A$

Thus, all the sufficient conditions are satisfied, and A is totally unimodular.

Counter example:

It is known that a matrix that has inside it the structure:

$$\begin{bmatrix} & \vdots & & \vdots & \\ \cdots & +1 & \cdots & +1 & \cdots \\ & \vdots & & \vdots & \\ \cdots & +1 & \cdots & -1 & \cdots \\ & \vdots & & \vdots & \end{bmatrix}$$

cannot be totally unimodular. (Why?)

The transportation problem

See Chapter 5.3 in [Williams, 2013].

Introduce variables x_{ij} to represent quantity sent from S_i to T_j in a year:

$$\begin{array}{lrcrcrc} \min & 132\,x_{11} & + & 97\,x_{13} & + & 103\,x_{14} & + \\ & 85\,x_{21} & + & 91\,x_{22} & + & 106\,x_{31} & + \\ & 89\,x_{32} & + & 100\,x_{33} & + & 98\,x_{34} & \end{array}$$

$$\begin{array}{lrcl} \text{s.t.} & x_{11}+x_{12}+x_{13}+x_{14} & \leq & 135 \\ & x_{21}+x_{22}+x_{23}+x_{24} & \leq & 56 \\ & x_{31}+x_{32}+x_{33}+x_{34} & \leq & 93 \\ & x_{11}+x_{21}+x_{31} & = & 62 \\ & x_{12}+x_{22}+x_{32} & = & 83 \\ & x_{13}+x_{23}+x_{33} & = & 39 \\ & x_{14}+x_{24}+x_{34} & = & 91 \end{array}$$

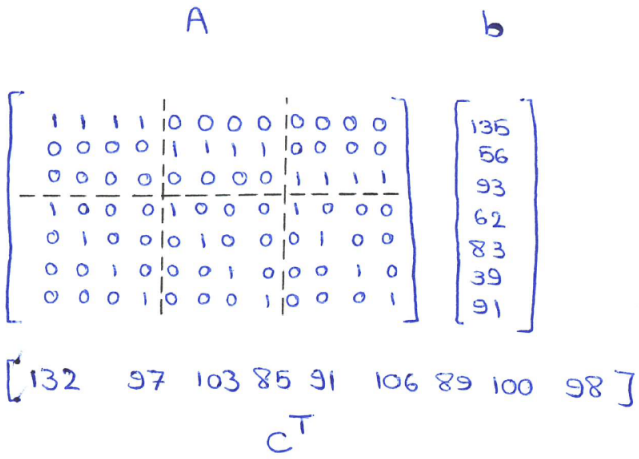


Figure 7.4: The transportation problem matrices.

We can only ship whole numbers of items. Integer solution looked for. So this is an IP problem.

A can be divided into two sub-matrices so that for each column one non-zero is in one sub-matrix and the other (there are only two in each column) non-zero element is in the other sub-matrix.

This is Good!

7.4 IP summarized

$$\left.\begin{array}{rl} \min & c^T x \\ \text{s.t.} & Ax = b \\ & x \geq 0 \end{array}\right\} \text{ordinary LP}$$

$$\left.\begin{array}{l} x \in \mathbb{N}^+ \end{array}\right\} \text{makes it IP}$$

The equation $Ax = b$, with A non-square, has many solutions (typically). Such solutions are given by invertible $m \times m$ sub-matrices of A.

The optimization problem now involves finding one solution that minimizes the objective function $c^T x$.

If the relaxed (LP) problem happened to give a solution where also $x \in \mathbb{N}^+$, then this is a solution to the IP problem.

If A is totally unimodular, then every invertible sub-matrix is uni-modular, its inverse exists and is integer.

Thus, if A totally unimodular, and b is integer, then all the solutions will be integer.

EEN025: Constraint Programming and Applied Optimization **Fall 2020**

Lecture 8: Mixed Integer Linear Programming I

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

General framework for solving optimum problems involving both discrete and continuous variables (with linear constraints)

$$\begin{aligned} \min \quad & c^T \begin{bmatrix} x \\ z \end{bmatrix} \\ \text{s.t.} \quad & A \begin{bmatrix} x \\ z \end{bmatrix} = b \\ & x \geq 0, z \geq 0 \\ & x \in \mathbb{R}^m, z \in \mathbb{Z}^n \end{aligned}$$

$$x \in \mathbb{R}^m : \text{ real}$$

$$z \in \mathbb{Z}^n : \text{discrete}$$

MILP example

(Kobetski, Fig 4.5)

$$\begin{aligned} \min \quad & -1000x_1 - 700x_2 - 200x_3 \\ \text{s.t.} \quad & 100x_1 + 50x_2 \leq 2425 \\ & 20x_2 \leq 510 \\ & x_3 \leq 0.5 \\ & x_1, x_2, x_3 \geq 0 \\ & x_1, x_2 \in \mathbb{Z}, x_3 \in \mathbb{R} \end{aligned}$$

LP-relaxation gives solution:

$x = \{11.5, 25.5, 0.5\}$ x_1, x_2 not integer
$p = -29\,450$

Solve two new problems with the added respective constraints:

$$x_2 \leq 25, \text{ and } x_2 \geq 26$$

The second part of the solution above ($x_2 \geq 26$) violates constraint $20\,x_2 \leq 510$

LP relaxation with the $x_2 \leq 25$ constraint added gives:

$x = \{11.75, 25, 0.5\}$ $x_1 = 11.75$ means this is not a valid solution
$p = -29350$ Note, worse (= larger) than previous p.

Solve two new problems, with constraints: $x_2 \leq 25$, and

$x_1 \leq 11$	$x_1 \geq 12$
$x = \{11, 25, 0.5\}$	$x = \{12, 24.5, 0.5\}$
$p = -28\,600$	$p = -29\,250$
A valid solution!	Not valid
Is it optimal?	

Solve two new problems, with constraints: $x_1 \geq 12$, and

$x_2 \leq 24$	$x_2 \geq 25$
$x = \{12.25, 24, 0.5\}$	violates constraint
$p = -29\,150$	$100x_1 + 50x_2 \leq 2425$
	since
	$100 \times 12 + 50x_2 = 2425$

Solve two new problems, with constraints: $x_2 \leq 24$, and

$x_1 \leq 12$	$x_1 \geq 13$
$x = \{12, 24, 0.5\}$	$x = \{13, 22.5, 0.5\}$
$p = -28\,900$	$p_{LP} = -28\,850$
OK , optimal?	Not IP. Continue?
Better (= lower) than $-28\,600$	No: $-28\,850 \geq -29\,150$

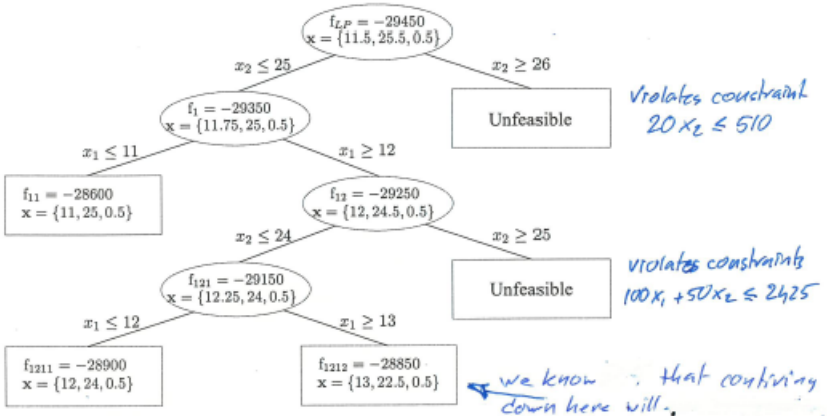


Figure 4.5: The relaxation tree for the MILP example.

the original constraints, for example x_2. This is done by making two copies of the relaxed constraint set and adding the constraints $x_2 \leq \lfloor 25.5 \rfloor = 25$ and $x_2 \geq \lceil 25.5 \rceil = 26$ to the first and the second copy respectively. Now, the second condition on x_2 clearly violates the constraint $20x_2 \leq 510$, which leaves only the first copied set to be examined more closely.

Proceeding in this way, the tree of different problem relaxations is constructed, see Figure 4.5. The nodes corresponding to f_{11} and f_{1211} close the search along their branches (such search-closing nodes are denoted by rectangles in the figure), since they represent feasible optimal solutions to the original problem. Also, the upper bound on the optimal solution is updated in these nodes to $f_U = -28600$ and $f_U = -28900$ respectively. On the other side of the scale are the unfeasible nodes, also closing the search. In between, there are the nodes that are feasible only within the relaxed context. The search along such nodes is pursued until they are known to be unpromising, i.e. until $f^*_{such\_node} \geq f_U$, which is exactly what happens at f_{1212}, concluding the search algorithm. The answer to the original problem is thus stored in the node f_{1211}, containing the minimal feasible solution to the original constraints.

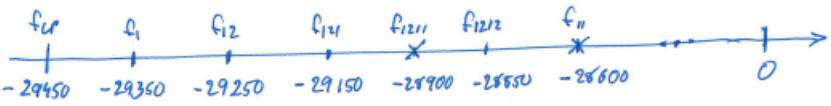


Figure 8.1: Simplex in Branch and Bound to solve MILP problem.

Another MILP example (actually IP)

$$\begin{aligned} \min\ & -x_1 - x_2 \\ \text{s.t.}\ & -2x_1 + 2x_2 \geq 1 \\ & -8x_1 + 10x_2 \leq 13 \\ & x_1, x_2 \geq 0 \quad x_1, x_2 \in \mathbb{Z} \end{aligned}$$

LP relaxation gives $x = \{4, 4.5\}$, $p = -8.5$

Solve two new problems with added constraints

$x_2 \leq 4$	$x_2 \geq 5$
$x = \{3.5, 4\}$	unfeasible
$p = -7.5$	

Solve two new problems with added constraints: $x_2 \leq 4$, and

$x_1 \leq 3$	$x_1 \geq 4$
$x = \{3, 3.7\}$	unfeasible
$p = -6.7$	

Solve two new problems with added constraints: $x_1 \leq 3$, and

$x_2 \leq 3$	$x_2 \geq 4$
$x = \{2.5, 3\}$	unfeasible
$p = -5.5$	

Solve two new problems with added constraints: $x_2 \leq 3$, and

$x_1 \leq 2$	$x_1 \geq 3$
$x = \{2, 2.9\}$	unfeasible
$p = -4.9$	

Solve two new problems with added constraints: $x_1 \leq 2$, and

$x_2 \leq 2$	$x_2 \geq 3$
$x = \{1.5, 2\}$	unfeasible
$p = -3.5$	

Two new problems with added constraints: $x_2 \leq 2$, and

$x_1 \leq 1$	$x_1 \geq 2$
$x = \{1, 2\}$	$x = \{1.5, 2\}$
$p = -3$	$p = -3.5$
optimal ?	Continue ?
	No, same as previous!

IP optimal solution

$$x^* = \{1, 2\}$$
$$p = -3$$

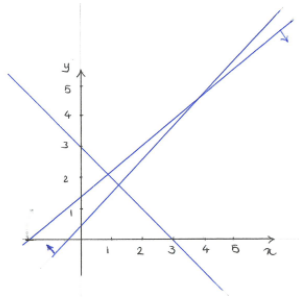


Figure 8.2: Optimal IP solution.

$$\begin{aligned} \min \; & -x - y \\ \text{s.t.} \; & -2x + 2y \geq 1 \\ & -8x + 10y \leq 13 \\ & x, y \geq 0 \\ & x, y \in \mathbb{Z} \end{aligned}$$

Big-M Example

Lets look at a variant of a familiar example, where either constraint 3 or constraint 4, but not both, should hold.

$$\begin{aligned} & \max 12x_1 + 9x_2 \\ & x_1 \leq 1000 \\ & x_2 \leq 1500 \\ 3) \quad & x_1 + x_2 \leq 1750 + M\gamma \\ 4) \quad & 4x_1 + 2x_2 \leq 4800 + M(1-\gamma) \\ & x_1, x_2 \geq 0 \qquad \gamma \in \{0, 1\} \end{aligned}$$

Fix $\gamma = 0$ then constraint 3 ($x_1 + x_2 \leq 1750 + M\gamma$) becomes $x_1 + x_2 \leq 1750$

Constraint 4 becomes

$$4x_1 + 2x_2 \leq 4800 + M \approx M \textbf{ with } M \textbf{ chosen large enough}$$

This formula can be rewritten as the following:

$$x_2 \leq -2x_1 + \frac{M}{2} \approx M$$
$$x_1 \leq -\frac{1}{2}x_2 + \frac{M}{4} \approx M$$

We already have:

$$x_1 \leq 1000$$
$$x_2 \leq 1500$$

so these add no extra constraints.

Thus, $\gamma = 0$ means constraint 4 goes away.

In the same way, $\gamma = 1$ means constraint 3 goes away.

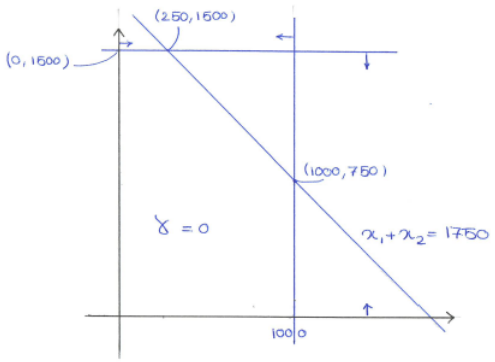


Figure 8.3: Problem with $\gamma = 0$.

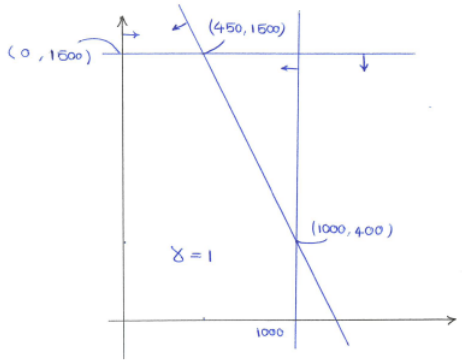


Figure 8.4: Problem with $\gamma = 1$.

With boolean [0,1] we can fix a combination and then solve for that particular combination, and then compare all solutions over all combinations.

Certain combinations may be invalid. If we know beforehand we can disregard those combinations.

However... remember the combinatorial explosion.

EEN025: Constraint Programming and Applied Optimization **Fall 2020**

Lecture 9: Mixed Integer Linear Programming II

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

9.1 Using 0-1 Variables

We cane express choices of the type that we have already seen, like "A before B, or B before A", by using a $0-1$-variable, typically named γ or δ, together with the `Big-M`, which represents a "suitably" large value. What is "suitable" depends on the magnitudes of the decision variables, we want `Big-M` to be as small as possible yet large enough to be able to "switch on/off" constraints.

For larger examples, there may be many 0–1-variables, γ_i, and then requiring that choices should be mutually exclusive, is a matter of summing the choice variables under a constraint

$$\sum_i \gamma_i \leq 1 \qquad \text{(sometimes exactly equal)}$$

Assume that the constraints:

$$\begin{aligned} h_1(x) &\leq b_1 \\ h_2(x) &\leq b_2 \\ &\vdots \end{aligned}$$

should be effective in a mutually exclusive manner, that is, each is effective only under certain circumstances. Then, introducing $0-1$-variables and a `Big-M`:

$$\begin{aligned} h_1(x) - M(1-\gamma_1) &\leq b_1 \\ h_2(x) - M(1-\gamma_2) &\leq b_2 \\ &\vdots \\ \sum_i \gamma_i &\leq 1 \end{aligned}$$

will express this type of mutual exclusion.

Assume $\gamma_j = 1$, then because of the summation, all other $\gamma_i (i \neq j)$ must be zero. Then:

$$h_j(x) - M(1-\gamma_j) = h_j(x) \leq b_j,$$

while for $i \neq j$:

$$\begin{gathered} h_i(x) - M(1-\gamma_i) = h_i(x) - M \leq b_i, \text{ which is equivalent to} \\ h_i(x) - b_i \leq M \end{gathered}$$

will always hold if M is chosen appropriately. Thus, these requirements will have no effect on the solution.

Boolean constraint coding

Let x_i represent (Boolean) propositions

Let $\delta_i = 1$ when x_i holds, $\delta_i = 0$ else ($\delta_i \in \{0, 1\}$)

Then

$$\begin{aligned}
x_i \vee x_j &\iff \delta_i + \delta_j \geq 1 \\
x_i \wedge x_j &\iff \delta_i = 1, \delta_j = 1 \\
\neg x_i &\iff \delta_i = 0 \qquad \iff 1 - \delta_i = 1 \\
x_i \Rightarrow x_j &\iff \delta_i - \delta_j \leq 0 \quad \iff \delta_i \leq \delta_j \\
x_i \Leftrightarrow x_j &\iff \delta_i - \delta_j = 0 \quad \iff \delta_i = \delta_j
\end{aligned}$$

A note on implication

$$(x_1 \vee x_2 \vee \ldots \vee x_n) \Longrightarrow x \tag{9.1}$$

is equivalent to:

$$\neg x \Longrightarrow \neg(x_1 \vee x_2 \vee \ldots \vee x_n)$$

is equivalent to:

$$\neg x \Longrightarrow \neg x_1 \wedge \neg x_2 \wedge \ldots \wedge \neg x_n$$

is equivalent to:

$$(\neg x \Longrightarrow \neg x_1) \wedge (\neg x \Longrightarrow \neg x_2) \wedge \ldots \wedge (\neg x \Longrightarrow \neg x_n)$$

is equivalent to:

$$(x_1 \Longrightarrow x) \wedge (x_2 \Longrightarrow x) \wedge \ldots \wedge (x_n \Longrightarrow x)$$

Which can be coded by $0-1$-variables as:

$$\left.\begin{array}{l} \delta_1 \leq \delta \\ \delta_2 \leq \delta \\ \quad \vdots \\ \delta_n \leq \delta \end{array}\right\}$$

or, as in (10.14), p.214 of [Williams, 2013]

$$\left.\begin{array}{l} \delta_1 - \delta \leq 0 \\ \delta_2 - \delta \leq 0 \\ \quad \vdots \\ \delta_n - \delta \leq 0 \end{array}\right\}$$

which guarantees total unimodularity

Note a subtle differences

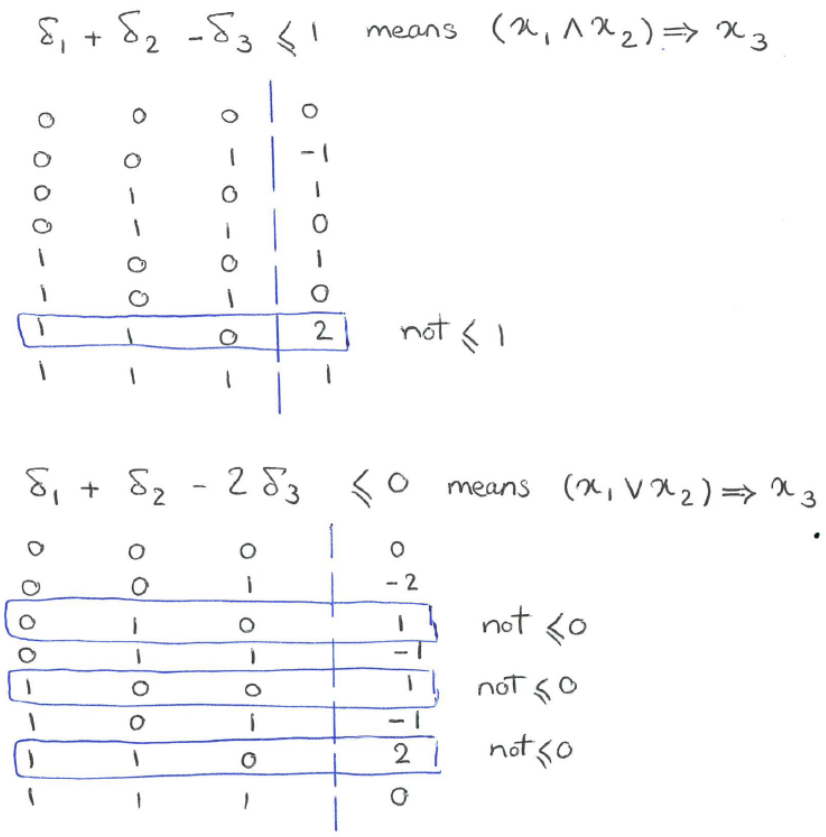


Figure 9.1: Shape 2

Sometimes you feel the need to write something like

$$\delta_i \delta_j = 0$$

This is not linear (unless one of δ_i or δ_j constant, which is not the case we are talking about)

It can be rewritten as

$$\delta_i = 0 \vee \delta_j = 0$$

This is the same as saying

$$\delta_i + \delta_j \leq 1$$

Example

If products A or B (or both) are produced, then at least one of the products C, D, E must also be.

Let x_i (for $i \in \{A, B, C, D, E\}$) denote "product i produced". Then we have the logical condition.

$$(x_A \vee x_B) \Longrightarrow (x_C \vee x_D \vee x_E)$$

Introduce $0-1$-variables δ_i, representing $\delta_i = 1$ if product i produced (x_i true), 0 else

$$\begin{array}{rll} x_A \vee x_B & \text{can then be represented by} & \delta_A + \delta_B \geq 1 \\ x_C \vee x_D \vee x_E & \text{can then be represented by} & \delta_C + \delta_D + \delta_E \geq 1 \end{array}$$

We rewrite $(x_A \vee x_B) \Rightarrow (x_C \vee x_D \vee x_E)$ as

1. $(x_A \vee x_B) \Rightarrow x_{AB}$
2. $x_{AB} \Rightarrow (x_C \vee x_D \vee x_E)$

Now introduce a variable δ_{AB}

Then we get for $(x_A \vee x_B) \Rightarrow x_{AB}$, $\delta_A + \delta_B - 2\delta_{AB} \leq 0$:

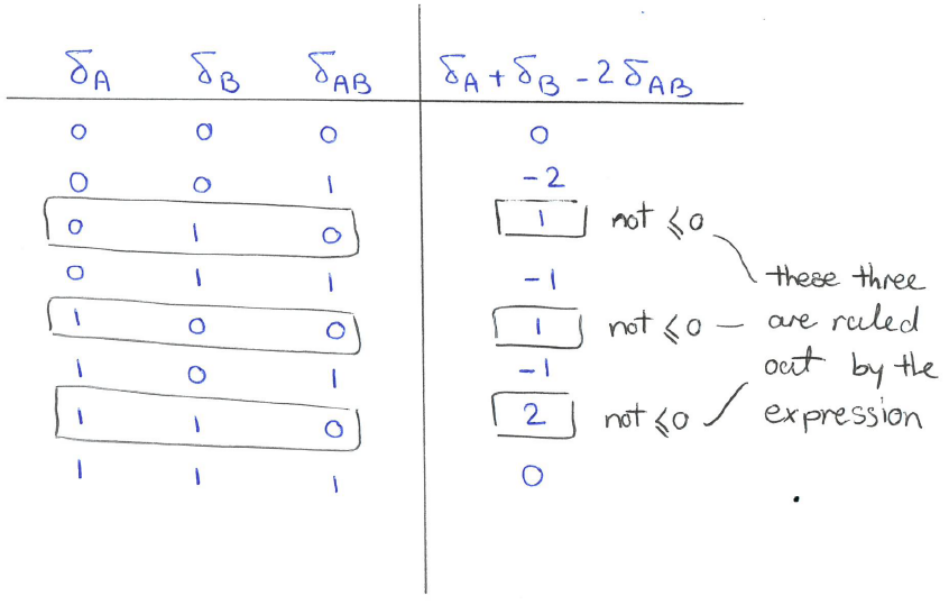


Figure 9.2: Truth table for $\delta_A + \delta_B - 2\delta_{AB} \leq 0$

We also need to impose the condition

$$x_{AB} \Rightarrow (x_C \vee x_D \vee x_E)$$

This we do in the following way:

$$\delta_{AB} \leq \delta_C + \delta_D + \delta_E \Longleftrightarrow \delta_{AB} - \delta_C - \delta_D - \delta_E \leq 0$$

So, we have:

$$\left.\begin{array}{rl} \delta_A + \delta_B - 2\delta_{AB} & \leq 0 \\ \delta_{AB} - \delta_C - \delta_D - \delta_E & \leq 0 \end{array}\right\} \tag{9.2}$$

We could merge these into a single constraint, but should we?

More is Less

> *there is often advantage in increasing rather than decreasing the number of integer variables in a model*
>
> [Williams, 2013], p.210

Consider the following from [Williams, 1978]:

$$(7) \quad \delta_1 + \delta_2 - 2\delta_5 \leq 0$$

$$\begin{cases} (8) & \delta_1 - \delta_5 \leq 0 \\ (9) & \delta_2 - \delta_5 \leq 0 \end{cases}$$

When $\delta_5 = 0$ or $\delta_5 = 1$, then constraint (7), and constraints (8) and (9) together, are equivalent. But if the problem is solved by LP-relaxation, then in-between the upper and lower domain constraints (1 and 0), things look different.

Assume $\delta_5 = 0.5$. Then the constraints look like:

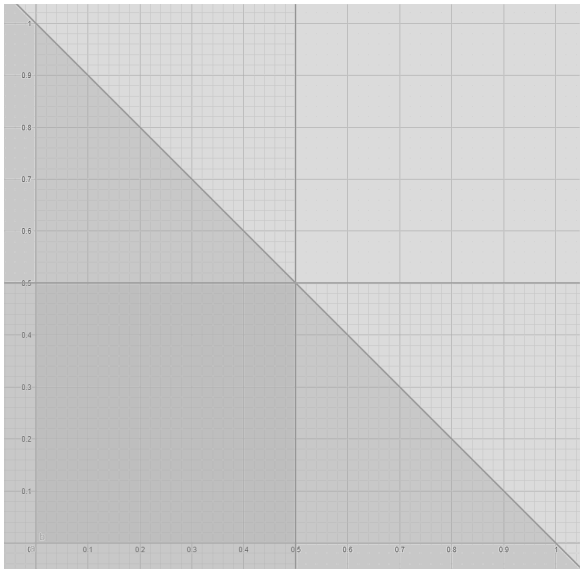


Figure 9.3: Comparison of tightness of constraint (7), and (8) and (9), respectively.

What we see is that the (8) and (9) constraints together give a smaller feasible set than constraint (7). As shown in [Williams, 1978], this results in a much shorter solution time.

Expanding integer constraints into $0-1$ constraints

Sometimes integers can be interpreted as binary numbers:

$$0 \leq x \leq u$$

where x is a non-negative integer, and u is an upper bound.

Then x may be encoded as:

$$\delta_0 + 2\,\delta_1 + 4\,\delta_2 + 8\,\delta_3 + \cdots + 2^n \delta_n$$

where n is such that $2^{n-1} \leq u \leq 2^n$, and the δ_i are $0-1$-variables.

When is Less More?

An important notable exception for when we want to have less, rather than more, variables is for highly symmetric problems, where instead the symmetry is best removed. See Example (10.2) [Williams, 2013].

9.2 Formulating LP/IP/MILP problems

1. Define what seems like the necessary variables
2. Define a set of constraints so the feasible set is closed (and preferably convex)
3. Define the objective function

If difficulties arise, define additional or alternative sets of variables and iterate.

Be picky!

Remember, we are formulating the problem to be solved by a computer. There is no hand-wavy "nudge, nudge, know what I mean, eh?"

Example

- n people to carry out n jobs.
- Each person assigned to exactly one job.
- Cost c_{ij} associated to person i doing job j
- Find minimum cost assignment

Define the variables:

$$x_{ij} = 1, \text{ if person } i \text{ does job } j, \text{ else } x_{ij} = 0$$

Define the constraints:

Each person i assigned to exactly one job:

$$\sum_{j=1}^{n} x_{ij} = 1 \text{ for } i = 1...n$$

Each job j assigned to exactly one person:

$$\sum_{i=1}^{n} x_{ij} = 1 \text{ for } j = 1...n$$

Variables are $0-1$-variables:

$$x_{ij} \in 0, 1 \text{ for } i, j = 1...n$$

Define the objective function:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

We must sum over both "person columns" (exactly one person to each job) and "job rows" (exactly one job to each person):

		Person				
		1	2	...	n	
Job	1					$\Sigma = 1$
	2					$\Sigma = 1$
	$\vdots$					$\vdots$
	n					$\Sigma = 1$
		$\Sigma = 1$	$\Sigma = 1$	...	$\Sigma = 1$	

MILP model, Train/Robot Mutex

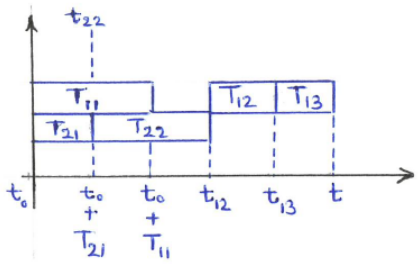


Figure 9.4: Train 2 before Train 1.

t_{12} is the time when operation 12 starts, this must come after the time operation 11 is done

Thus $t_{12} \geq t_{11} + T_{11}$

Where $t_{11} \geq t_0 = 0$

In the same way

$$t_{13} \geq t_{12} + T_{12}$$

and the time when all of process 1 is done must come after the time when operation 13 is done

$$t \geq t_{13} + T_{13}$$

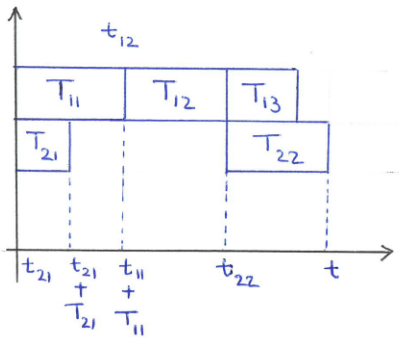


Figure 9.5: Train 1 before Train 2.

t_{22} is the time when operation 22 starts, must come after the time when operation 21 is done

Thus $t_{22} \geq t_{21} + T_{21}$

Where $t_{21} \geq t_0 = 0$

The time when all of process 2 is done, must come after the time when operation 22 is done

$$t \geq t_{22} + T_{22}$$

Mutex constraint

Start of operation 12 must be after operation 22 is done, OR start of operation 22 must be after operation 12 is done

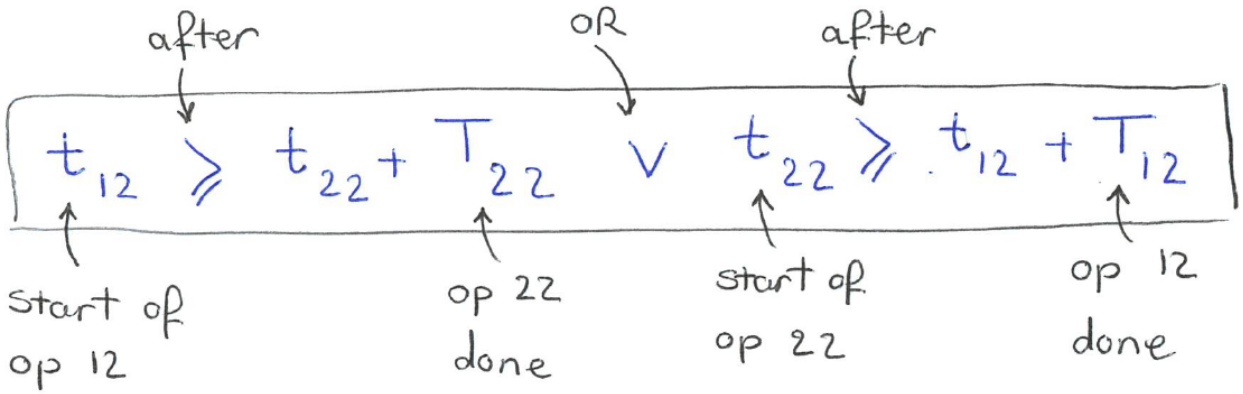


Figure 9.6: Shape 3

This constraint is *not* linear.

Can we make it linear? Yes. with the `Big-M`. Let:

$$x = 1 \text{ for process 1 first in mutex}$$
$$x = 0 \text{ for process2 first in mutex}$$

then

$$t_{12} \geq t_{22} + T_{22} - Mx$$
$$t_{22} \geq t_{12} + T_{12} - M(1 - x)$$

$$x = 1 \Rightarrow \begin{cases} t_{12} \geq -M \\ t_{22} \geq t_{12} + T_{12} \end{cases} \qquad x = 0 \Rightarrow \begin{cases} t_{12} \geq t_{22} + T_{22} \\ t_{12} \geq -M \end{cases}$$

The two constraints with $-M$ add nothing in the respective cases

Other constraints are more restrictive

$$\min t_{done}$$

$$\begin{array}{ll} t_{11} \geq t_0 & t_{21} \geq t_0 \\ t_{12} \geq t_{11} + T_{11} & t_{22} \geq t_{21} + T_{21} \\ t_{13} \geq t_{12} + T_{12} & \\ t_{done} \geq t_{13} + T_{13} & t_{done} \geq t_{22} + T_{22} \end{array}$$

$$\begin{aligned} & t_{12} \geq t_{22} + T_{22} - My \\ & t_{22} \geq t_{12} + T_{12} - M(1-y) \\ & t_i \geq 0, \quad y \in 0, 1 \end{aligned}$$

Here, t_i is the start time of O_p;
T_i is the executing time of O_p;
t_done is when both robots are done
M is a sufficiently large constant
y is a decision variable, 0-1

Manipulating LP formulations

Consider this example

$$\begin{array}{lrrrrrl} \max & 2x_1 & +3x_2 & -x_3 & - & x_4 & \\ \text{s.t.} & x_1 & + & x_2 & + x_3 & -2x_4 & \leq 4 \\ & -x_1 & - & x_2 & + x_3 & - & x_4 \leq 1 \\ & x_1 & + & & & & x_4 \leq 3 \\ & & & & & & x_i \geq 0 \end{array}$$

We want to maximize the objective function

x_3 lowers the value of the objective function

x_3 has only positive coefficients in the constraints

All constraints involving x_3 are the type of $\leq$

Thus, x_3 must optimally be 0, and can be considered redundant

So we have:

$$\begin{array}{rl} \max & 2x_1 + 3x_2 - x_4 \\ \text{s.t.} & x_1 + x_2 - 2x_4 \leq 4 \\ & -x_1 - x_2 - x_4 \leq 1 \\ & x_1 + x_4 \leq 3 \\ & x_i \geq 0 \end{array}$$

Now, the constraints in line three of the formula above becomes redundant

$-(x_1 + x_2 + x_4) \leq 1$ is *always* fulfilled.

We end up with:

$$\begin{array}{rl} \max & 2x_1 + 3x_2 - x_4 \\ \text{s.t.} & x_1 + x_2 - 2x_4 \leq 4 \\ & x_1 + x_4 \leq 3 \\ & x_i \geq 0 \end{array}$$

Which is potentially much easier to solve

If in addition we have a MILP problem

$$\begin{aligned} \max\ & 2x_1 + 3x_2 - x_4 \\ \text{s.t. } & x_1 + x_2 - 2x_4 \leq 4 \\ & x_1 + x_4 \leq 3 \\ & x_i \geq 0 \\ & x_4 \in \{0, 1\} \end{aligned}$$

Then we can solve this as two problems:

One with $x_4 = 0$

$$\begin{aligned} \max\ & 2x_1 + 3x_2 \\ \text{s.t. } & x_1 + x_2 \leq 4 \\ & x_1 \leq 3 \\ & x_i \geq 0 \end{aligned}$$

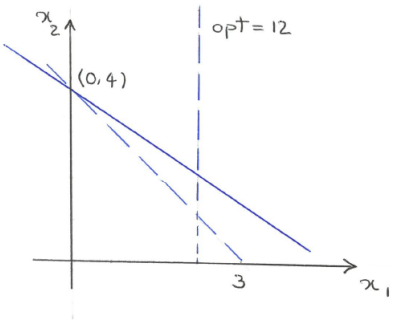


Figure 9.7: $x_4 = 0$

One with $x_4 = 1$

$$\begin{aligned} \max\ & 2x_1 + 3x_2 - 1 \\ \text{s.t. } & x_1 + x_2 - 2 \leq 4 \\ & x_1 + 1 \leq 3 \\ & x_i \geq 0 \end{aligned}$$

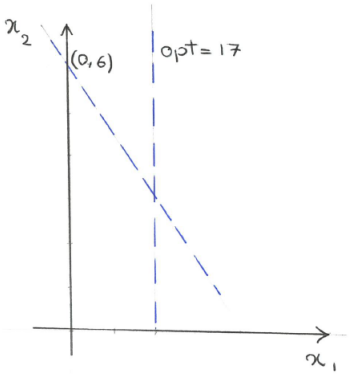


Figure 9.8: $x_4 = 1$

EEN025: Constraint Programming and Applied Optimization **Fall 2020**

Lecture 10: Discrete Optimization I

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

10.1 Discrete optimization

Constraints are given as a *graph*, a general collection of nodes (a.k.a vertices) and edges

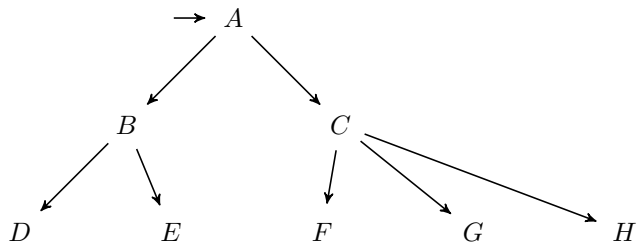
Edges connect nodes, either

- directed (one-directional), or
- undirected (same as bi-directed, both ways)

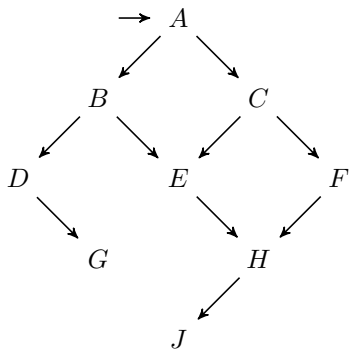
One or more path(s) *may* exist between nodes

A *tree* is a graph with specific properties:

- one initial *root* node, and
- at most *one* path between two nodes



This is a tree.



This is a graph.

A tree is a graph, but not the other way around

A graph can be converted to a tree by duplicating nodes

Trees are easier to search since

- can do efficient branch cutting

Trees are harder to search since

- if you get down the wrong branch there's no way into the right one but to go up

Automata as Graph

An *automaton* is a graph, where:

- the states are the nodes, and
- the transitions are the edges.

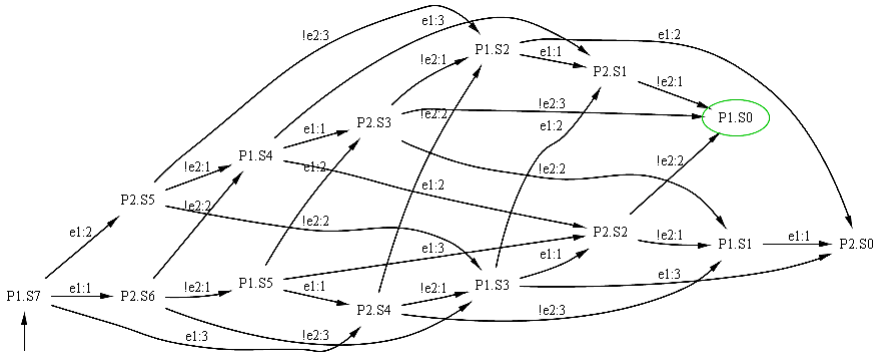


Figure 10.1: An automaton modeling the stick-picking game.

On Costs

A graph can have costs:

- on the edges (what it costs to go from one node to the next)
- in the nodes (what it costs to visit a specific node)

Note We can always convert from one representation to the other

We will consider costs on the edges

Typically costs are non-negative:

- time
- money
- energy
- ...

Then, monotonically increasing, $\text{cost}(A \to B) \leq \text{cost}(A \to B \to C)$

We will consider only non-negative costs

Costs can be:

- uniform, all costs equal (and then might as well be equal to 1)
- non-uniform, some edges have different costs.

Note For uniform costs, shortest path equals least cost

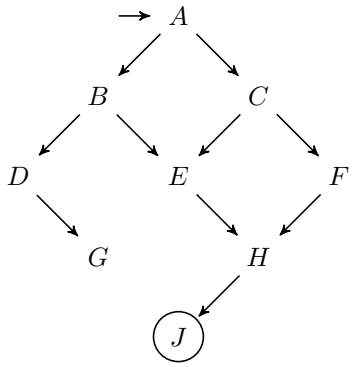
Graph search algorithms

- Breadth/depth first search
- Dijkstra's algorithm
- A^*

These are all *branch and bound* type of algorithms

- Branch, take a decision. Left or right
  Create two (or more) sub-problems
- Bound, compare current cost with best seen cost so far
  Can cut off branches if we know costs are monotonically increasing

Example graph



Breadth/Depth First Search

Uniform costs (let's start with that)

Algorithm

1. $Q.put(q_0)$
2. While not $Q.empty()$
   - $q := Q.get()$
   - if q not already visited
     - set $q.visited$
     - for each q' in $\delta(q, \cdot)$
       $Q.put(q')$

Note: Q is typically called "open"

Note: Typically a queue/list called "closed" keeps track of visited nodes.

Questions:

- How do we "Q.get()"?
- How is the return node picked at?

Two obvious variants:

- add to beginning, pick from end (FIFO queue)
- add to beginning, pick from the beginning (LIFO stack)

SearchAlgorithms.java 2007-07-13

```
/******************************** Breadth First Search
 *  O.put(q0)
 *  while not O.empty()
 *       q := O.get()                          // get the next node
 *       if q == goal, goto done               // if this is the goal, we're done
 *       if not C.find(q)                      // if not already visited (ignore nodes already seen)
 *           C.put(q)                          // mark that we've been here already
 *           for each q' in \delta(q,\sigma)   // extract all nodes reachable from here
 *               O.put(q')                     // put them up for to be visisted
 *               q'.parent := q                // mark q as q's parent
 *  return failure                             // O empty, goal not found
 *
 *  done:
 *  return path extracted from q.parent
 ********************************************************************************/
 /* BFS visits the nodes "levelwise" in the graph, and the first time you ever see a certain node,
  * you know for sure that this is through the shortest path from the init node. Thus, if we keep
  * track of the level in teh graph, we can find teh shortest path to a certain node. We can keep
  * track of teh levels in at least three ways.
  * 1. look to see if the node is already on open, that means its already seen, and seeing it again
  *     must be through a longer (or equal) path
  * 2. set the nodes "parent" the first time it's seen, and check for this; on subsequent seeings we
  *     disregard an already seen node
  * 3. count the levels, and increase the level of a node by the level of its parent node + 1; we must
  *     compare levels to cath subsequent seeings.
  * Note that the third approach easily generalizes to weighted arcs, simply replace "1" by the arc cost
  * Drawbacks:
  * 1. Must search through the open queue, and a queue is typically not efficient for searching
  * 2. Need storage to keep track of the parent
  * 3. Must set all initial costs to infinity, except for the initial node which must have zero cost
  */
```

Figure 10.2: Breadth/depth first search algorithm.

For FIFO, we get breadth first

For LIFO, we get depth first

FIFO
queue

q	Q			
	~~**A**~~			
A	**C**	~~**B**~~		
B	**E**	**D**	~~C~~	
C	**F**	~~E~~	E	~~D~~
D	**G**	F	~~E~~	
E	**H**	G	~~F~~	
F	~~H~~	H	~~G~~	
G	~~H~~			
H	~~J~~			
J				

Breadth first search
BFS

LIFO
stack

q	Q		
	~~**A**~~		
A	~~**C**~~	**B**	
C	~~**F**~~	**E**	B
F	~~**H**~~	E	B
H	~~**J**~~	E	B
J	~~E~~	B	
E	~~H~~	~~B~~	
B	~~**D**~~		
D	~~**G**~~		
G			

Depth first search
DFS

Bold nodes denote the first time seen.

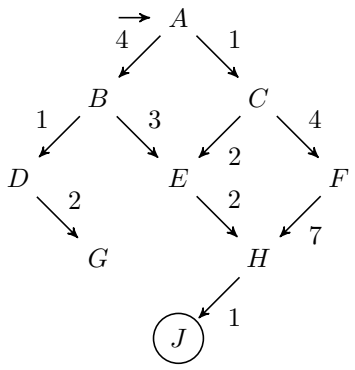
Slashed nodes like ~~C~~ represent the node taken out

Crossed nodes like ~~E~~ are seen but not stored since they are already stored, or already visited

Note: assume G was a *goal node* that we were looking for. BFS guarantees for uniform-cost to find the shortest path $ABDG$, reconstructed via the first seen occasion. In fact, the first time BFS sees the goal node, it is seen via the shortest path.

DFS gives no such guarantees

Let us now try it with non-uniform costs and a goal state:



Note: Now "same" nodes with different costs are not really the same

Find cheapest path to J

Costs accumulate:

$$\text{cost}(A \to B) = 4$$
$$\text{cost}(A \to B \to D) = \text{cost}(A \to B) + \text{cost}(B \to D) = 4 + 1 = 5$$

$ACFHJ : 13$
$ACEHJ : 6 \quad \longleftarrow$ want to find this one
$ABEHJ : 10$
$ABDG : \infty \quad$ never reaches J!

And we want to find it as quickly as possible, looking at as few nodes as possible

BFS

q	Q			
	~~**A:0**~~			
A:0	**C:1**	~~**B:4**~~		
B:4	**E:7**	**D:5**	~~C:1~~	
C:1	**F:5**	**E:3**	~~E:7~~	~~D:5~~
D:5	**G:7**	F:5	~~E:3~~	
E:3	**H:5**	G:7	~~F:5~~	
F:5	~~**H:12**~~	H:5	~~G:7~~	
G:7	~~H:5~~			
H:5	~~**J:6**~~			
(J:6)				

DFS

q	Q		
	~~**A:0**~~		
A:0	~~**C:1**~~	**B:4**	
C:1	~~**F:5**~~	**E:3**	B:4
F:5	~~**H:12**~~	E:3	B:4
H:12	~~**J:13**~~	E:3	B:4
J:13	~~E:3~~	B:4	
E:3	~~**H:5**~~	B:4	
H:5	~~**J:6**~~	B:4	
J:6	~~B:4~~		
B:4	~~**E:7**~~	**D:5**	
E:7	~~**H:9**~~	D:5	
H:9	~~**J:10**~~	D:5	
J:10	~~D:5~~		
D:5	~~**G:7**~~		
G:7			

Bold nodes denote the first time seen.

Slashed nodes like ~~C:1~~ represent the node taken out

Crossed nodes like ~~**H:12**~~ are seen but not stored since they are already stored, or already visited

Backtracking we get: J:6 → H:5 → E:3 → C:1 → A:0

Note, BFS first got J:13 out from Q, but this was *not* the least costly path

BFS/DFS are "uninformed"

"Informed" algorithms generally perform better

$$\left.\begin{array}{l}\text{Dijkstra's}\\ A^*\end{array}\right\} \text{ informed algorithms}$$

Algorithms "guided" by information.

What information?

Dijkstra's algorithm guided by *current cost*. BFS, but nodes are picked out based on current minimal cost.

A^* is guided by current cost, plus *estimate to reach the goal*. So it is Dijkstra's algorithm, but with more information.

Dijkstra's Algorithm

1. $Q.put(q_0)$
2. While not $Q.empty()$

   $q := Q.get\_min()$

   if q is a goal node, goto 3

   if q not already visited

   set $q.visited()$

   for each $q' := \delta(q, \cdot)$

   $Q.put(q')$

   if $g(q') > g(q) + c(q, q')$

   $g(q') := g(q) + c(q, q')$

   $q'.parent := q$
3. Extract path from $q.parent()$

Note: All costs initially set to ∞, except for q_0 for which $g(q_0) = 0$

SearchAlgorithms.java 2007-07-13

```
/****************************** Dijkstras Algo - prioirty queue + guidance
 * O.put(q0)
 * while not O.empty()
 *      q := O.get_min()                    // priority queue
 *      if q == goal, goto done
 *      if not C.find(q)
 *          C.put(q)
 *          for each q' in \delta(q,\sigma)
 *              O.put(q')
 *              if g(q') > g(q)+ c(q,q')    // if the cost to get to q' from init is higher than from init to q
 *                  g(q'):= g(q) + c(q,q')  // plus from q to q' then the path through q is better
 *                  q'.parent := q          // mark this path
 *  return failure                          // O empty, goal not found
 *  done:
 *  return path extracted from q.parent
 ********************************************************************************/
```

Figure 10.3: Dijkstra's algorithm.

To define a notion of a node with a least cost, so that we know which one to get, we must keep track of the costs, $g(q)$ does this.

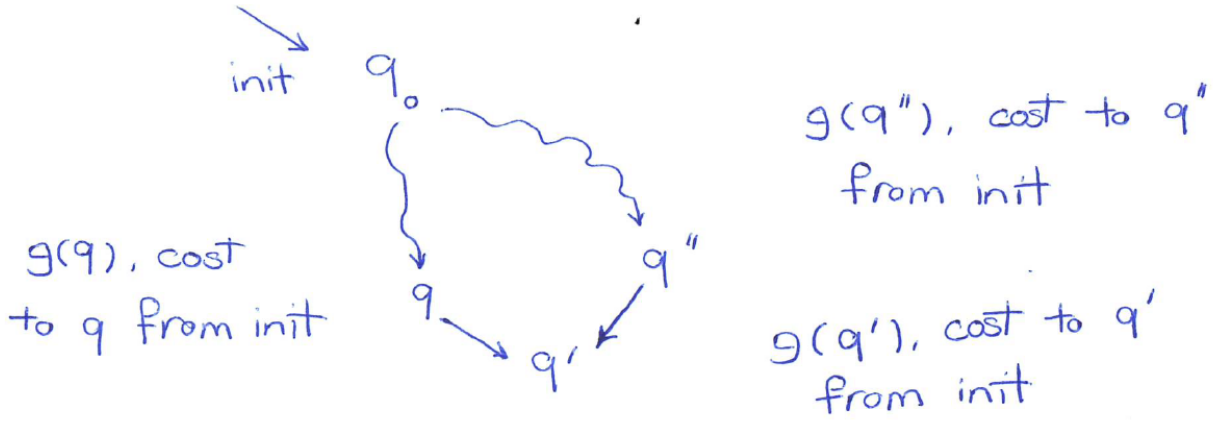


Figure 10.4: $q_0 \to q$ cost more than $q_0 \to q''$, but $q_0 \to q'$ costs less via q than via q''.

We know $g(q)$ and $g(q'')$, have already been there

Assume $g(q) > g(q'')$ then we pick out q'' before q (because of $Q.get\_min()$)

Expand q'', gives us q'

Have not been at q' before, $g(q') \approx \infty$

Thus $g(q') > g(q'') + c(q'', q')$, and so

$g(q') := g(q'') + c(q'', q')$

$q'.parent := q''$ // keep track of how we come here

Next, q is expanded, give again q'

Assume now $g(q') > g(q) + c(q, q')$, then

$g(q') := g(q) + c(q, q')$

$q'.parent := q$ // change the back route

Example graph, Dijkstra's

Dijkstra's algorithm is guided by the current cost

q	Q			
	~~**A:0**~~			
A:0	~~**C:1**~~	**B:4**		
C:1	**F:5**	~~**E:3**~~	B:4	
E:3	**H:5**	F:5	~~B:4~~	
B:4	~~B:7~~	~~**D:5**~~	H:5	F:5
D:5	**G:7**	~~H:5~~	F:5	
H:5	~~**J:6**~~	G:7		
(J:6)	G:7			

Bold nodes denote the first time seen.

Slashed nodes like ~~C:1~~ represent the node taken out

Crossed nodes like ~~B:7~~ are seen but not stored since they are already stored, or already visited

Backtracking we get: $J:6 \to H:5 \to E:3 \to C:1 \to A:0$

Note: at B:4 and D:5, we had several nodes with same smallest cost. The ties were resolved by picking in order, but we could have done it differently. Best would have been to pick H:5 at B:4, but we couldn't know that.

A^*

The A^* algorithm is guided by:

1. the cost, plus
2. an estimate of the cost from the current node to the goal

Nodes are taken out from the queue according to smallest

$$f(n) := g(n) + h(n)$$
$$g(n) : \text{current cost from init to node } n$$
$$h(n) : \text{estimated cost from node } n \text{ to the goal}$$

If $h(n) \leq h^*(n)$ (where $h^*(n)$ is the true cost to goal) then A^* is guaranteed to find the optimum

If $h(n)$ monotone then "first find is best", which means that A^* looks at the smallest possible number of nodes, A^* is itself optimal

Note the difference between "finding the optimum", and "being optimal". Not the same thing (but often confused). An algorithm that enumerated all possible paths, and then compared them one by one and remembered the least costly path, would find the optimum, but that algorithm would not be optimal.

A^* is optimal in the sense that there cannot exist any other algorithm that with the same information, current cost and the estimate, would search fewer nodes than A^* does to find the optimum. This is proven in [Russell and Norvig, 2009].

SearchAlgorithms.java

```
/******************************** A* - priority queue + guidance + estimate
 *  O.put(q0)
 *  while not O.empty()
 *      q := O.get_min()        // get the one with lowest f(q) := g(q) + h(q)
 *      if q == goal, goto done
 *      C.put(q)
 *      if not C.find(q)        // if not already visited
 *          for each q' in \delta(q,\sigma)
 *              g(q') := g(q) + c(q,q')
 *
 *              q'' := C.find(q')
 *              if q'' and q'' <= q'    // better cost seen already?
 *                  continue            // the one already on C is better
 *
 *              q'' := O.find(q')
 *              if q'' and q'' <= q'    // better cost seen already?
 *                  continue            // the one already on O is better
 *
 *              parent(q') := q         // else, this is the best cost so far
 *              O.put(q')               // (re-)open for new examination
 *
 *  return failure                  // O empty, goal not found
 *  done:
 *  return path extracted from q.parent
 ************************************/
```

Figure 10.5: A^* in Java

The same node can turn up with different costs

Need to compare for best cost

Sometimes we may need to "re-open" a node as we may get fooled by the heuristic

A^* priority queue

- From OPEN, select node with min $f = g + h$
- Several nodes with same f may exit
- If OPEN returns among same f nodes
  - LIFO, "depth first search" among same f paths
  - FIFO, "breadth first search" among same f paths

A^* heuristics

- Dijkstra's algorithm is A^* with $h(n) = 0$ for all nodes
- To guarantee to find the optimum the heuristic must not over estimate:

  $$h \leq h^*$$

  the heuristic is then said to be *admissible*
- If the heuristic is *monotone*:

  $$h(x) \leq c(x, y) + h(y)$$

  then A^* is itself *optimal*, meaning it looks at the least number of nodes
- Just like `Big-M` is an estimate of an *upper* bound, $h(n)$ is an estimate of a *lower* bound
- The tighter the estimate, both for `Big-M` and $h(n)$, the easier it gets for the solver to find the optimum, smaller relaxed feasible set, less nodes to be looked at
- Heuristics are application dependent

Example graph, A^*

q	Q		
	A:0		
A:0	~~**C:1**~~	**B:4**	
C:1	**F:5**	~~**E:3**~~	B:4
E:3	~~**H:5**~~	F:5	B:4
H:5	**J:6**	F:5	B:4
(J:6)	F:5	B:4	

estimates	
A	4
B	3
C	3
D	∞
E	2
F	2
G	∞
H	1
J	0

Estimates, $h(n)$, is the number of steps to J. Obviously $h(n) \leq h^*(n)$

Bold nodes denote the first time seen.

Slashed nodes like ~~C:1~~ represent the node taken out

Backtracking we get: $J:6 \rightarrow H:5 \rightarrow E:3 \rightarrow C:1 \rightarrow A:0$

Note: at E:3, H:5 was picked because current cost plus estimate was H:$5+1=6$, compared to F:$5+2=7$ and B:$4+3=7$.

Summary of a sort

- Breadth first search — uninformed
- Depth first search — uninformed
- Dijkstra — uses min cost so far as heuristics
- A^*— uses min cost so far + guess of cost to go

Robot system heuristics – special case

- Robots sharing resources
- "Products" visiting "machines" – job shop
- Total cycle time to optimize
- Total time can never be shorter than the longest robot path running by itself
- Total time can never be shorter than max of two robots by themselves. (This can also be the case for three or more robots)
- Can be calculated exactly (by Dijkstra's (or even A^*), for instance!)

Two Robots/Two Trains Example

Let us examine the example of two robots working in shared space, a.k.a. two trains sharing a track, to see a little more realistic example of how good heuristics can improve the search.

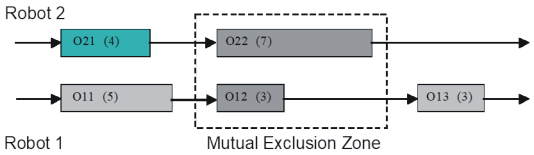


Figure 10.6: Two robots sharing mutual work space.

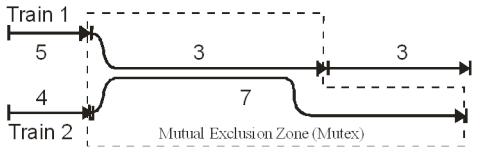


Figure 10.7: Two trains sharing mutual track.

We can model this as a job-shop problem, with two "jobs" (the robots/trains) using four "machines" (the work-spaces/track-segments). The first job uses machine M_1 followed by M_3 and then M_4, while the second job uses M_2 followed by M_3. So, M_3 is the shared mutually exclusive machine.

We can model the jobs in the following way, using SUPREMICA as a modeling tool. The graphs are then *extended finite state-machines*, which makes it easy to build a global model out of smaller "local" models.

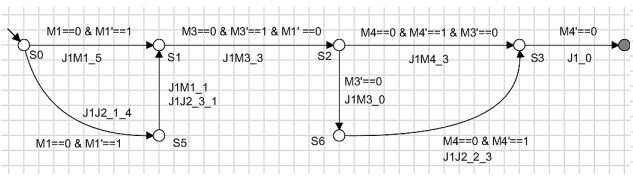


Figure 10.8: Graph model of job J_1

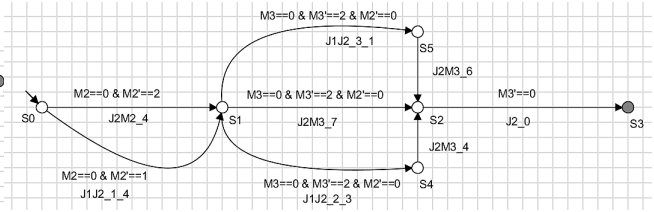


Figure 10.9: Graph model of job J_2

In these models, going from one state to the next is conditioned on the availability of the machines. For instance, for J_2 to access M_3, M_3 has to be available ($M3 == 0$) and if M_3 is available, one of the three transitions is taken and M_3 is claimed for J_2 ($M3' == 2$) and at the same time M_2 is released by J_2 ($M2' == 0$). Thus the prime represents the next value.

The last number of an event represents the time, so the event J2M3_7 represents that J_2 uses M_3 for 7 time units. Events that start J1J2 represent that the two jobs work in parallel. These are the events that the model *synchronize* on when the global *monolithic* model is built.

Composing the local models to build the global model results in the following graph:

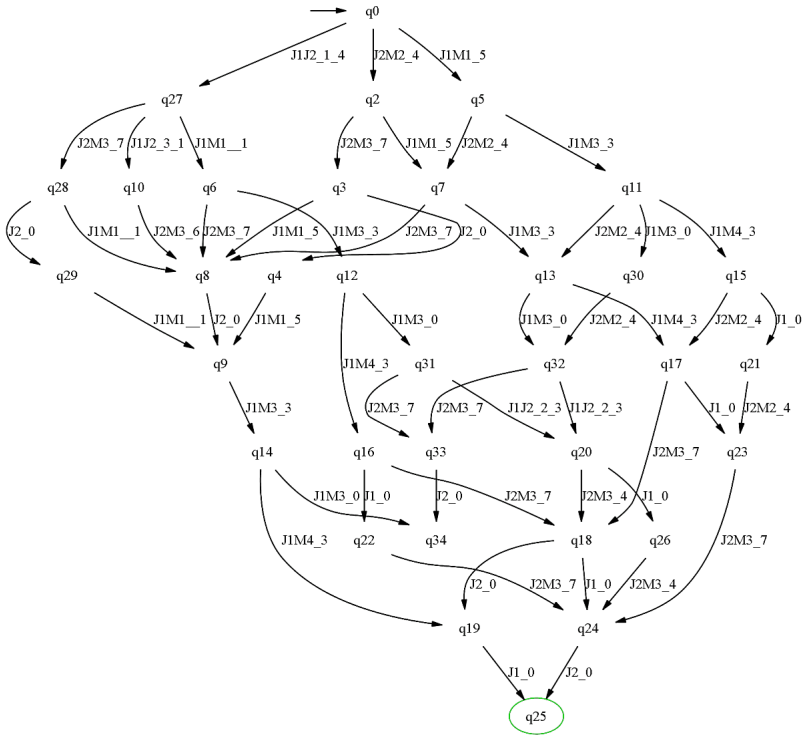


Figure 10.10: Search graph for the two robots (trains) job-shop example.

This graph describes all the possible ways that the two jobs can run through this four machine system, including ways that we are not really interested in, like first J_1 runs to completion and only then J_2 starts, which is given by the outer right path.

Running Dijkstra's algorithm on this graph, we get the following:

```
Dijkstras -- Two Robots (JSP) --
q0[0]:   q27[4]: q5[5]: q2[4]: :
q27[4]:  q2[4]: q5[5]: q10[5]: q6[5]: q28[11]: :
q2[4]:   q5[5]: q6[5]: q10[5]: q28[11]: q7[9]: q3[11]: :
q5[5]:   q6[5]: q7[9]: q10[5]: q28[11]: q3[11]: q11[8]: :
q6[5]:   q10[5]: q7[9]: q11[8]: q28[11]: q3[11]: q12[8]: q8[12]: :
q10[5]:  q11[8]: q7[9]: q12[8]: q28[11]: q3[11]: q8[11]: :
q11[8]:  q12[8]: q7[9]: q30[8]: q28[11]: q3[11]: q8[11]: q15[11]: q13[12]: :
q12[8]:  q30[8]: q31[8]: q8[11]: q7[9]: q3[11]: q13[12]: q15[11]: q28[11]: q16[11]: :
q30[8]:  q31[8]: q7[9]: q8[11]: q16[11]: q3[11]: q13[12]: q15[11]: q28[11]: q32[12]: :
q31[8]:  q7[9]: q16[11]: q8[11]: q28[11]: q3[11]: q13[12]: q15[11]: q32[12]: q20[11]: q33[15]: :
q7[9]:   q16[11]: q28[11]: q8[11]: q20[11]: q3[11]: q13[12]: q15[11]: q32[12]: q33[15]: :
q16[11]: q28[11]: q20[11]: q8[11]: q22[11]: q3[11]: q13[12]: q15[11]: q33[15]: q32[12]: q18[18]: :
q28[11]: q20[11]: q22[11]: q8[11]: q32[12]: q3[11]: q13[12]: q15[11]: q33[15]: q18[18]: q29[11]: :
q20[11]: q29[11]: q22[11]: q8[11]: q26[11]: q3[11]: q13[12]: q15[11]: q33[15]: q32[12]: q18[15]: :
q29[11]: q22[11]: q26[11]: q8[11]: q32[12]: q3[11]: q13[12]: q15[11]: q33[15]: q18[15]: q9[12]: :
q22[11]: q26[11]: q3[11]: q8[11]: q32[12]: q9[12]: q13[12]: q15[11]: q33[15]: q18[15]: q24[18]: :
q26[11]: q3[11]: q32[12]: q8[11]: q33[15]: q9[12]: q13[12]: q15[11]: q18[15]: q24[15]: :
q3[11]:  q8[11]: q4[11]: q15[11]: q32[12]: q9[12]: q13[12]: q24[15]: q18[15]: q33[15]: :
q8[11]:  q4[11]: q9[11]: q15[11]: q32[12]: q18[15]: q13[12]: q24[15]: q33[15]: :
q4[11]:  q9[11]: q32[12]: q15[11]: q33[15]: q18[15]: q13[12]: q24[15]: :
q9[11]:  q15[11]: q32[12]: q13[12]: q33[15]: q18[15]: q24[15]: q14[14]: :
q15[11]: q21[11]: q14[14]: q32[12]: q33[15]: q18[15]: q24[15]: q13[12]: q17[15]: :
q21[11]: q32[12]: q14[14]: q13[12]: q33[15]: q18[15]: q24[15]: q17[15]: q23[15]: :
q32[12]: q13[12]: q14[14]: q23[15]: q33[15]: q18[15]: q24[15]: q17[15]: :
q13[12]: q14[14]: q17[15]: q23[15]: q33[15]: q18[15]: q24[15]: :
q14[14]: q34[14]: q17[15]: q24[15]: q33[15]: q18[15]: q23[15]: q19[17]: :
q34[14]: q17[15]: q33[15]: q24[15]: q19[17]: q18[15]: q23[15]: :
q17[15]: q23[15]: q33[15]: q24[15]: q19[17]: q18[15]: :
q23[15]: q18[15]: q33[15]: q24[15]: q19[17]: :
q18[15]: q33[15]: q24[15]: q19[15]: :
q33[15]: q19[15]: q24[15]: :
q19[15]: q24[15]: q25[15]: :
q24[15]: q25[15]: :
q25[15]: :

Goal node found --
```

```
q25[15]: q19[15]: q18[15]: q20[11]: q31[8]: q12[8]: q6[5]: q27[4]: q0[0]:
```

Using the heuristic that the time to the goal from a node cannot be shorter than what remains for the longest job from that node, we get the following with A^*:

```
A* -- Two Robots (JSP) --
q0[0,11]:  q27[4,7]: q5[5,11]: q2[4,11]: :
q27[4,7]:  q10[5,6]: q6[5,7]: q2[4,11]: q5[5,11]: q28[11,7]: :
q10[5,6]:  q6[5,7]: q5[5,11]: q2[4,11]: q28[11,7]: q8[11,6]: :
q6[5,7]:   q2[4,11]: q12[8,7]: q8[11,6]: q28[11,7]: q5[5,11]: :
q2[4,11]:  q12[8,7]: q5[5,11]: q8[11,6]: q28[11,7]: q7[9,7]: q3[11,11]: :
q12[8,7]:  q31[8,7]: q7[9,7]: q5[5,11]: q28[11,7]: q3[11,11]: q8[11,6]: q16[11,7]: :
q31[8,7]:  q20[11,4]: q16[11,7]: q7[9,7]: q28[11,7]: q3[11,11]: q8[11,6]: q5[5,11]: q33[15,3]: :
q20[11,4]: q26[11,4]: q18[15,0]: q5[5,11]: q7[9,7]: q3[11,11]: q8[11,6]: q33[15,3]: q28[11,7]: q16[11,7]: :
q26[11,4]: q18[15,0]: q24[15,0]: q5[5,11]: q7[9,7]: q3[11,11]: q8[11,6]: q33[15,3]: q28[11,7]: q16[11,7]: :
q18[15,0]: q24[15,0]: q19[15,0]: q5[5,11]: q7[9,7]: q3[11,11]: q8[11,6]: q33[15,3]: q28[11,7]: q16[11,7]: :
q24[15,0]: q19[15,0]: q25[15,0]: q5[5,11]: q7[9,7]: q3[11,11]: q8[11,6]: q33[15,3]: q28[11,7]: q16[11,7]: :
q19[15,0]: q25[15,0]: q7[9,7]: q5[5,11]: q16[11,7]: q3[11,11]: q8[11,6]: q33[15,3]: q28[11,7]: :
q25[15,0]: q7[9,7]: q28[11,7]: q5[5,11]: q16[11,7]: q3[11,11]: q8[11,6]: q33[15,3]: :

The results of A* --
q25[15,0]: q24[15,0]: q26[11,4]: q20[11,4]: q31[8,7]: q12[8,7]: q6[5,7]: q27[4,7]: q0[0,11]:
```

Lecture 11: Discrete Optimization II

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

11.1 Admissible heuristic

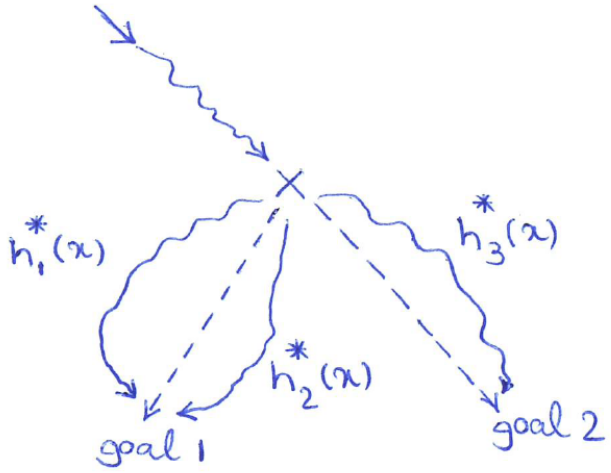


Figure 11.1: Admissible heuristic

A heuristics estimate is *admissable*, if it never over-estimates the cost of any goal.

$$h(x) \leq \min_i h_i^*(x)$$

Here, $h_i^*(x)$ is the true cost from x to a goal along the i'th path. The true optimum is $h^*(x) = \min_i h_i^*(x)$

The estimate is a *lower bound* on the true cost.

If h is admissable, then A^* will find the optimal path.

Monotone heuristics

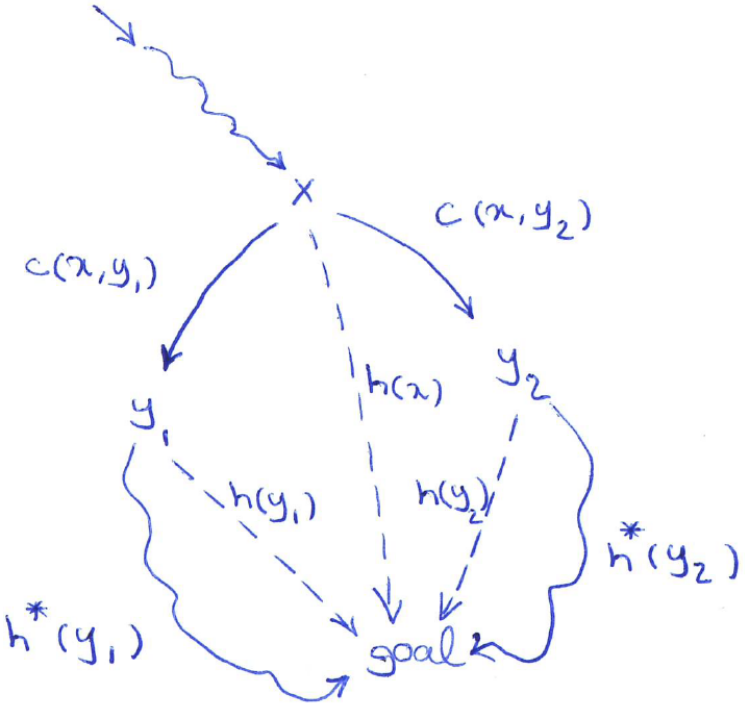


Figure 11.2: Monotone heuristic.

A heuristic is *monotone*, if taking a known step does not allow to guess significantly better

$$h(x) \leq \min_i \text{cost}(x, y_i) + h(y_i)$$

Here, y_i is the i'th node reachable from x in one step

$h(x) - \text{cost}(x, y_i) \leq h(y_i)$ the guess from y_i cannot be better (smaller) than the guess from x minus the cost of the known step

Compare the *triangle inequality*

If h is monotone, then A^* will search as few nodes as possible (never needs to step back)

Relation between monotonicity, admissable

Assuming $h(goal) = 0$, a reasonable assumption,

$$\text{monotonicity} \implies \text{admissable}$$

This is equivalent to:

$$\text{not admissable} \implies \text{not monotone}$$

Note: h can be admissable without being monotone:

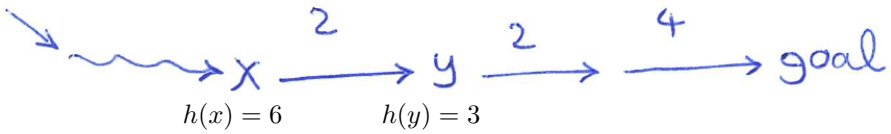


Figure 11.3: Admissible but not monotone heuristic example.

$$\text{Assuming: } \left.\begin{array}{l} h(x) = 6 \\ h(y) = 3 \end{array}\right\} \begin{array}{l} \text{is admissable (no over-estimation)} \\ \text{but not monotone (} h(y) < h(x) - c(x,y) \text{)} \end{array}$$

The only way for one node to over-estimate, with h still being monotone, is for *all succeeding nodes* (including the goal) to also over-estimate.

EEN025: Constraint Programming and Applied Optimization **Fall 2020**

Lecture 12: Constraint Programming I

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

12.1 Constraint Programming

A.k.a. Constraint *Satisfaction* Programming, or even Constraint Satisfaction Problem

Alternative approach to "programming" (as in "mathematical programming")

Developed in CS community, mainly European, since mid-1990's (so still just teenage)

Uses a combination of reasoning and computing

CP has shown to be remarkably efficient, still evolving

CP = model + search

Model consists of *variables* and *constraints*

"Search" draws conclusions from current values and domains, and the constraints

When all conclusions are made

- Make a guess
- Draw new conclusions
- When inconsistency detected, backtrack

Model formulation

Triplet $\langle X, D, C \rangle$ where

- X, a set of decision variables, $x_i \in X$
- D, set of domain constraints, D_i for variable x_i
- C, set of (global) constraints over a (sub)set of the variables

Domains

A *domain* is (without loss of generality) a finite set of discrete values, $D_i \subseteq \mathbb{Z}$

The domain D of a CSP is given by:

$$D_1 \times D_2 \times ... \times D_n$$

for decision variables $x_1, x_2, ...x_n$

Note that if $\exists D_i = \emptyset$, then $D_1 \times D_2 \times ... \times D_n = \emptyset$

$$\begin{aligned} \text{A domain has a lower bound, } & \min(D_i) \\ \text{and an upper bound, } & \max(D_i) \end{aligned}$$

with $\min(D_i) \leq \max(D_i)$ ([Apt, 2003] calls these l_i and h_i, respectively).

CSP solving can take advantage of domains being intervals (ILOG CP) but there is no loss of generality to regard them as discrete values

Variables

A (decision) variable x_i has a domain D_i

- x_i is *bound* when $|D_i| = 1$
- x_i is *inconsistent* when $|D_i| = 0 \quad (\iff D_x = \emptyset)$
- x_i is *free* when $|D_i| \geq 2$

The variable x_i takes values from D_i

We use $\min(x_i)$ to mean $\min(D_x)$, and similar for max

For a *subdomain* $\delta \subseteq D_1 \times D_2 \times ... \times D_n$, $\delta(x_i)$ is the set of values in δ corresponding to D_i:

$$\begin{aligned} D_1 &= \{0..1\} \\ D_2 &= \{2..3\} \\ D_1 \times D_2 &= \{\langle 0,2\rangle, \langle 0,3\rangle, \langle 1,2\rangle, \langle 1,3\rangle\} \\ \text{Let } \delta &= \{\langle 0,2\rangle, \langle 1,2\rangle\} \\ \text{Then } \delta(x_1) &= \{0,1\} \\ \delta(x_2) &= \{2\} \end{aligned}$$

Constraints

A *constraint* is a relation over a subset of the variables, a logical formula defining the allowed combinations of values for the involved variables

A constraint can be defined by (se also [Apt, 2003] p.55):

- *intension*, defining an explicit relation between the variables: $x \neq y; \quad D_x = \{1,2\}, D_y = \{2,3\}$
- *extension*, defining the valid tuples: $\{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$

Given a sub-domain $\delta \subseteq D_1 \times D_2 \times ... \times D_n$, for which $|\delta(x_i)| = 1$

A constraint $c \in C$ can be evaluated on δ, $c(\delta)$ to be either true or false:

$$\begin{aligned} \text{Let } D_x &= \{0\,..\,3\} \\ D_y &= \{1\,..\,3\} \\ c &:= (3x - 5y = 4) \\ \delta_1 &= \{\langle 0,2\rangle\} \\ \delta_2 &= \{\langle 3,1\rangle\} \end{aligned}$$

Then:

$$\begin{aligned} c\,(\delta_1) &= 3 \times 0 - 5 \times 2 = -7 \neq 4, \text{ is false} \\ c\,(\delta_2) &= 3 \times 3 - 5 \times 1 = 4, \text{ is true} \end{aligned}$$

CSP Solution

Given a set of decision variables $X = \{x_1, x_2, ..., x_n\}$ with domain $D = D_1 \times D_2 \times ... \times D_n$, a sub-domain $\delta \subseteq D$ such that

$$\forall x_i \in X \quad |\delta(x_i)| = 1$$

is said to be a *solution*.

CSP feasible solution

Given a CSP $\langle X, D, C \rangle$, a solution $\delta \subseteq D$ is *feasible* if

$$\bigwedge_{c \in C} c(\delta) \text{ is true}$$

CSP solution set

Given a CSP $\langle X, D, C \rangle$, the *solution set*

$$S(\langle X, D, C \rangle) = \{\delta_i \in D \,|\, \forall x \in X \; |\delta_i(x)| = 1 \wedge \bigwedge_{c \in C} c(\delta_i)\}$$

That is, the solution set is the set of all feasible solutions

A CSP can be

- *consistent*, a solution (or more than one) exists
- *inconsistent*, no solution exists
- *solved*, the domains are singletons and together a solution to the CSP
- *failed*, one (or more) empty domains, or only singleton domains that are together not a solution to the CSP

A failed CSP is inconsistent.

CSP Solving

- Reducing domains
- Propagating constraints
- Transforming constraints
- Introducing new constraints
- Removing redundant constraints

All these procedures aim to tighten the bounds on the problem, until only a solution is left.

Most of the procedures are rather straightforward, yet applying them iteratively lead to powerful solution procedures.

Constraint propagation

Propagators look at current domains, and make deductions

Example:

$$x + 2y < z \qquad x, y \in \{2\,..\,10\}, z \in \{4\,..\,10\}$$

$$\left.\begin{array}{rl} \max(z) & = 10 \\ \min(y) & = 2 \end{array}\right\} \Longrightarrow \max(x) = \max(z) - 2\min(y) = 6$$

$$\left.\begin{array}{rl} \max(z) & = 10 \\ \min(x) & = 2 \end{array}\right\} \Longrightarrow \max(y) = \frac{1}{2}(\max(z) - \min(x)) = 4$$

So now we know that:

$$x \in \{2\,..\,6\}, \quad y \in \{2\,..\,4\}, \quad z \in \{4\,..\,10\}.$$

Originally we had: $9 \times 9 \times 7 = 567$ combinations

After constraint propagation: $5 \times 3 \times 7 = 105$ combinations

And we can do better:

$\min(x) + 2\min(y) = 6$

which means that $\min(z) = 6$ (actually 7. Why?)

Thus, $x \in \{2\,..\,6\}, y \in \{2\,..\,4\}, z \in \{6\,..\,10\}$

Only 75 combinations left!

More constraint propagation

An important constraint is `allDifferent` (see the 42(!) pages on this constraint in [Hoeve, 2001]), which requires that all the variables it applies to have different values.

`allDifferent` (x, y, z, v)

$$x \in \{1,2\} \quad y \in \{1,2\} \quad z \in \{1,2,3,4\} \quad v \in \{2,4\}$$

Since x and y are either 1 or 2, respectively,

$$z \in \{\cancel{1},\cancel{2},3,4\} \quad v \in \{\cancel{2},4\}$$
$$x \in \{1,2\} \quad y \in \{1,2\} \quad z \in \{3,4\} \quad v \in \{4\}$$

Since v can only be 4, z is not

$$x \in \{1,2\} \quad y \in \{1,2\} \quad z = 3 \quad v = 4$$

n-Queens

The canonical example for `allDifferent` is the n-Queens problem, where on an $n \times n$ chess board we are to place n queens in such a way that they do not attack each other. We cannot have two queens on the same row, in the same column or the same diagonal.

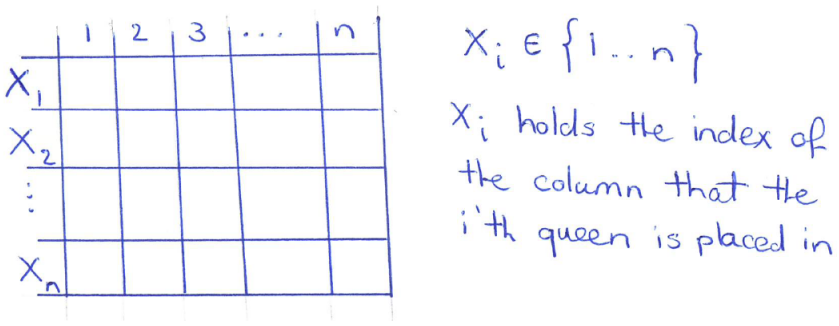


Figure 12.1: $n \times n$ chess board.

- Not two queens in the same column

$$\begin{array}{llll} x_2 \neq x_1, & x_3 \neq x_1, & \ldots, & x_n \neq x_1 \\ & x_3 \neq x_2, & \ldots, & x_n \neq x_2 \\ & \ddots & & \vdots \\ & & & x_n \neq x_{n-1} \end{array}$$

  This can be posed as: `allDifferent` $(x_1, x_2, \ldots, x_n)$.

- Not two queens on the same left diagonal

$$\begin{array}{llll} x_2 \neq x_1 - 1, & x_3 \neq x_1 - 2, & \ldots, & x_n \neq x_1 - (n-1) \\ & x_3 \neq x_2 - 1, & \ldots, & x_n \neq x_2 - (n-2) \\ & \ddots & & \vdots \\ & & & x_n \neq x_{n-1} - 1 \end{array}$$

  This is $x_j \neq x_i - (j - i)$ for $1 \leq i < j \leq n$

Add j to both sides of $x_j \neq x_i - (j - i)$ to get $x_j + j \neq x_i + i$:

$$\begin{array}{llll} x_2 + 2 \neq x_1 + 1, & x_3 + 3 \neq x_1 + 1, & \ldots, & x_n + n \neq x_1 + 1 \\ & x_3 + 3 \neq x_2 + 2, & \ldots, & x_n + n \neq x_2 + 2 \\ & \ddots & & \vdots \\ & & & x_n + n \neq x_{n-1} + (n-1) \end{array}$$

And this can then be posed as: `allDifferent` $(x_1 + 1, x_2 + 2, \ldots, x_n + n)$

- Not two queens on the same right diagonal

$$\begin{array}{llll} x_2 \neq x_1 + 1, & x_3 \neq x_1 + 2, & \ldots, & x_n \neq x_1 + (n-1) \\ & x_3 \neq x_2 + 1, & \ldots, & x_n \neq x_2 + (n-2) \\ & \ddots & & \vdots \\ & & & x_n \neq x_{n-1} + 1 \end{array}$$

  Subtract j from both sides of $x_j \neq x_i + (j - i)$ to get $x_j - j \neq x_i - i$:

$$\begin{array}{llll} x_2 - 2 \neq x_1 - 1, & x_3 - 3 \neq x_1 - 1, & \ldots, & x_n - n \neq x_1 - 1 \\ & x_3 - 3 \neq x_2 - 2, & \ldots, & x_n - n \neq x_2 - 2 \\ & \ddots & & \vdots \\ & & & x_n - n \neq x_{n-1} - (n-1) \end{array}$$

  Thus: `allDifferent` $(x_1 - 1, x_2 - 2, \ldots, x_n - n)$

5-queens diagonal constraints

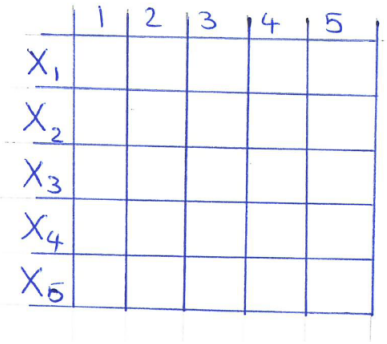


Figure 12.2: 5×5 chess board

Left diag $x_j \neq x_i - (j - i) \quad \Leftrightarrow$

$$\begin{array}{llll} x_2 \neq x_1 - 1, & x_3 \neq x_1 - 2, & x_4 \neq x_1 - 3, & x_5 \neq x_1 - 4 \\ & x_3 \neq x_2 - 1, & x_4 \neq x_2 - 2, & x_5 \neq x_2 - 3 \\ & & x_4 \neq x_3 - 1, & x_5 \neq x_3 - 2 \\ & & & x_5 \neq x_4 - 1 \end{array}$$

Right diag $x_j \neq x_i + (j - i)$

$$\begin{array}{llll} x_2 \neq x_1 + 1, & x_3 \neq x_1 + 2, & x_4 \neq x_1 + 3, & x_5 \neq x_1 + 4 \\ & x_3 \neq x_2 + 1, & x_4 \neq x_2 + 2, & x_5 \neq x_2 + 3 \\ & & x_4 \neq x_3 + 1, & x_5 \neq x_3 + 2 \\ & & & x_5 \neq x_4 + 1 \end{array}$$

Right diag, subtract n from both sides of $x_n \neq \ldots$

$$\begin{array}{llll} x_2 - 2 \neq x_1 - 1, & x_3 - 3 \neq x_1 - 1, & x_4 - 4 \neq x_1 - 1, & x_5 - 5 \neq x_1 - 1 \\ & x_3 - 3 \neq x_2 - 2, & x_4 - 4 \neq x_2 - 2, & x_5 - 5 \neq x_2 - 2 \\ & & x_4 - 4 \neq x_3 - 3, & x_5 - 5 \neq x_3 - 3 \\ & & & x_5 - 5 \neq x_4 - 4 \end{array}$$

Results in $x_n - n \neq x_m - m (for n \neq m)$

This can now be written as

$$\text{allDifferent}(x_1 - 1, x_2 - 2, ..., x_n - n)$$

Left diag add n to both sides of $x_n \neq ...$

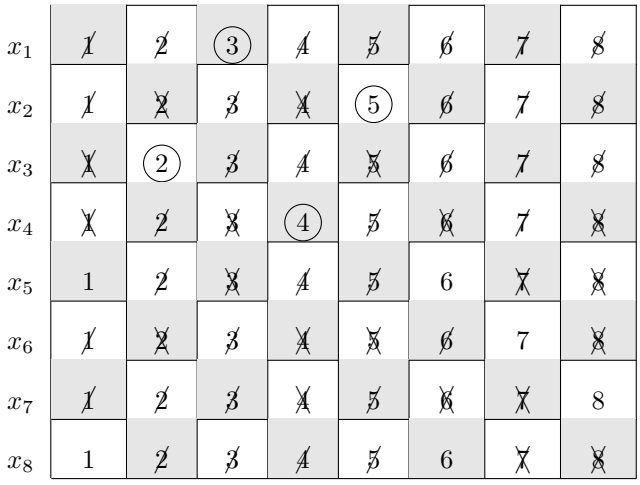
$$\begin{array}{llll} x_2 + 2 \neq x_1 + 1, & x_3 + 3 \neq x_1 + 1, & x_4 + 4 \neq x_1 + 1, & x_5 + 5 \neq x_1 + 1 \\ & x_3 + 3 \neq x_2 + 2, & x_4 + 4 \neq x_2 + 2, & x_5 + 5 \neq x_2 + 2 \\ & & x_4 + 4 \neq x_3 + 3, & x_5 + 5 \neq x_3 + 3 \\ & & & x_5 + 5 \neq x_4 + 4 \end{array}$$

This can now be written as

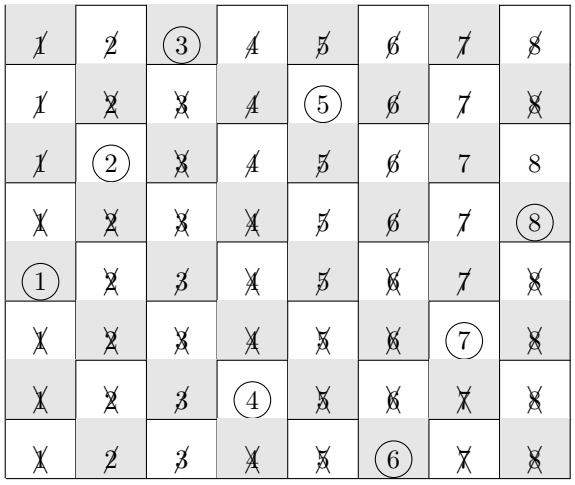
$$\text{allDifferent}(x_1 + 1, x_2 + 2, ..., x_n + n)$$

8-queens

allDifferent$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$
allDifferent$(x_1 - 1, x_2 - 2, x_3 - 3, x_4 - 4, x_5 - 5, x_6 - 6, x_7 - 7, x_8 - 8)$
allDifferent$(x_1 + 1, x_2 + 2, x_3 + 3, x_4 + 4, x_5 + 5, x_6 + 6, x_7 + 7, x_8 + 8)$



Here we picked $x_4 = 4$
This leaves only $x_6 = 7$
and $x_7 = 8$, which is
NOT GOOD



Here we picked $x_4 = 8$
But then we had to be
careful with x_6, x_7, x_8, so
We set $x_5 = 1$

Lecture 13: Constraint Programming II

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

13.1 Inequality constraints (Apt, 6.4.1)

$$\sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \leq b$$

- a_i positive integer for $i \in POS \bigcup NEG$
- x_i and x_j are different variables for $i, j \in POS \bigcup NEG$ and $i \neq j$
- b is integer

POS is the set of indices for the variables with positive contribution to the constraint

NEG is the set of indices for the variables with negative contribution to the constraint

The constraint is then said to be *normalized*.

Assume the domains are integer intervals

$$x_i \in \min(x_i) \,..\, \max(x_i)$$

Pick a variable x_j with $j \in POS$

Then we can write for the linear inequality constraint:

$$x_j \leq \frac{b - \sum_{i \in POS \setminus j} a_i x_i + \sum_{i \in NEG} a_i x_i}{a_j}$$

where $POS \setminus j$ is the set POS with the index j removed

a_j is the positive integer coefficient for x_j

This expression puts an *upper bound* on the possible values of x_j. Thus, it constrains the upper limit of x_j's domain, $\max(x_j)$. Obviously, $\max(x_j)$ cannot be larger than the max value of the RHS, above. This value occurs when we subtract the least, and add the most to b, which happens at the min and max boundaries of the respective variables.

$$\max(x_j) \leq \left\lfloor \frac{b - \sum_{i \in POS \setminus j} a_i \min(x_i) + \sum_{i \in NEG} a_i \max(x_i)}{a_j} \right\rfloor$$

Here $\lfloor y \rfloor$ means the largest integer smaller or equal to y

- If this is larger than $\max(x_j)$, no change
- If this is smaller than $\max(x_j)$, the domain of x_j is adjusted accordingly

Pick a variable x_j with $j \in NEG$

Then we can write for the linear inequality constraint:

$$-x_j \leq \frac{b - \sum_{i \in POS} a_i x_i + \sum_{i \in NEG \setminus j} a_i x_i}{a_j}$$

where $NEG \setminus j$ is the index set NEG with j removed

Multiply by -1 to get rid of the minus sign for x_j

$$x_j \geq \frac{-b + \sum_{i \in POS} a_i x_i - \sum_{i \in NEG \setminus j} a_i x_i}{a_j}$$

This expression puts a *lower bound* on the possible values of x_j. Thus, it constrains the lower limit of x_j's domain, $\min(x_j)$. Obviously, $\min(x_j)$ cannot be smaller than the min value of the RHS, above. This value occurs when we add the least, and subtract the most from $-b$, which happens at the min and max boundaries of the respective variables.

$$\min(x_j) \geq \left\lceil \frac{-b + \sum_{i \in POS} a_i \min(x_i) - \sum_{i \in NEG \setminus j} a_i \max(x_i)}{a_j} \right\rceil$$

Here $\lceil y \rceil$ means the smallest integer larger or equal to y

- If this is smaller than $\min(x_j)$, no change
- If this is larger than $\min(x_j)$, the domain of x_j is adjusted accordingly

Notes on linear inequality constraints domain reduction:

- The reduction needs to be *iterated* until they "stabilize" (a fix-point is reached)
- The calculated domain bounds may or may not result in domain reduction
  If the calculated bounds are less constraining than the original bounds, then no reduction.

Linear equality constraint **(Apt, 6.4.2)**

$$\sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i = b$$

This can be rewritten as two linear inequality constraints that must hold simultaneously

$$\sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \leq b$$

and

$$\sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \geq b$$

Multiply by -1 to get it to $\leq$ form

$$\sum_{i \in NEG} a_i x_i - \sum_{i \in POS} a_i x_i \leq -b$$

Now we have two linear inequality constraints and we can proceed as earlier.

Derivation of it all, p 194-197

$$\sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i = b$$

For $j \in POS$

$$x_j = \frac{b - \sum_{i \in POS \setminus j} a_i x_i + \sum_{i \in NEG} a_i x_i}{a_j}$$

$$\max x_j = \left\lfloor \frac{b - \sum_{i \in POS \setminus j} a_i \min x_i + \sum_{i \in NEG} a_i \max x_i}{a_j} \right\rfloor \qquad (\alpha_j)$$

$$\min x_j = \left\lceil \frac{b - \sum_{i \in POS \setminus j} a_i \max x_i + \sum_{i \in NEG} a_i \min x_i}{a_j} \right\rceil \qquad (\gamma_j)$$

For $j \in NEG$

$$-x_j = \frac{b - \sum_{i \in POS} a_i x_i + \sum_{i \in NEG \setminus j} a_i x_i}{a_j}$$

$$x_j = \frac{-b + \sum_{i \in POS} a_i x_i - \sum_{i \in NEG \setminus j} a_i x_i}{a_j}$$

$$\max x_j = \left\lfloor \frac{-b + \sum_{i \in POS} a_i \max x_i - \sum_{i \in NEG \setminus j} a_i \min x_i}{a_j} \right\rfloor \qquad (\gamma_j)$$

$$\min x_j = \left\lceil \frac{-b + \sum_{i \in POS} a_i \min x_i - \sum_{i \in NEG \setminus j} a_i \max x_i}{a_j} \right\rceil \qquad (\beta_j)$$

$\lfloor y \rfloor$ is the largest integer smaller than y
$\lceil y \rceil$ is the smallest integer larger than y

Derivation of γ_j and δ_i, p.197

$$\sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \geq b$$

For $j \in POS$

$$x_j \geq \frac{b - \sum_{i \in POS \setminus j} a_i x_i + \sum_{i \in NEG} a_i x_i}{a_j}$$

$$\min x_j \geq \left\lceil \frac{b - \sum_{i \in POS \setminus j} a_i \max x_i + \sum_{i \in NEG} a_i \min x_i}{a_j} \right\rceil \qquad (\gamma_j)$$

For $j \in NEG$

$$-x_j \geq \frac{b - \sum_{i \in POS} a_i x_i + \sum_{i \in NEG \setminus j} a_i x_i}{a_j}$$

$$x_j \leq \frac{-b + \sum_{i \in POS} a_i x_i - \sum_{i \in NEG \setminus j} a_i x_i}{a_j}$$

$$\max x_j \leq \left\lfloor \frac{-b + \sum_{i \in POS} a_i \max x_i - \sum_{i \in NEG \setminus j} a_i \min x_i}{a_j} \right\rfloor \qquad (\delta_j)$$

SEND + MORE = MONEY, domain reduction (Apt p.201)

Normalized form

$$\underbrace{(9000\,M + 900\,O + 90\,N + Y)}_{POS} - \underbrace{(1000\,S + 91\,E + 10\,R + D)}_{NEG} = 0$$

$$M, S \in \{1..9\} \qquad O, N, Y, E, R, D \in \{0..9\}$$

Study for now only M, S, and O (since the others do not change):

$$\max M = \left\lfloor \frac{-(900 \min O + 90 \min N + \min Y) + (91 \max E + \max D + 1000 \max S + 10 \max R)}{9000} \right\rfloor = 1$$
$$\min M = \left\lceil \frac{-(900 \max O + 90 \max N + \max Y) + (91 \min E + \min D + 1000 \min S + 10 \min R)}{9000} \right\rceil = 1$$

$$\max O = \left\lfloor \frac{-(9000 \min M + 90 \min N + \min Y) + (91 \max E + \max D + 1000 \max S + 10 \max R)}{900} \right\rfloor = 1$$
$$\min O = \left\lceil \frac{-(9000 \max M + 90 \max N + \max Y) + (91 \min E + \min D + 1000 \min S + 10 \min R)}{900} \right\rceil = 0$$

Note that the expression inside $\lceil\rceil$ above gives a value lower than the original $\min O$, which is 0.

$$\max S = \left\lfloor \frac{(9000 \max M + 900 \max O + 90 \max N + \max Y) - (91 \max E + \max D + 10 \max R)}{1000} \right\rfloor = 9$$
$$\min S = \left\lceil \frac{(9000 \min M + 900 \min O + 90 \min N + \min Y) - (91 \max E + \max D + 10 \max R)}{1000} \right\rceil = 9$$

All of the variables should have different values, and since $M = 1$, O cannot be 1, so $O = 0$. Assigning those values to M, S, and O gives:

$$(90\,N + Y) - (91\,E + 10\,R + D) = 0.$$

We also know now that the domains of these variables are:

$$N, Y, E, R, D \in \{2..8\}.$$

Iterating the linear equality procedure until a fix-point is reached, we can further reduce the domains to:

$$E \in \{4..7\},\ N \in \{5..8\},\ Y, R, D \in \{2..8\}.$$

How to handle disequalities (Apt 6.4.3)

$$\langle S \neq T; D \rangle$$

- S and T are not single variables
- D is the set of domains for the variables in S and T

Introduce now a new variable x, and rewrite as two constraints

$$\begin{array}{ll} x \neq T & D \\ x = S & x \in \{\min(S) \,..\, \max(S)\} \end{array}$$

Why would this help?

Apply it now again on $x \neq T$, we end up with

$$\begin{array}{ll} x \neq y & x \in \{\min(S) \,..\, \max(S)\} \\ x = S & y \in \{\min(T) \,..\, \max(T)\} \\ y = T & D \end{array}$$

Now we have a simple disequality $x \neq y$, which we might be able to process further

Note on $x = S$, S^+, S^- **(Apt p.199)**

$$
\begin{aligned}
S &:= \sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \\
x &= S \quad \text{gives then that} \\
\max(x) &= \sum_{i \in POS} a_i \max(x_i) - \sum_{i \in NEG} a_i \min(x_i) = S^+ \\
\min(x) &= \sum_{i \in POS} a_i \min(x_i) - \sum_{i \in NEG} a_i \max(x_i) = S^-
\end{aligned}
$$

It can never hold that this gives

$$\max(x) < \min(x)$$

Because then

$$\sum_{i \in POS} a_i \max(x_i) - \sum_{i \in NEG} a_i \min(x_i) < \sum_{i \in POS} a_i \min(x_i) - \sum_{i \in NEG} a_i \max(x_i)$$

which is eq to

$$\sum_{i \in POS} a_i \max(x_i) - \sum_{i \in POS} a_i \min(x_i) < \sum_{i \in NEG} a_i \min(x_i) - \sum_{i \in NEG} a_i \max(x_i)$$

eq to

$$
\begin{aligned}
\sum_{i \in POS} a_i(\max(x_i) - \min(x_i)) < \sum_{i \in NEG} a_i(\min(x_i) - \max(x_i)) \\
\sum_{i \in POS} a_i(\max(x_i) - \min(x_i)) \geq 0 \\
\sum_{i \in NEG} a_i(\min(x_i) - \max(x_i)) \leq 0
\end{aligned}
$$

Boolean CSP **(Apt, Section 6.3)**

Variables over boolean domains, $x \in \{0, 1\}$

Boolean connectives,

$\neg$ negation, "not"
$\wedge$ conjunction, "and"
$\vee$ disjunction, "or"

Equality relation, $=$

Simple Boolean constraints (x, y, z variables)

- $x = y$ equality constraint
- $\neg x = y$ NOT constraint
- $x \wedge y = z$ AND constraint
- $x \vee y = z$ OR constraint

Compound ("non-simple") Boolean constraint can always be transformed into simple constraints, by introducing new variables

Transformation rules for Boolean CSPs **(Apt, Section 6.3.1)**

Let $S := s_1 \vee (s_2 \wedge s_3)$, with s_j variables

Then $\neg S = t$ is transformed into $\neg x = t$, $S = x$

Example:

$$\frac{\neg(s_1 \vee (s_2 \wedge s_3)) = t}{\neg x = t,\ s_1 \vee (s_2 \wedge s_3) = x}$$

Apply rule for binary connectives (bottom of p.185)

$$\frac{s_1 \vee (s_2 \wedge s_3) = x}{s_1 \vee y = x,\ s_2 \wedge s_3 = y}$$

Now we have only simple constraints

$$\begin{aligned} &\neg x = t \\ &s_1 \vee y = x \\ &s_2 \wedge s_3 = y \end{aligned}$$

Domain reduction rules for Boolean CSPs

We can define *lots* of domain reduction rules for simple Boolean constraints, see Apt Section 6.3.2, Table 6.1

The rules are all derived for basic properties of the Boolean connectives

For instance, for AND6 we have

$$x \wedge y = z,\ z = 1$$

from which we can conclude that

$$x = 1 \text{ and } y = 1$$

Note: There is no rule for

$$x \wedge y = z,\ z = 0$$

as no definitive conclusion about x and/or y can be drawn from this (likewise for $x \vee y = z,\ z = 1$)

Note small thing that might slip by:

$$y \in \emptyset$$

means the domain of y is empty, hence the CSP fails

Full Adder Circuit

(Apt Sec 6.3.3)

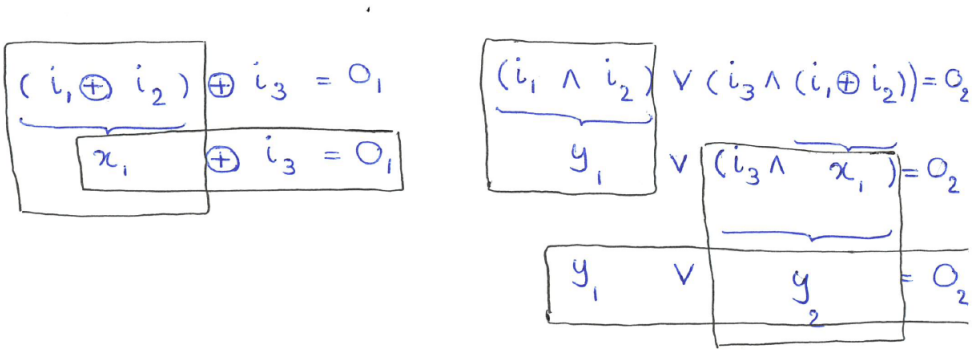


Figure 13.1: Full adder circuit model.

We study what happens if we know $i_3 = 0, O_2 = 1$

These are then domain expressions for i_3 and O_2, the other variables so far have $\{0, 1\}$ as domains

$i_1 \bigoplus i_2 = x_1$ XOR1 $\Rightarrow x_1 = 0$
$i_1 \wedge i_2 = y_1$ AND6 $\Rightarrow i_1 = i_2 = 1$
$x_1 \bigoplus i_3 = O_1$ XOR2 $\Rightarrow O_1 = 0$
$i_3 \wedge x_1 = y_2$ AND4 $\Rightarrow y_2 = 0$
$y_1 \vee y_2 = O_2$ OR4 $\Rightarrow y_1 = 1$
$i_3 = 0$, $O_2 = 1$

EEN025: Constraint Programming and Applied Optimization **Fall 2020**

Lecture 14: Constraint Programming III

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

Hand written notes, sorry for that, start on next page.

Constraint propagation, example

$$x + 2y \leq 10 \qquad x \in \{6..10\}$$
$$y + z \geq 5 \qquad y \in \{2..10\}$$
$$z \in \{0..10\}$$

From the first constraint we get

$$\max x = 10 - 2 \min y = 6$$

Thus, $x = 6$ since $\min x = 6$ from the original domain

When $x = 6$, then

$$\max y = (10 - 6)/2 = 2$$

Thus, $y \in \{2..2\}$

From the second constraint, we now get

$$\min z = 5 - \max y = 3$$

Thus, $z \in \{3..10\}$

In this way, knowledge about x propagates through the constraints to other variables

Constraint propagation

Variables depend on each other by being part of the same constraint(s)

Thus, when one variable is "changed" (is assigned a value, has its domain reduced, etc) the variables that are dependant on this variable may also change

These changes propagate through the constraints

This is called constraint propagation

Types of constraints

* Unary constraint, involves only a single variable, $x \geq 0$

* Binary constraint, relates two variables, $x \neq y$

  A CSP with only binary constraints is said to be binary. An arbitrary CSP can always be converted into an equivalent binary CSP.

* n-ary constraint, relates n variables

  Sometimes called global constraints, but is not necessarily a constraint over the entire set of variables

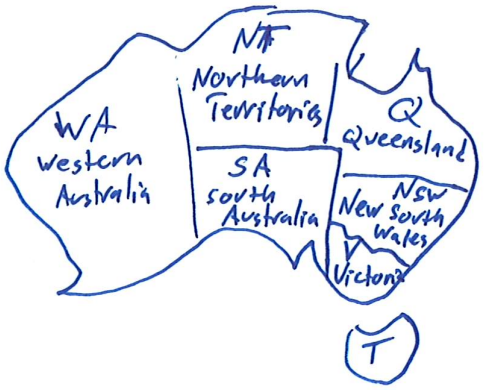
  Example: allDifferent$(x_1, x_2, \ldots x_n)$

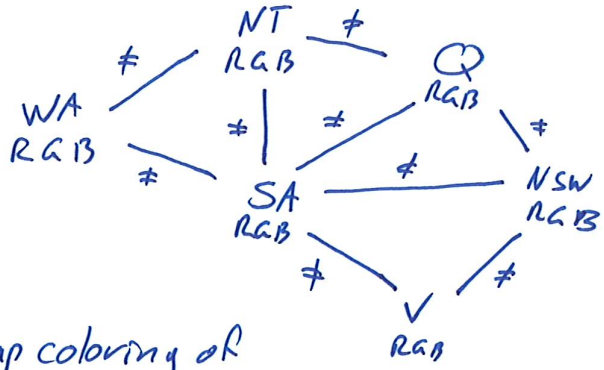
Apt speaks of "arbitrary constraint" instead of "global"

Constraint graph

A constraint graph is a graph with
* the variables associated to the nodes
* the constraints associated to the edges

Straightforward for binary relations





Map coloring of Australian states, given three colors

For non-binary relations, the constraint graph becomes a bi-partite graph, with two different types of nodes, ordinary nodes representing the variables, and hypernodes that represent n-ary constraints.

This is called a constraint hypergraph

CS3600 – Homework 2.5 – Constraint Satisfaction Problems

Recall that constraint satisfaction problems (CSPs), define a set of variables, domains of the possible values for each variable, and a set of constraints, each involving one or more variables, that must be satisfied in a solution to the CSP.

In this homework you will solve a CSP by hand. The problem is a cryptarithmetic puzzle (ever read "Sideways Stories from Wayside School"?). There are a set of variables, each represented by a capital letter, and each variable can be any integer from 0 to 9, inclusive. Each variable must be a different digit. Also, there must be no leading zeros. The variables must satisfy the following base-10 addition problem:

$F, T \in \{1..9\}$

$U, W, R, O \in \{0..9\}$

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

In order to solve this problem, we must define auxiliary variables and constraints representing the carrying digits for the addition:

$$O+O=R+10\cdot X_1$$
$$X_1+W+W=U+10\cdot X_2$$
$$X_2+T+T=O+10\cdot X_3$$
$$X_3=F$$

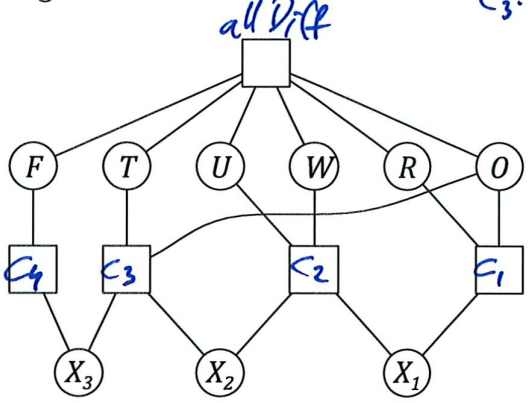
$C_1: X_1 = \frac{1}{10}(O+O-R)$

$C_2: X_2 = \frac{1}{10}(X_1+W+W-U)$

$C_4: X_3 = F$

Your task for this homework is to define and label the constraints in the hypergraph depicting this CSP (reproduced below), and then to solve the CSP by hand. Whenever possible, you should use backtracking, the minimum remaining values (MRV) heuristic, the least-constraining-value heuristic, and forward checking.

$C_3: X_3 = \frac{1}{10}(X_2+T+T-O)$

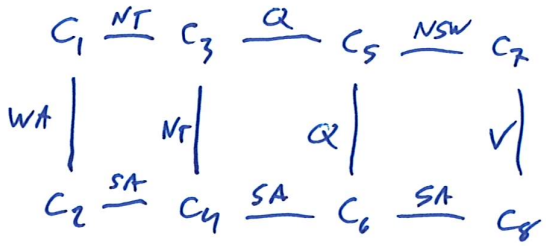


The dual graph

The dual graph has the constraints as the nodes and the edges representing the variable connections

For map coloring of Australia:

C_1: $WA \neq NT$
C_2: $WA \neq SA$
C_3: $NT \neq Q$
C_4: $NT \neq SA$
C_5: $Q \neq NSW$
C_6: $Q \neq SA$
C_7: $NSW \neq V$
C_8: $V \neq SA$



Conversion to binary CSP

The dual graph can be used to convert a non-binary CSP to a binary one

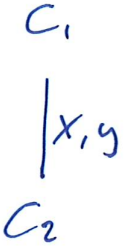
C_1: $x + y = z$

C_2: $x < y$

$x \in \{1, 2\}$

$y \in \{3, 4\}$

$z \in \{5, 6\}$

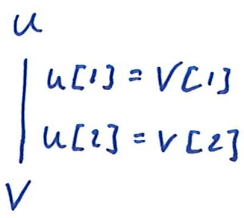


Create new variables, u and v, representing C_1, C_2 resp

$u \in \{\langle 1,4,5\rangle, \langle 2,3,5\rangle, \langle 2,4,6\rangle\}$

$v \in \{\langle 1,3\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 2,4\rangle\}$

Add constraints saying that the first element of u (corresponding to x) must always be the same as the first element of v



The nth element of u and v represent the same variable, and so must always be the same

This is now a binary CSP, with

C_1': $u[1] = v[1]$

C_2': $u[2] = v[2]$

Conversion to binary CSP (ctd)

Another way to convert a non-binary CSP to a binary one is the hidden representation

For each C_i, add a variable h_i which has as its domain a unique identifier for each of the valid tuples of the constraint

$C_1: x+y=z$ $\quad h_1 \in \{1, 2, 3\}$ with $1 \to \langle 1,4,5\rangle$, $2 \to \langle 2,3,5\rangle$ $3 \to \langle 2,4,6\rangle$

$C_2: x<y$

$x \in \{1,2\}$
$y \in \{3,4\}$
$z \in \{5,6\}$

$h_2 \in \{1,2,3,4\}$ with
$1 \to \langle 1,3\rangle$, $2 \to \langle 1,4\rangle$, $3 \to \langle 2,3\rangle$, $4 \to \langle 2,4\rangle$

Define now constraints between h_i and its respective original variables

$C_{h_1 x} = \{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,2\rangle\}$

$C_{h_1 y} = \{\langle 1,4\rangle, \langle 2,3\rangle, \langle 3,4\rangle\}$

$C_{h_1 z} = \{\langle 1,5\rangle, \langle 2,5\rangle, \langle 3,6\rangle\}$

$C_{h_2 x} = \{\langle 1,1\rangle, \langle 2,1\rangle, \langle 3,2\rangle, \langle 4,2\rangle\}$

$C_{h_2 y} = \{\langle 1,3\rangle, \langle 2,4\rangle, \langle 3,3\rangle, \langle 4,4\rangle\}$

This is now a binary CSP

On consistency (Apt ch 5)

A CSP is consistent if it has a solution
(else it is "inconsistent", or "failed")

Remember, a solution is a domain, so that if for a CSP, $D_1 \times D_2 \times \ldots \times D_n$ has

* a single element, a unique solution exists
* no element, no solution exists
* has multiple elements, multiple solutions exist

Assuming that constraints do not conflict, which is what solving the CSP aims to find out

A CSP is node consistent if for every unary constraint over the variable x, that constraint is valid over D_x

(A CSP with no unary constraints is by default node consistent)

$$\frac{\langle C;\ x \in D_x \rangle}{\langle C;\ x \in C \cap D_x \rangle}$$

A CSP is node consistent if it is closed under this rule

Node consistency does not imply consistency

~~$\langle x=0,\ y=0;\ x \in N,\ y \in N \rangle$~~ Globally consistent but not node consistent

Consistency does not imply node consistency

On consistency (ctd)

A CSP is arc consistent if for every binary constraint over the variables x and y, the constraint is valid over the whole of D_x and D_y

(A csp with no binary constraints is by default arc consistent)

$$\frac{\langle C;\ x \in D_x,\ y \in D_y \rangle}{\langle C;\ x \in D'_x,\ y \in D'_y \rangle}$$

A CSP is arc consistent if it is closed under this rule

with $D'_x = \{ a \in D_x \mid \exists b \in D_y \ \langle a, b \rangle \in C \}$

$D'_y = \{ b \in D_y \mid \exists a \in D_x \ \langle a, b \rangle \in C \}$

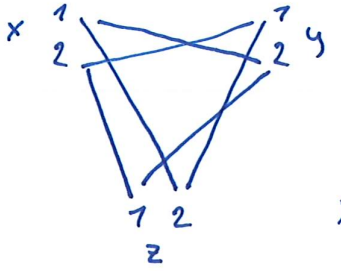
b is called a support of a

a is called a support of b

In general, arc consistency is not related to consistency

But, for certain CSP structures arc consitency does imply consistency. The cross word puzzle (Apt p. 142) was solved simply by repeated application of the AC rule

It is known that enforcing AC establishes satisfiability of CSPs of binary constraints with no cycles



$x \neq y$
$y \neq z$
$z \neq x$

$x, y, z \in \{1, 2\}$

Arc consistent, but no solution exists

Compare coloring Australia with 2 colors

On consistency (ctd)

(Apt, 5.3)

A CSP is hyper-arc consistent if for every constraint C_i for every involved domain each element participates in a solution to C

$$\frac{\langle C_i\ X_1 \in D_1 \ldots X_k \in D_k \rangle}{\langle C_i\ X_1 \in D_1' \ldots X_k \in D_k' \rangle}$$

with $D_i' = \{ a \in D_i \mid \exists d \in C\ a = d[i] \}$

Example

$C: x < y < z$ $\quad \{ \langle 0,1,2 \rangle, \langle 1,2,3 \rangle \}$

$x \in \{0..3\}$
$y \in \{0..3\}$
$z \in \{0..3\}$

For x, 2 and 3 do not participate in a solution to C

For y, 0 and 3 do not participate in a solution to C

For z, 0, and 1 do not participate in a solution to C

Hyper-arc consistency then means:

$x \in \{0,1\}$
$y \in \{1,2\}$
$z \in \{2,3\}$

Viewing this as two binary constraints $x<y$, $y<z$ arc consistency will give the same result

Hyper Arc Consistency example (Apt Ex. 5.8)

C_1: $X \wedge Y = Z$, $X = ?$, $Y, Z \in \{0, 1\}$

X	Y	Z	$X \wedge Y = Z$
1	0	0	T
1	0	1	F
1	1	0	F
1	1	1	T

For X, Y, Z their whole domains participate in combinations that make C_1 true, thus this is hyper-arc consistent

C_2: $X \wedge Y = Z$, $X, Y \in \{0, 1\}$, $Z = 1$

X	Y	Z	$X \wedge Y = Z$
0	0	1	F
0	1	1	F
1	0	1	F
1	1	1	T

For X, Y only one element of their respective domains participate in a combination that make C_2 true, thus this is <u>not</u> hyper-arc consistant

Path consistency (Apt, Def 5.18)

A two-variable set $\{x_i, x_j\}$ is path consistent wrt a third variable x_k, if for every assignment to x_i and x_j consistent with the constraints on $\{x_i, x_j\}$ there is an assignment to x_k that satisfies the constraints on $\{x_i, x_k\}$ and $\{x_k, x_j\}$

We can think of a "path" from x_i to x_j with x_k in the middle

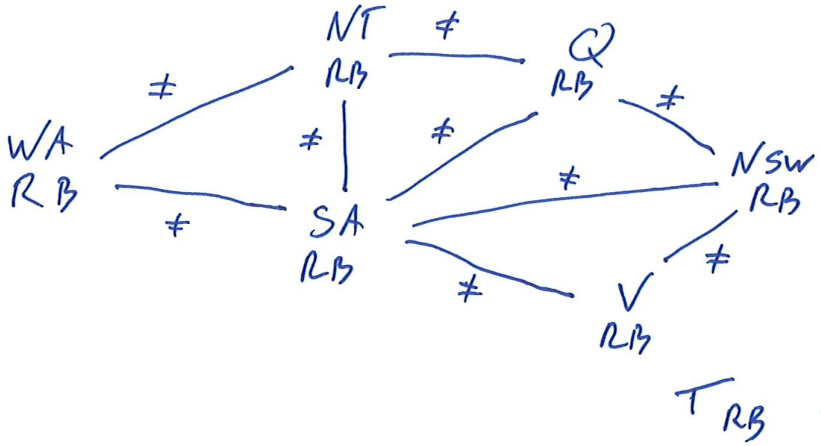
Arc consistency tightens the domains (unary constraints) using the arcs (binary constraints)

Path consistency tightens the binary constraints by using implicit constraints inferred from looking at triples of variables

Generalizing to higher order than triples, what is called m-path consistency, gains us nothing as m-path consistency is equivalent to path consistency (Apt, Theorem 5.23)

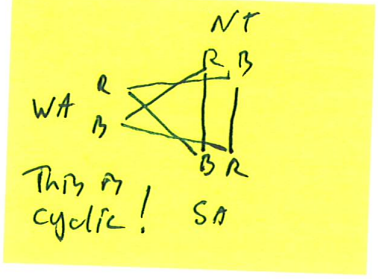
Path consistency example

Consider coloring the Australia map with only two colors, R and B



First note that this is arc consistent, for every binary constraint (as WA ≠ SA) this constraint is valid over the whole of D_{WA} and D_{SA} (as R ≠ B and B ≠ R)

However, if we consider path consistency for say {WA, SA} wrt NT we look at the possible assignments between WA and SA {<R,B>, <B,R>} and then try to find for each of these valid assignments to NT. Of course, no such assignment can be found!

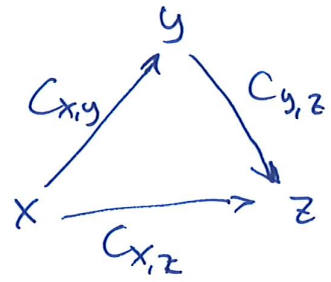


Variant of example 5.20 (Apt p. 151)

$C_{x,y}$: $x < y$ $\quad x \in [0..2]$

$C_{y,z}$: $y < z$ $\quad y \in [1..3]$

$C_{x,z}$: $x < z$ $\quad z \in [4..5]$



Examine $\langle x, z \rangle$ wrt y

$C_{x,z}$	$C_{x,y}$	$C_{y,z}$
0, 4	0, 1	1, 4
1, 4	1, 2	2, 4
2, 4	2, 3	3, 4
0, 5	0, 3	3, 5
1, 5	1, 3	3, 4
2, 5	2, 3	3, 4

All satisfying combinations

Some satisfying combination

Note that the value of y is the same in both columns

Now consider instead $z \in [3..5]$

$C_{x,z}$	$C_{x,y}$	$C_{y,z}$
0,3	0,1	1,3
1,3	1,2	2,3
2,3	2, *	,3
0,4	0,1	1,4
1,4	1,2	2,4
2,4	2,3	3,4
0,5	0,1	1,5
1,5	1,2	2,5
2,5	2,3	3,5

← No value of $D_y = [1..3]$ exists that satisfies both $C_{x,y}$ and $C_{y,z}$ for the given values of x, z.

Here, only the single value $3 \in D_y$ satisfies both $C_{x,y}$ and $C_{y,z}$ for the given values of x, z

This is NOT path consistent

k-consistency

A CSP is k-consistent if for any k-1 variables and any consistent assignment to those variables, a consistent value can be assigned to any k-th variable

1-consistency is the same as node consistency

2-consistency is the same as arc consistency

3-consistency is for binary CSP same as path consistency

Unfortunately, k-consistency even with k large in relation to the total number of variables, in isolation does not say anything about global consistency

However, an accumulated effect of the k-consistency does indeed relate to global consistency

Strong k-consistency (Apt, section 5.8)

A CSP is strongly k-consistent if it is

k-consistent, and
(k-1)-consistent, and
(k-2)-consistent, and
⋮
1-consistent

Theorem (Consistency 1): (Apt, Theorem 5.38)

A CSP with $k \geq 1$ number of variables, s.t.
* at least one domain is non-empty, and
* it is strongly k-consistent,
is globally consistent.

Proof idea:

Choose a consistent value for x_1. Since the CSP is 2-consistent, we are now able to choose a consistent value for x_2. Since the CSP is 3-consistent, we can then choose a consistent value for x_3. And so on...

For each variable x_i, we need only search through the values of D_{x_i} to find a value consistent with $x_1, x_2, \ldots x_{i-1}$

Bounds propagation

For large domains, it is not practical to represent all values in the domain explicitly

Instead, the domain is represented by its bounds

Thus, it is not is not possible to use the local consistency rules to gradually reduce the domain.

Instead, bounds propagation can be used to make the CSP bounds consistent (Art 6.4.7)

Bounds consistency is what we propagated with the linear constraint for SEND=MORE+MONEY

Example from R&N CSP chapter (2.5)

$F_1 \in [0..165]$ $F_2 \in [0..385]$

$F_1 + F_2 = 420$

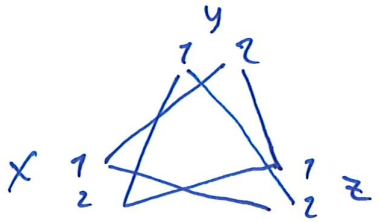
this reduces the domains to

$F_1 \in [35..165]$ $F_2 \in [255..385]$

Last time we looked at increasingly stronger notions of local consistency.

And they are called "local" because they consider subsets of variables/constraints/domains, never the whole global problem.

For example, arc consistency



This is arc consistent, but is not global consistent, because we have a 3-variable all different constraint, but only 2 domain values

We arrived at the nice result that strong k-consistency, with $k = |V|$, implies global consistency.

Unfortunately, this result is rather useless, as the computational cost is too high, exponential in the number of variables (k), also memory requirements are exponential in k, and even worse than the computation time

In practice 2-consistency and maybe 3-consistency is used to try to find solutions for a CSP

Local consistency can only take us so far...

The SEND + MORE = MONEY (sec 6.4.6), after several local consistency transformations we have

$$S=9,\ M=1,\ O=0,\ E\in\{4..7\},\ N\in\{5..8\}$$
$$Y, D, R\in\{2..8\}$$

$$90N + Y - (91E + D + 10R) = 0$$
$$\text{allDifferent}(E, N, D, R, Y)$$

This is closed under the local consistency rules (that we want to use)

We need something more...

In principle, with E, N, D, R, Y the unassigned variables of the SEND+MORE=MONEY example, we could do:

```
foreach e in D_E // each value in the domain of E
{
    assignment.add(E, e)

    foreach n in D_N
    {
        assignment.add(N, n)

        foreach d in D_D
        {
            assignment.add(D, d)

            foreach r in D_R
            {
                assignment.add(R, r)

                foreach y in D_y
                {
                    assignment.add(Y, y)

                    if assignment.isConsistent()
                        return success;    // solution found!

                    assignment.remove(Y, y)
                }
                assignment.remove(R, r)
            }
            assignment.remove(D, d)
        }
        assignment.remove(N, n)
    }
    assignmnet.remove(E, e)
}
return failure;    // unsatisfiable problem
```

In principle!

For the current case we have $4 \times 4 \times 7 \times 7 \times 7 = 5\,488$ combinations to try, in the worst case. For real examples, such an exhaustive enumeration is not tractable.

Real case: 500 variables, avg domain size 6, 50 000 constraints

The number of possible combinations: $6^{500} \approx 10^{390}$

Assume picking out one full assignment and checking it through all 50 000 constraints takes one nanosecond, then checking 1/3 of the total number of combinations will take 10^{375} hours.

The life of the universe is $\sim 10^{12}$ hours

Backtracking Search

Depth-first search that chooses values for one variable at a time, and backtracks when a variable has no legal values left (compare [Apt, 2003], Fig 8.21, p. 331)

```
function BacktrackingSearch(CSP) -- returns a solution, or failure
{
    return Backtrack({}, CSP)
}

function Backtrack(asignment, CSP) -- returns a solution, or failure
{
    if assignment is complete then return assignment

    var := SelectUnassignedVariable(CSP)

    foreach value in OrderDomainValues(var, assignment, CSP)
    {
        assignment.add(var, value)

        if assignment.isConsistent()
        {
            propagates := Propagate(var, value, CSP)

            if propagates != failure
            {
                assignment.add(propagates)
                result := Backtrack(assignment, CSP)

                if result != failure then return result

            }

            assignment.remove(var, value, propagates)
        ]
    }

    return failure
}
```

Why depth-first search? Did we not conclude that breadth-first search is "better"?

BacktrackingSearch does DFS because each "level" corresponds to one variable, and we look for assignments of *all* variables. Using BFS, we would look at partial assignments.

SelectUnasssignedVariable

Simplest strategy is to select the next variable according to some variable ordering. This is the way it is done by [Apt, 2003], Fig. 8.21.

This seldom results in an efficient search!

The *minimum remaining values* MRV strategy is better. It selects the most constrained variable in a "fail first" attempt

Example:

After coloring WA = red and NT = green, SA has only one value in its domain

The *degree heuristic* attempts to reduce the branching factor of future choices by selecting the variable involved in the largest number of constraints with other unassigned variables.

Example:

Initially SA has the highest degree, 5.

(In fact, once SA is assigned, any consistent color can be chosen at each choice point, which results ina solution with no backtracking. Compare tree-structured CSPs discussed later.)

OrderDomainValues

The *least constraining value* heuristic is often a good choice. It prefers the value that rules out the *fewest* choices for the neighboring variables in the constraint graph.

Tries to leave the maximum flexibility for subsequent variable assignments.

Example:

After WA = red, NT = green, if our next variable is Q, blue is a bad choice as it leaves no value for SA.

Note:

Variable selection is "fail first"
Value selection is "fail last"

Choosing variables with minimum remaining values helps minimize the search tree

Choosing values that leaves the most flexibility for the other variables increases the chances of quickly finding a solution

Propagate

There are several useful ways to propagate constraints through the constraint network. We will look at a few of increasing computational burden.

Forward Checking (Apt, 8.4.1, 8.6.1)

Whenever a variable x_i is assigned, the forward checking (FC) process establishes *arc consistency* for it.

For any unassigned variable x_j connected to x_i (by a constraint), delete from D_{x_j} values inconsistent with the value currently chosen for x_i.

Given a specific value for x_i, FC does domain reduction so as to achieve arc consistency for all unassigned variables that are connected to x_i by some constraint.

If some domain then becomes empty, return failure so that the search algorithm can select another value to try.

FC goes well together with the MRV heuristic.

FC works well, but only makes the current variable arc consistent, it does not look ahead to make all the (unassigned) variables arc consistent.

Partial Lookahead (Apt, 8.4.2, 8.6.2)

Whenever a variable x_i is assigned, partial lookahead checks all pairs of variables $\langle x_j, x_k \rangle$ in the "forward" direction, meaning $j = i \,..\, n$, $k = j + 1 \,..\, n$

FC only checks $\langle x_i, x_j \rangle$ with $j = i + 1 \ldots n$

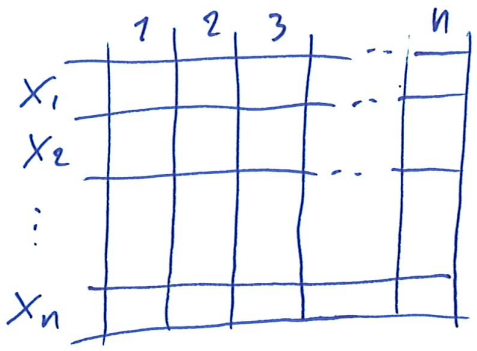
Full Lookahead (Apt, 8.4.3, 8.6.3)

Whenever a variable x_i is assigned, full lookahead checks all pairs of variables $\langle x_j, x_k \rangle$ in both forward and backward direction.

Maintaining Arc Consistency (Alt 8.6.3)

MAC, aka full lookahead, does what FC does, plus recursively propagate constraints when changes are made to the domains

n-Queens formulation



$X_i \in \{1..n\}$

X_i holds the index of the column that the i'th queen is placed in

* Not two queens in the same column

$$X_1 \neq X_2,\ X_1 \neq X_3,\ \ldots,\ X_1 \neq X_n$$
$$X_2 \neq X_3,\ \ldots,\ X_2 \neq X_n$$
$$\ddots \qquad \vdots$$
$$X_{n-1} \neq X_n$$

Can be posed as

$$\text{allDifferent}(X_1, X_2, \ldots X_n)$$

* Not two queens on the same left diagonal

$$X_2 \neq X_1 - 1 \quad X_3 \neq X_1 - 2 \quad \ldots \quad X_n \neq X_1 - (n-1)$$
$$X_3 \neq X_2 - 1 \quad \ldots \quad X_n \neq X_2 - (n-2)$$
$$\ddots \qquad \vdots$$
$$X_n \neq X_{n-1} - 1$$

This is $X_j \neq X_i - (j-i)$ for $1 \leq i < j \leq n$

Add j to both sides of $x_j \neq x_i - (j-i)$

$$x_2+2 \neq x_1+1 \quad x_3+3 \neq x_1+1 \quad \dots \quad x_n+n \neq x_1+1$$
$$x_3+3 \neq x_2+2 \quad \dots \quad x_n+n \neq x_2+2$$
$$\ddots \qquad \vdots$$
$$x_n+n \neq x_{n-1}+(n-1)$$

And this can now be written/posed as

$$\text{all Different}\,(x_1+1,\ x_2+2,\ \dots,\ x_n+n)$$

* Not two queens on the same right diagonal

$$x_2 \neq x_1+1 \quad x_3 \neq x_1+2 \quad \dots \quad x_n \neq x_1+(n-1)$$
$$x_3 \neq x_2+1 \quad \dots \quad x_n \neq x_2+(n-2)$$
$$\ddots \qquad \vdots$$
$$x_n \neq x_{n-1}+1$$

Subtract j from both sides of $x_j \neq x_i + (j-i)$

$$x_2-2 \neq x_1-1 \quad x_3-3 \neq x_1-1 \quad \dots \quad x_n-n \neq x_1-1$$
$$x_3-3 \neq x_2-2 \quad \dots \quad x_n-n \neq x_2-2$$
$$\ddots \qquad \vdots$$
$$x_n-n \neq x_{n-1}-(n-1)$$

Thus: $\text{all Different}\,(x_1-1,\ x_2-2,\ \dots,\ x_n-n)$

Right drag, subtract n from both sides of $x_n \neq \ldots$

$$x_2 - 2 \neq x_1 - 1 \quad x_3 - 3 \neq x_1 - 1 \quad x_4 - 4 \neq x_1 - 1 \quad x_5 - 5 \neq x_1 - 1$$
$$x_3 - 3 \neq x_2 - 2 \quad x_4 - 4 \neq x_2 - 2 \quad x_5 - 5 \neq x_2 - 2$$
$$x_4 - 4 \neq x_3 - 3 \quad x_5 - 5 \neq x_3 - 3$$
$$x_5 - 5 \neq x_4 - 4$$

Results in $x_n - n \neq x_m - m$ (for $n \neq m$)

This can now be written as

$$\text{allDifferent}(x_1 - 1, x_2 - 2, \ldots, x_n - n)$$

Left drag add n to both sides of $x_n \neq \ldots$

$$x_2 + 2 \neq x_1 + 1 \quad x_3 + 3 \neq x_1 + 1 \quad x_4 + 4 \neq x_1 + 1 \quad x_5 + 5 \neq x_1 + 1$$
$$x_3 + 3 \neq x_2 + 2 \quad x_4 + 4 \neq x_2 + 2 \quad x_5 + 5 \neq x_2 + 2$$
$$x_4 + 4 \neq x_3 + 3 \quad x_5 + 5 \neq x_3 + 3$$
$$x_5 + 5 \neq x_4 + 4$$

This can now be written as

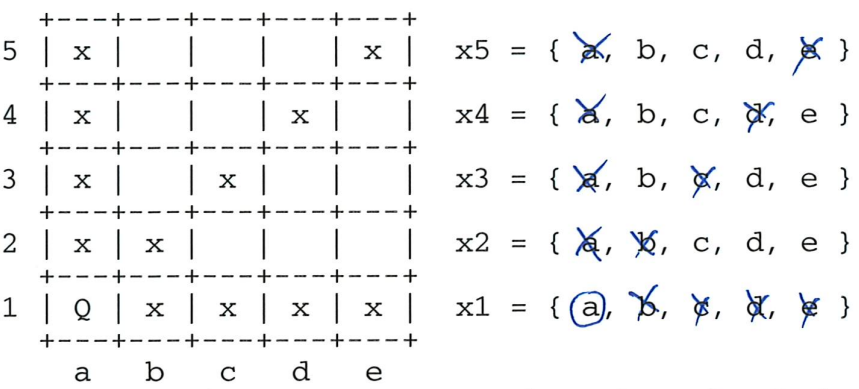
$$\text{allDifferent}(x_1 + 1, x_2 + 2, \ldots, x_n + n)$$

Apt, p. 318-323

Constraints:

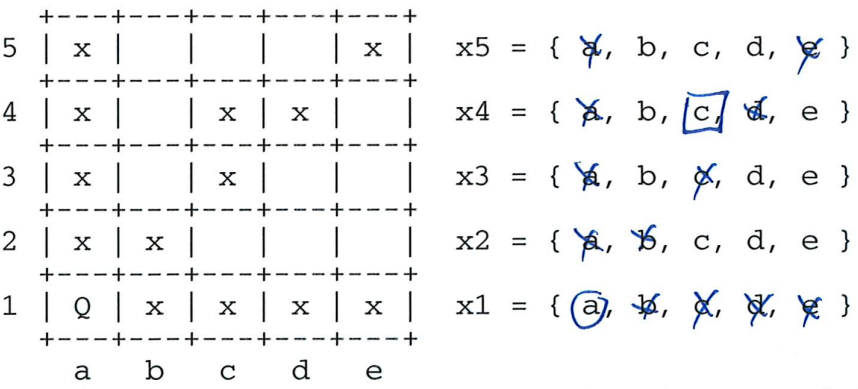
alldifferent(x1, x2, x3, x4, x5)
alldifferent(x1-1, x2-2, x3-3, x4-4, x5-5}
alldifferent(x1+1, x2+2, x3+3, x4+4, x5+5)

Forward checking:



For the assignmnet x1=a, go through each of the other variables and remove from their domain values inconsistent with x1=a. If anyone turns out empty, report it so we can backtrack.

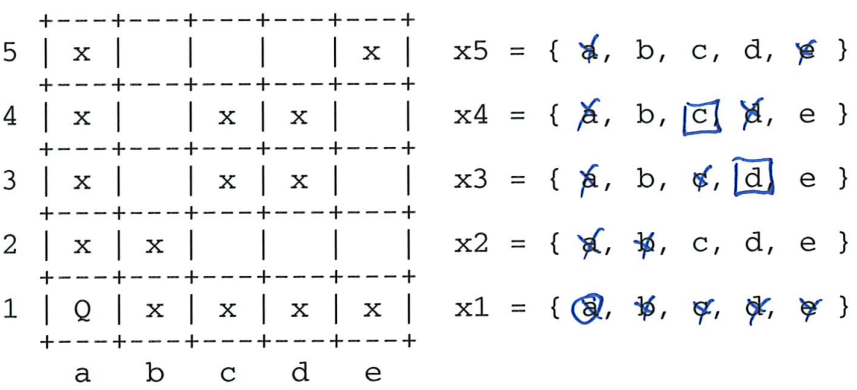
Partial lookahead:



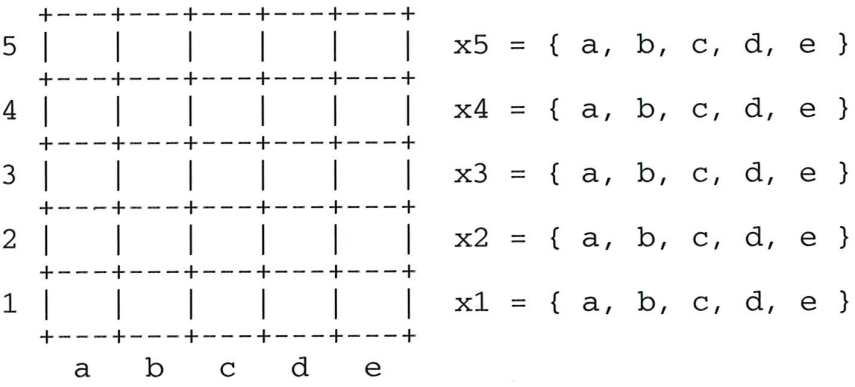
Partial lookahead checks for each pair of variables in the forward direction (x2, x3), (x2, x4), (x2, x5), (x3, x4) etc if some assignment from the domain is in conflict and thus resulting in an empty domain. This finds here the x4=c, which would give x5 an empty domain, and so c must be removed from the domain of x4.

Note that (x_1, x_i) for $i = 2..5$ is what FC checks

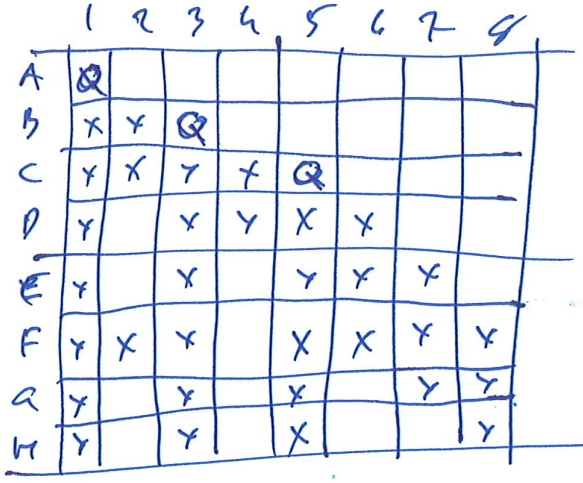
Full lookahead:



Full lookahead looks at all pairs, not only in forward direction, and finds that x3=d will make the domain of x2 empty, and so d must be removed from x3.



8-Queens FC



all Different
$(Q_A, Q_B, \ldots Q_H)$
$(Q_A+1, Q_B+2, \ldots Q_H+8)$
$(Q_A-1, Q_B-2, \ldots Q_H-8)$

$Q_A = 1$
$Q_B = 3$
$Q_C = 5$
} current assignment

$Q_D \in \{2, 7, 8\}$
$Q_E \in \{2, 4, 8\}$
$Q_F \in \{4\}$
$Q_G \in \{2, 4, 6\}$
$Q_H \in \{2, 4, 6, 7\}$

Forward checking for Q_D

What happens if $Q_D = 2$? (affects Q_E, Q_F, Q_G, Q_H)

Then for Q_E, 2 goes away (because of $Q_D \neq Q_E$)

for Q_F, 4 goes away (because of $Q_D - 4 \neq Q_F - 6$)

And then Q_F's domain is empty!

So $Q_D = 2$ is not good!

What happens if $Q_D = 7$? (this effects only Q_E and Q_G and Q_H)

Then for Q_E, 8 goes away (because $Q_D - 4 \neq Q_E - 5$)

for Q_G, 4 goes away (because $Q_D + 4 \neq Q_G + 7$)

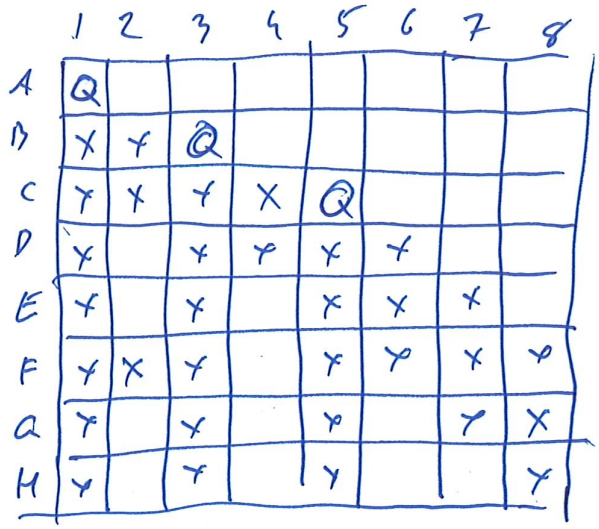
for Q_H, 7 goes away (because $Q_D \neq Q_H$)

What happens if $Q_D = 8$? (affects only Q_E and Q_H)

Then for Q_E, 8 goes away (because $Q_D \neq Q_E$)

Q_H, 4 goes away (because $Q_D + 4 \neq Q_H + 8$)

8-Queens FC+MRV



$Q_A = 1$

$Q_B = 3$

$Q_C = 5$

$Q_D \in \{2, 7, 8\}$

$Q_E \in \{2, 4, 8\}$

$Q_F \in \{4\}$

$Q_G \in \{2, 4, 6\}$

$Q_H \in \{2, 4, 6, 7\}$

Using the MRV heuristic, we examine Q_F instead.

Q_F has only one value in its domain, so either that value works, or the current partial assignment needs to be backtracked

What happens if $Q_F = 4$? (affect all unassigned variables)

For Q_D, 2 goes away (because $Q_F - 6 \neq Q_D - 4$)

Q_E, 4 goes away (because $Q_F \neq Q_E$)

Q_G, 4 goes away

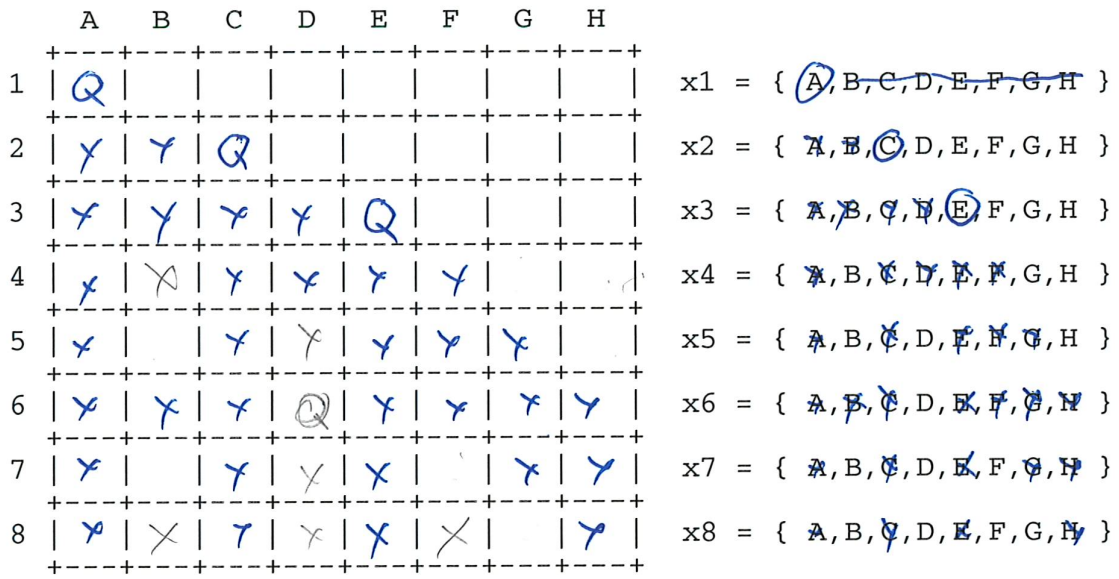
Q_H, 2, 4, 6 goe away

So we put $Q_F = 4$, and then Q_H is the MRV

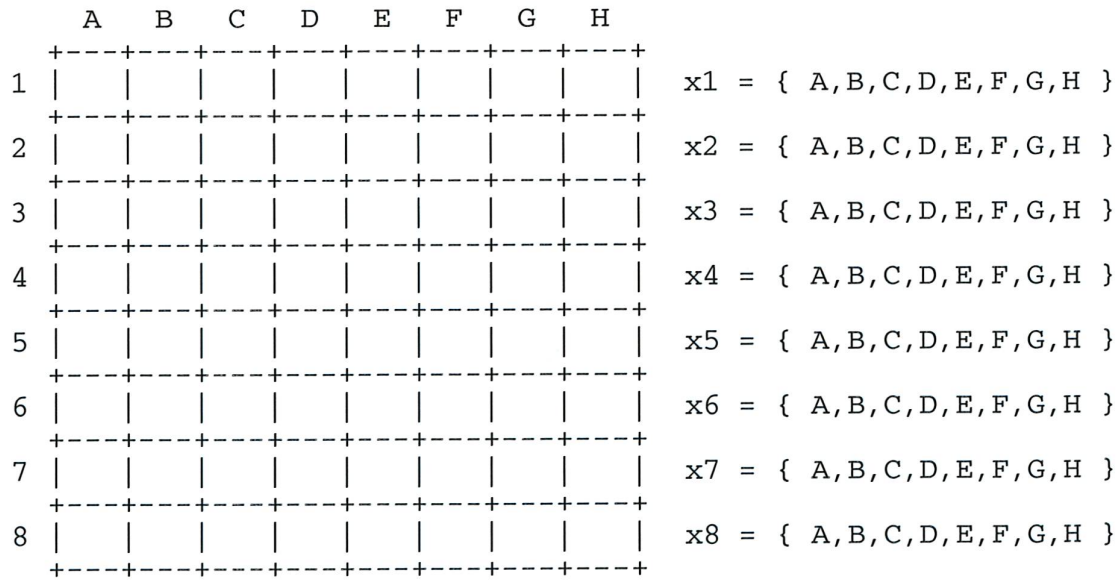
What happens if $Q_H = 7$?

...

```
Constraints: for i = 1..8
    alldifferent(xi, ...)
    alldifferent(xi-i, ...)
    alldifferent(xi+1, ...)
```



Propagating the effects of the assignment $x_6 = D$, full arc consistency finds that it is not a consistent assignment, so we need to backtrack. As x_6 can take no other value, we need to backtrack to check another value for x_3 (if this was the latest assignment)



8-Queens

If the assigment $X_6 = D$ is made:

FC, $X_6 = D$ excludes these

$X_7 = \{ \cancel{A}\ B\ \cancel{C}\ [D]\ \cancel{E}\ F\ \cancel{G}\ \cancel{H} \}$

$X_8 = \{ \cancel{A}\ [B]\ \cancel{C}\ [D]\ \cancel{E}\ [F]\ G\ \cancel{H} \}$

PLA, from what FC has calculated, PLA then checks $\langle X_7, X_8 \rangle$

$X_7 = \{ \cancel{A}\ B\ \cancel{C}\ \cancel{D}\ \cancel{E}\ (F)\ \cancel{G}\ \cancel{H} \}$

$X_8 = \{ \cancel{A}\ \cancel{B}\ \cancel{C}\ \cancel{D}\ \cancel{E}\ \cancel{F}\ [G]\ \cancel{H} \}$

If X_7 is assigned F, then X_8 becomes failed, need to backtrack

Comparison

Forward checking

Checks the effect of a tentative assignment for every variable, one at a time

Partial lookahead

Enforces single step directional arc consistency in the "forward" direction for x_i, x_j with $i \leq j \leq n$ (Needs some idea of ordering)

Full lookahead

Enforces arc consistency between every unassigned pair x_i, x_j (only single step ?)

Arc consitency

Iteratively checks arc consistency for all pairs until stabilizing

Why do propagation during search instead of just simple forward checking and evaluating the tree until solution or inconsistency is found?

* Find inconsistencies early by doing local examination of neighboring variables
* Shrink the domains of the variables so that the search tree is narrowed "underneath" the current partial assignment
* Avoid searching identical subtrees underneath different assignments

Searching for all solutions

It is easy to adapt the CSP search algorithm to find *all* solutions, instead of just one

We need to store any find solution and add it to our constraints in such a way that it is ruled out, so that on the next search we find a different assignment

This we continue, thus strengthening our constraints, until no assignment can be found

See [Apt, 2003], Section 8.6.4 (though note that the algorithm in Fig 8.22, does an exhaustive search and simply prints all found solutions, it does not store found solutions to rule them out on subsequent iterations)

Example:

$$\begin{aligned} x + y &= 4 & D_x = D_y &= \{1, 2, 3\} \\ y &< z & D_z &= \{2, 3\} \end{aligned}$$

First iteration finds $x = 2, y = 2, z = 3$

Add to the constraints the logical expression $\neg(x = 2 \wedge y = 2 \wedge z = 3)$

Next iteration now finds $x = 3, y = 1, z = 3$

Add to the constraints the logical expression $\neg(x = 3 \wedge y = 1 \wedge z = 3)$

next iteration now finds $x = 3, y = 1, z = 2$

Add to the constraints the logical expression $\neg(x = 3 \wedge y = 1 \wedge z = 2)$

Now, no new solution can be found. we are done

Constrained Optimization Problems

We have: (see [Apt, 2003] Chapter 8.7)

- a CSP, $\langle C; x_i \in D_i \rangle$
- an objective function, $obj : D_1 \times D_2 \times \ldots \times D_n \to \mathcal{R}$
- a heuristic function, $h : 2^{D_1} \times 2^{D_2} \times \ldots \times 2^{D_n} \to \mathcal{R}$

The objective function assigns a value to each solution. The objective is to maximize (wlog, as we know)

The heuristic is easier to compute than the objective function, and can be computed for incomplete assignments

The heuristic function provides an *upper bound* on the objective value for *all* solutions that extend the incomplete assignment

Bound: $obj(\langle d_1, d_2, \ldots, d_n \rangle) \leq h(d_1, d_2, \ldots D_i, \ldots, D_n)$

Monotonicity: if $E_1 \subseteq E_2$, then $h(E_1) \leq h(E_2)$

Compare the heuristic of A^*

CSP Model structures

The structure of a CSP has a huge impact on how hard it is to solve

One obvious approach is to divide the CSP into several independent subproblems, each of which can be solved (hopefully) rather quickly

Dividing a Boolean CSP with 80 variables, into four subproblems reduces the worst-case solution time from the lifetime of the universe down to less than a second.

Divide a CSP into $\frac{n}{c}$ subproblems, with c the number of variables (out of the total n) in each subproblem. Each of the subproblems then take at most d^c amounts of work to solve, where d is the size of the domains. The total amount of work is then $d^c \frac{n}{c}$, which is linear in n. Solving the original CSP would instead amount to a total work of d^n, which is exponential in n

Connected components

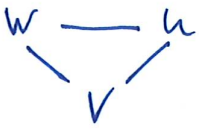
Finding a decomposition into independent subproblems can be done by finding in the constraint graph, connected components

In a graph, a connected component is a "subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph" (wikipedia)

$x < y$

$w \geq u$
$u \geq v$
$w \neq v$

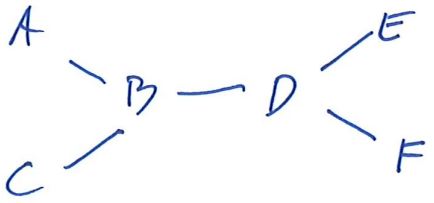
$x - y$



Connected components can be found in linear time (in terms of $\max(|V|, |E|)$) by BFS or DFS

Tree-structured CSPs

A constraint graph is a *tree* if any two variables are connected by only one path



The graph is *acyclic*

A tree-structured constraint graph can be given a *topological ordering*, the variables are ordered such that each variable appears after its parent in the tree

(with alphabetical ordering)

A tree-structured CSP can be solved in time *linear* in the number of variables

Tree-structured CSPs (ctd)

The key to solving tree-structured CSPs in linear time is directional arc consistency

A CSP is directionally arc consistent under an ordering of the variables $x_1, x_2 \ldots x_n$ if every x_i is arc consistent with each x_j, for $1 \leq i < j \leq n$

$X < Y$

$X \in \{2..4\}$
$Y \in \{3..4\}$

Is not arc consistent, since $X=4$ has no support in Y. But for the ordering where Y comes before X, it is directionally arc consistent, since all of Y's possible values have support in X

Any tree with n nodes has $n-1$ arcs, so we can make the graph directionally arc consistent in n steps, each of which must compare up to d domain values for two variables, giving nd^2 amount of work

Once the graph is directionally arc consistent, just assign the variables in order choosing for each any remaining value. No backtracking!

Reducing arbitrary CSPs to tree-structured

There are two ways to make an arbitrary CSP tree-structured:

* cycle cutset
* tree decomposition

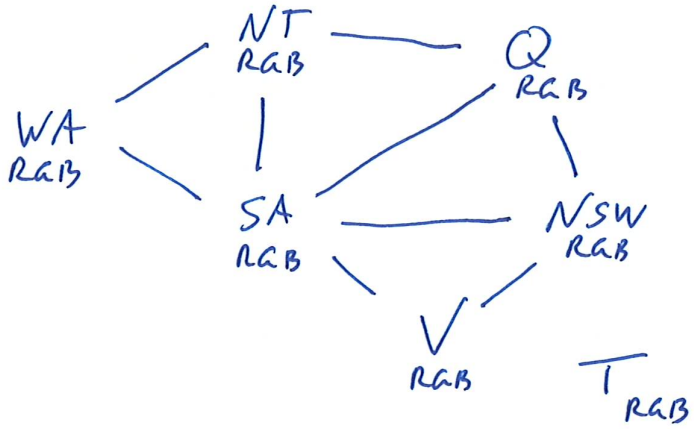
Cycle cutset (aka cutset conditioning)

1. Choose a subset S of the variables so that after removal of S the CSP is a tree
2. For each consistent assignment to S
   a) remove from the variables not in S the values inconsistent with the assignment to S
   b) if the remaining (tree-structured) CSP has a solution, return it together with the assignment to S

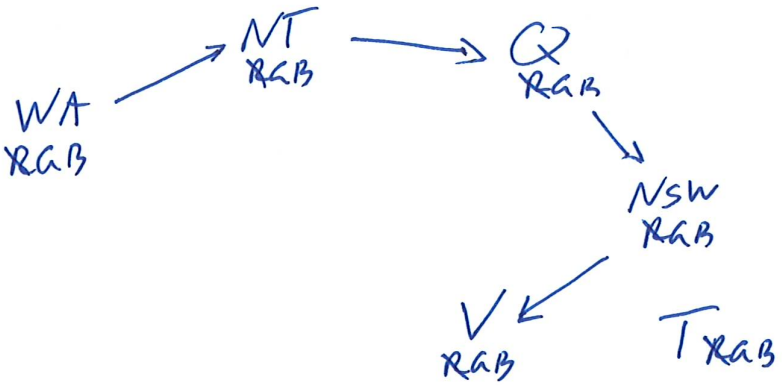
S is called the cycle cutset

Cycle cutset (ctd)

Australia map coloring:

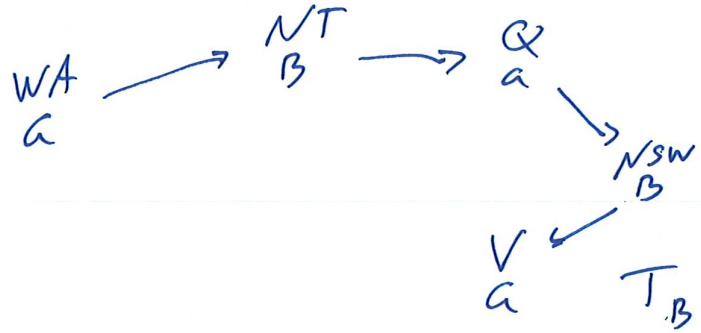


Obviously, SA is a cycle cutset



Select a color for SA, say R

Now select in order WA → NT → Q → NSW → V → T
This is already directionally consistent, so just choose alternating colors



Tree decomposition

Decompose the constraint graph into a set of connected subproblems, that are solved independently with the global solution combined from the independent ones

A tree decomposition must satisfy:

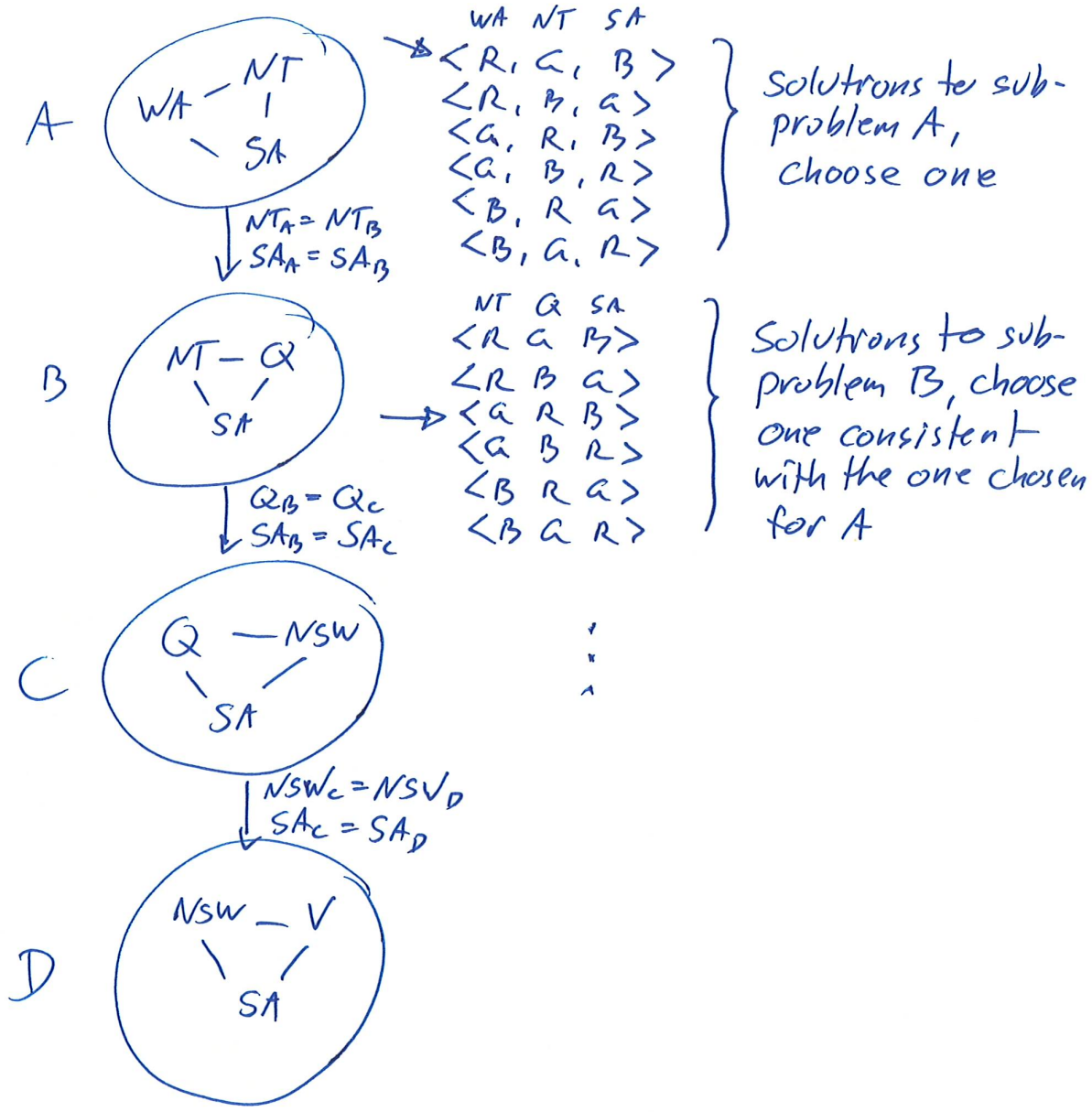
1. Every variable in the original CSP appears in at least one of the subproblems
2. Two variables connected by a constraint must appear together (with the constraint) in at least one of the subproblems
3. If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting the two subproblems

1 and 2 guarantees that all variables and all constraints appear in the decomposition

3 reflects the constraint that a variable must have the same value in every subproblem

Subproblems are solved independently. If a subproblem has no solution, the original problem is inconsistent

Tree decomposition (ctd)



Note: finding the optimal decomposition is NP-hard, heuristic methods that work well in practice do exist

A Solution symmetry is a permutation of the set of <variable, value> pairs that preserves the set of solutions

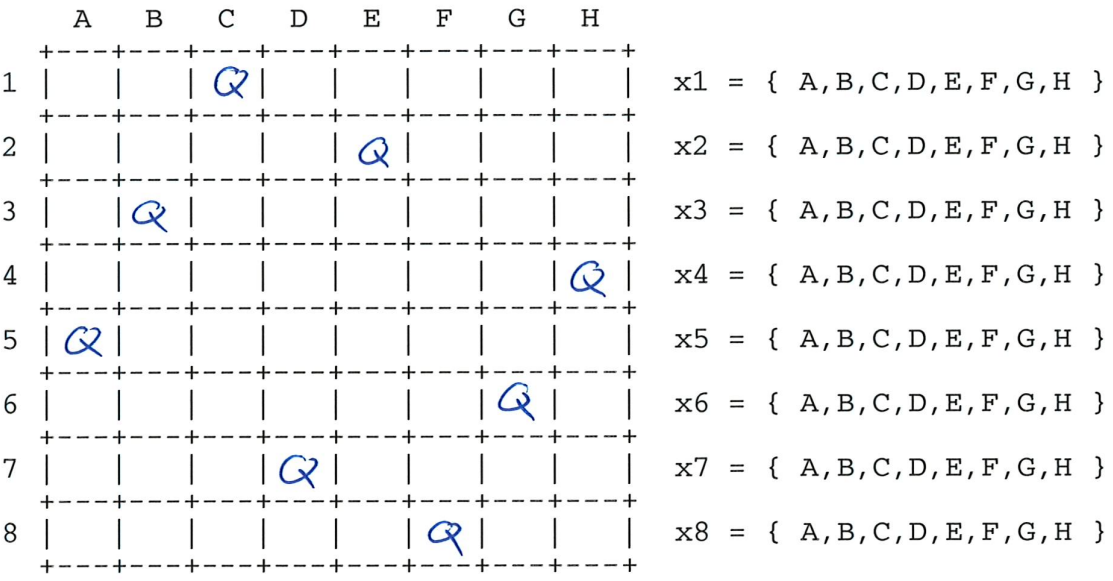
A Problem symmetry is a permutation of the set of <variable, value> pairs that preserves the set of constraints (aka constraint symmetry)

Variable symmetry permutes only the variables

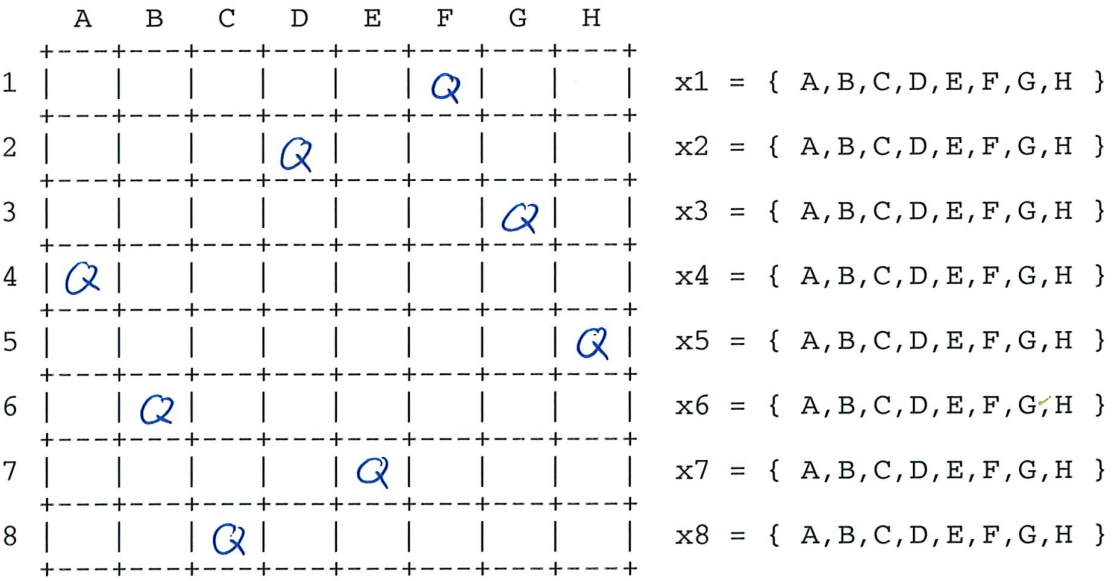
Value Symmetry permutes only the values

? what does it mean to preserve "the set of constraints"?

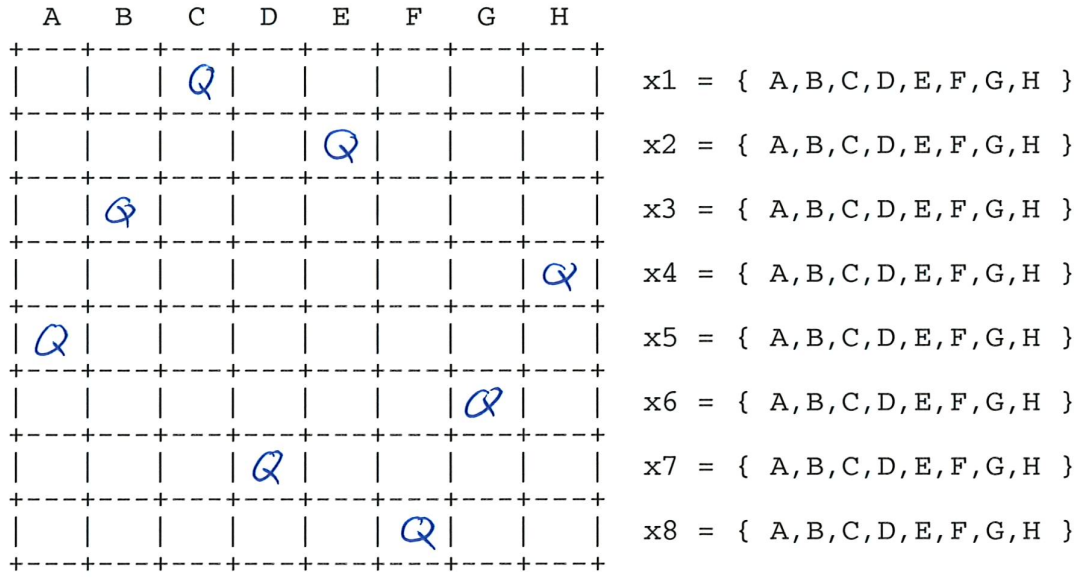
```
Constraints: for i = 1..8
    alldifferent(xi, ...)
    alldifferent(xi-i, ...)
    alldifferent(xi+1, ...)
```



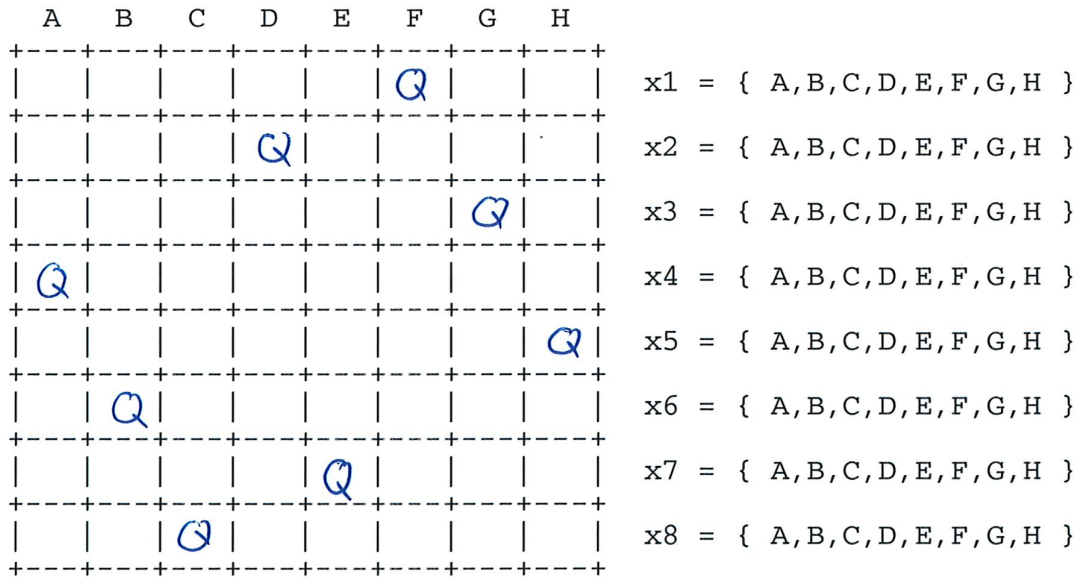
Mirror in the x-axis $r_i = j \rightarrow r_{n-i+1} = j$



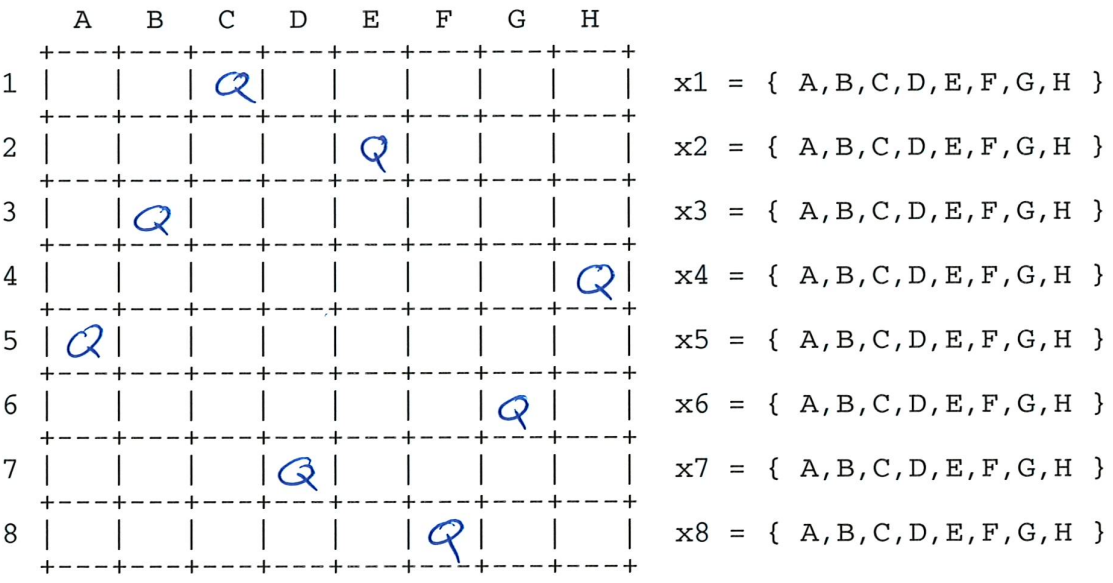
```
Constraints: for i = 1..8
    alldifferent(xi, ...)
    alldifferent(xi-i, ...)
    alldifferent(xi+1, ...)
```



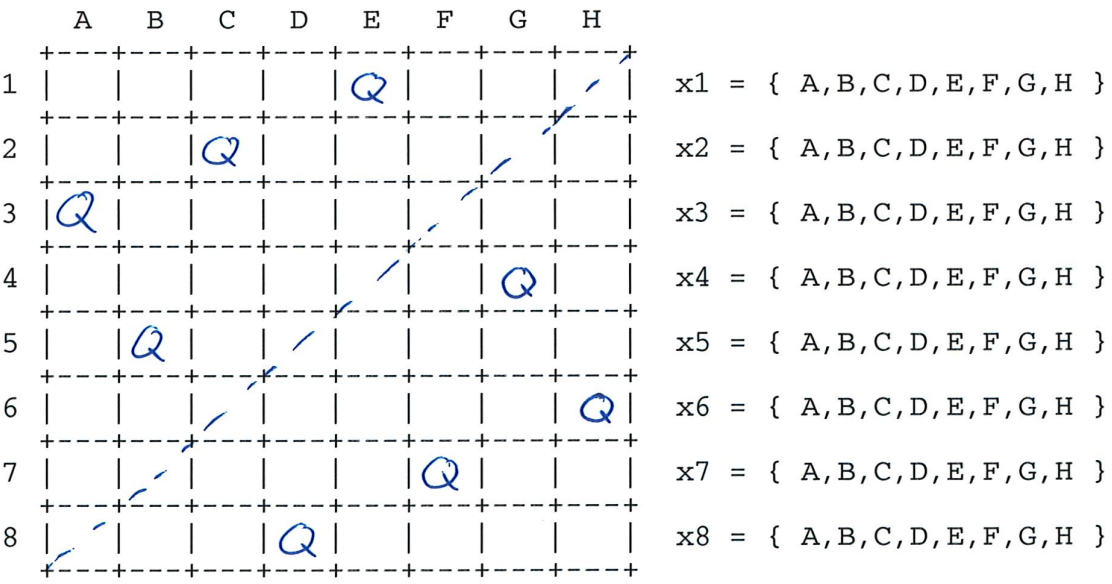
Mirror in the y-axis, $r_i = j \rightarrow r_i = n - j + 1$



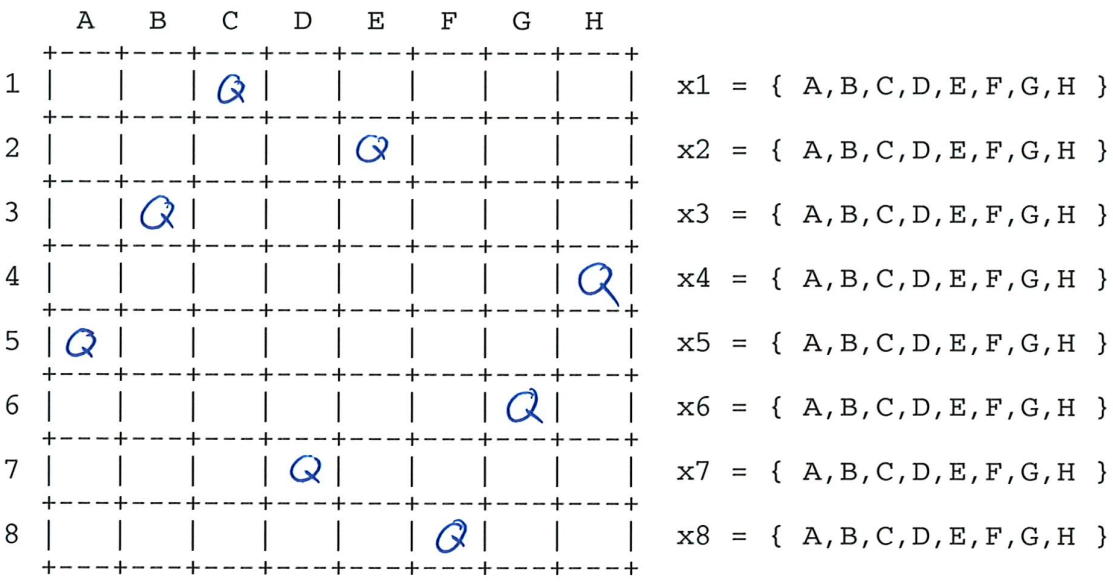
```
Constraints: for i = 1..8
    alldifferent(xi, ...)
    alldifferent(xi-i, ...)
    alldifferent(xi+1, ...)
```



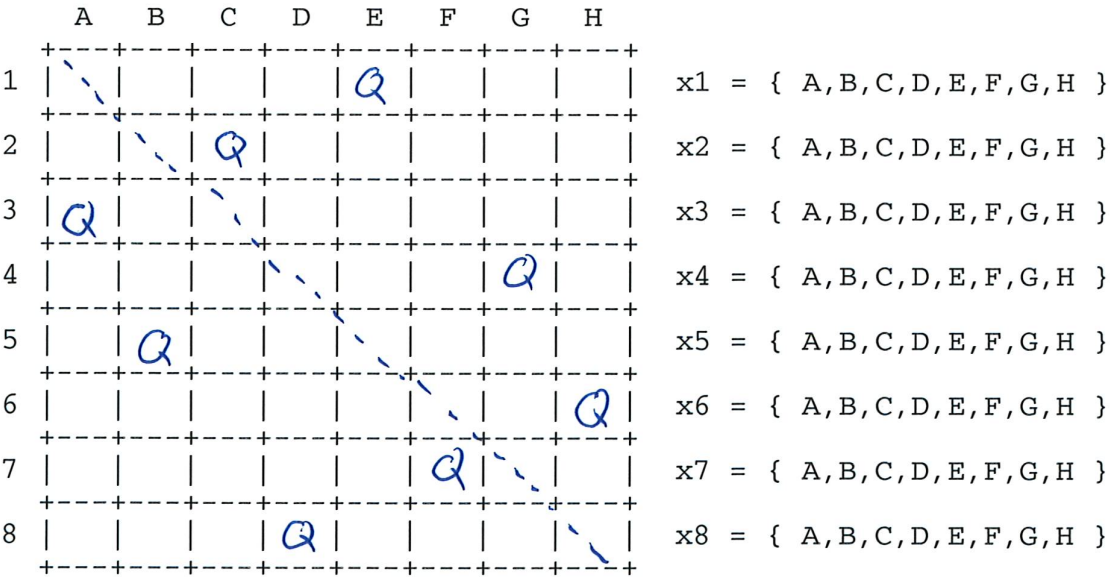
Mirror in the R to L diagonal, $r_i = s \rightarrow r_{n-s+1} = n - i + 1$



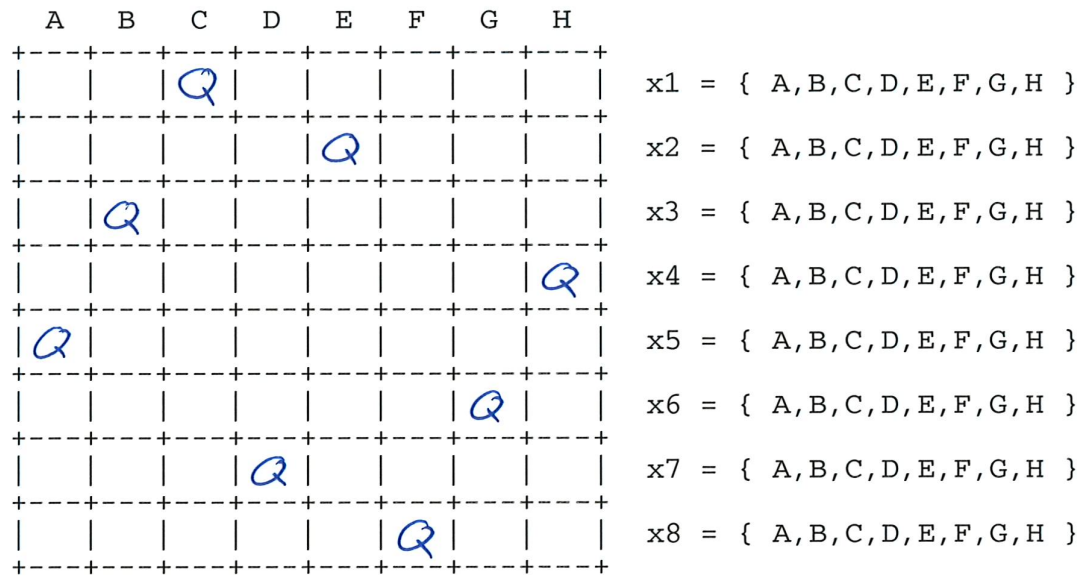
```
Constraints: for i = 1..8
    alldifferent(xi, ...)
    alldifferent(xi-i, ...)
    alldifferent(xi+1, ...)
```



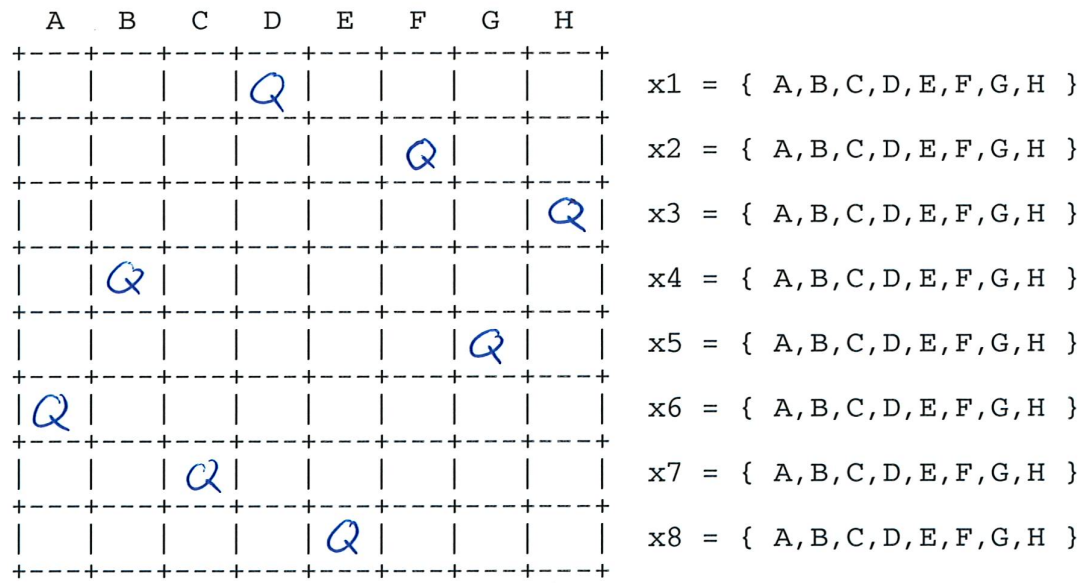
Mirror in L to R diagonal, $r_i = j \rightarrow r_j = i$



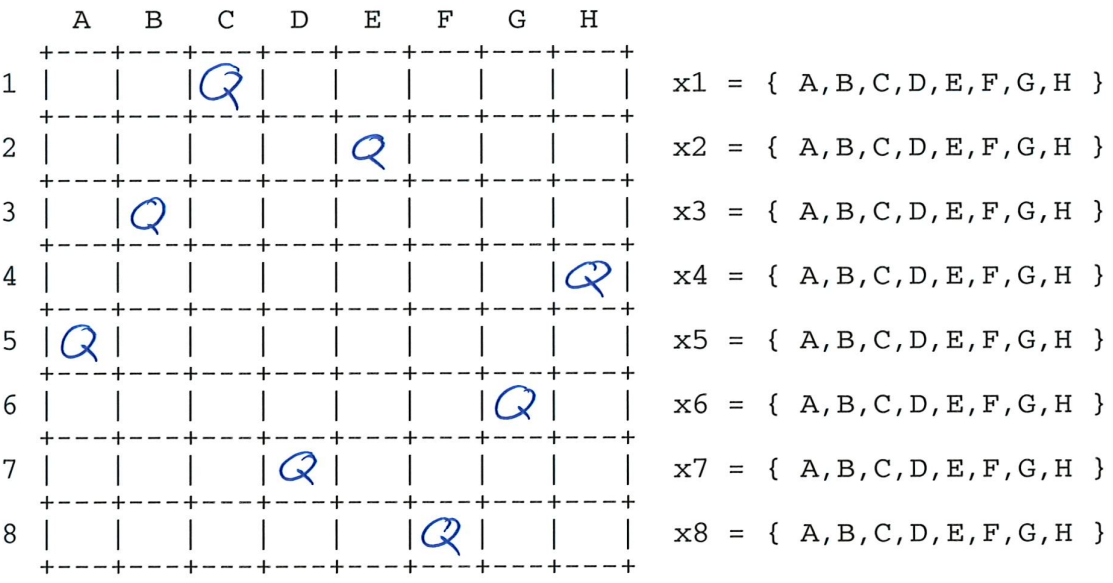
```
Constraints: for i = 1..8
    alldifferent(xi, ...)
    alldifferent(xi-i, ...)
    alldifferent(xi+1, ...)
```



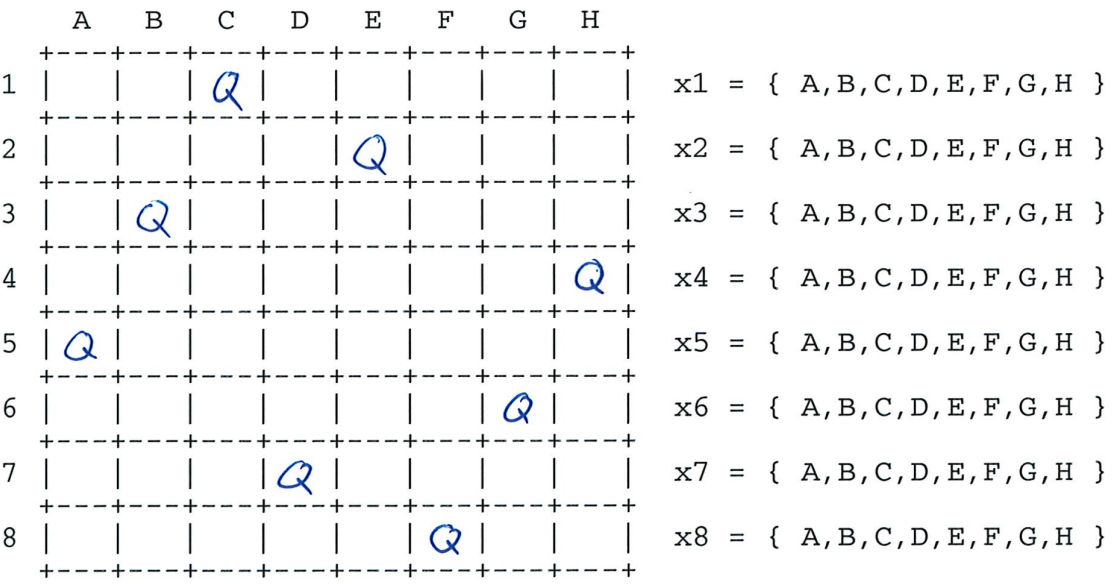
Rotate 90 degrees, $r_i = j \rightarrow r_j = n - i + 1$



```
Constraints: for i = 1..8
    alldifferent(xi, ...)
    alldifferent(xi-i, ...)
    alldifferent(xi+1, ...)
```

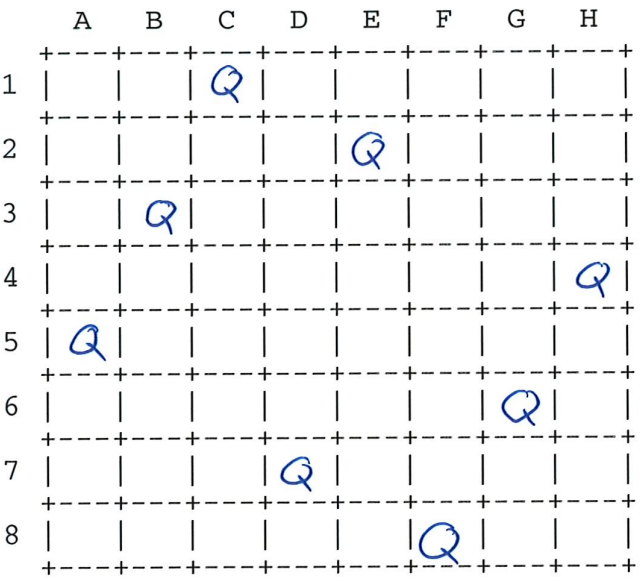


Rotate 180 degrees, $r_i = j \rightarrow r_{n-i+1} = n-j+1$

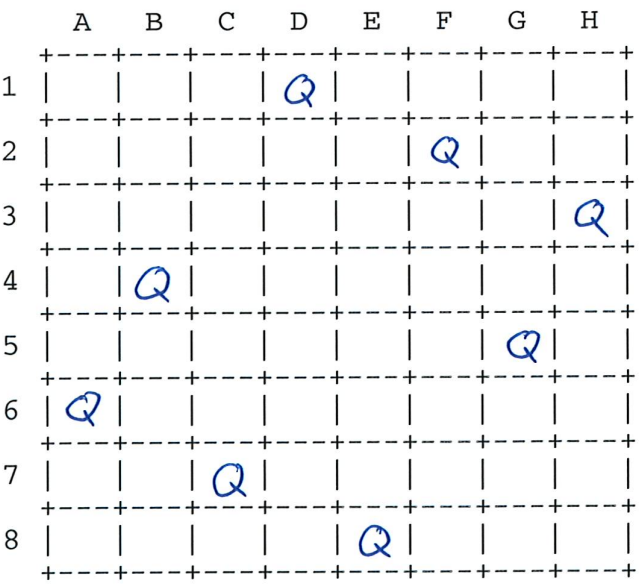


This solution is equivalent to its 180 degree rotation, so only 4 different variants exist

```
Constraints: for i = 1..8
    alldifferent(xi, ...)
    alldifferent(xi-i, ...)
    alldifferent(xi+1, ...)
```



Rotate 270 degrees, $r_i = j \rightarrow r_{n-j+1} = i$



EEN025: Constraint Programming and Applied Optimization **Fall 2020**

Lecture 15: Computational Complexity and Stuff

Lecturer: Martin Fabian *Assistant: Sabino Francesco Roselli*

More handwritten notes follows, sorry.

On Computational Complexity

On theories

CS researchers talk about different theories

A theory in this context is a set of rules that define what we are allowed to express

First-order theories, which is what we are interested in, are defined over

* Variables, X
* Logical connectives, $\wedge, \vee, \neg$, quantifiers $\forall, \exists$
* Non-logical symbols, functions, constants
* Syntax rules, define well formed formulas

In addition, we need to have meaning assigned to the symbols, this is called semantics

We will only look at theories with well known semantics, such as + meaning summation, = meaning equality

Compare Presburger Arithmetic: $\{0, 1, +, =\}$

Presburger Arithmetic

In early 1930's, a guy named Mojźesz Presburger thought to himself something like "what happens if I restrict myself to the logical connectives and only the numbers 0 and 1, and the + operator"?

He was a master student at the time at a Polish university

So he set up 5 axioms that provided the starting point:

1. $\neg(0 = x + 1)$
2. $x + 1 = y + 1 \implies x = y$
3. $x + 0 = x$
4. $x + (y + 1) = (x + y) + 1$
5. $(P(0) \wedge \forall x P(x) \implies P(x + 1)) \implies \forall y P(y)$

The last one is induction, and it really describes an infinite number of axioms.

Note that we can introduce abbreviations for long additions, like $5 = 1 + 1 + 1 + 1 + 1$, and then $5 + 5 = (1 + 1 + 1 + 1 + 1) + (1 + 1 + 1 + 1 + 1)$. So what we are talking about is really a theory of natural numbers (including 0)

We can also introduce "higher-order" operators, like $<$, by defining this as $x < y \iff \exists z : x + z + 1 = y$
Of course, then follows $x \leq y$, and $x > y$, $x \geq y$

Furthermore, we can even allow multiplication with constants, since this is nothing but repeated addition: $5x = x + x + x + x + x$

It looks like we are starting to be able to describe IP-problems. . . and we are!

Assuming we are only concerned with rational coefficients, we can *wlog* assume that our IP-problems have integer coefficients; we can also scale so that *wlog* we only deal with non-negative values, and thus Presburger Arithmetic allows to formulate the transportation problem, for instance.

But the most important thing is that Presburger showed as his Masters thesis work that under these rules, we can never write an expressions the validity of which cannot be algorithimically decided

Put another way, any expression we formulate under the rules of Presburger Arithmetic, we can feed to a computer program and have the expression determined as true or false.

We say that Presburger Arithmetic is *decidable*

Presburger himself ended his research paper with something like "in the area of integer arithmetic restricted to the set A_x there are no more undecidable problems."

Peano Arithmetic

In early 1900's a guy named Giuseppe Peano formulated mathematical logic and set theory, and gave the axioms for an arithmetic over the natural numbers similar to Presburger but including also multiplication

This makes a huge difference!

Suddenly we can write expressions for which we cannot program a computer to decide whether they are true or false

This was a big surprise and a big problem for mathematicians for 20 years or so, before a guy named Kurt Gödel used mathematical logic to show that there are limits to what we can compute

Peano Arithmetic is not decidable!

On decidability

Given a theory, it is decidable if there exists for every well formed formula a procedure that terminates with the answer "true" for a true formula, and "false" for a false formula

"Procedure" means in practice algorithm

When it comes to such procedures we talk of

* soundness, if "true" is returned, then the formula is indeed true, and
* completeness, the procedure terminats and if the formula is true, the the procedure indeed returns "true"

"False" is not mentioned, but for formula f = false, we have ¬f = true

Surprisingly, maybe, not all theories are decidable
For example, Peano arithmetic $\{0, 1, +, *, =\}$

formula valid ⇒ returns "valid"

Soundness: returns ~~"valid" when formula is valid~~ for valid formula "valid"

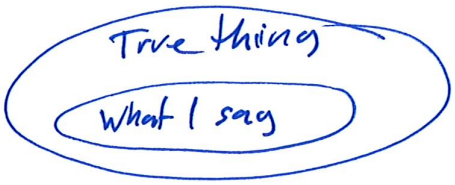
Completeness: terminates and returns "valid" ⇒ formula valid

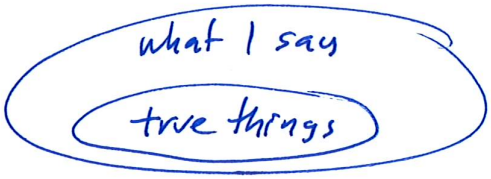
Soundness

If analysis says that X is true, then X is indeed true

Completeness

If X is true, then the analysis says that X is true





Sound and complete:
Say exactly the true things

Is the "complete" in NP-complete related to completeness?

Assignment problems can be regarded as decidability problems in that we can regard each possible combination and decide whether it is a solution or not.

But if we look at assignment problems specifically, we can say the following...

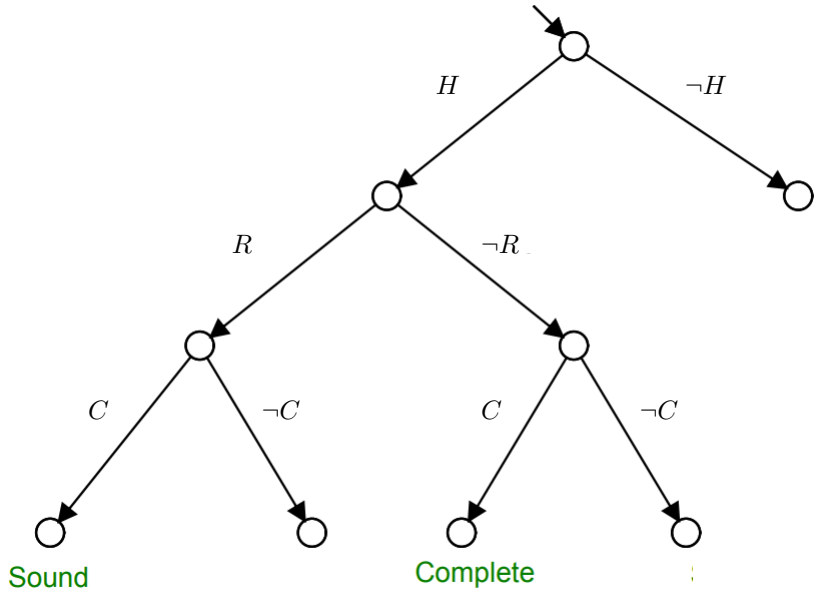
An algorithm that is given an assignment problem can eventually halt (H) or not ($\neg H$)

If it does not halt, then it is useless

If it halts, then it can return a result (R) or a failure ($\neg R$)

If it returns a result, this result can be correct (C) or incorrect ($\neg C$)

But also, if it returns failure this can be correct, no solution exists, or incorrect (one or more solutions do exist, but the algorithm did not manage to find any)



This is why [Apt, 2003] talks of "incomplete" solvers. The various forms of local consistency that we have looked at cannot determine whether a CSP is inconsistent. So the term "incomplete" is used in exactly this sense.

Even so, though these incomplete approaches cannot determine inconsistency, they have still been used to find solutions for large size consistent CSPs. Thus, if they return a solution, that solution is a true solution to the CSP, hence these methods are sound.

On expressibility

Given a theory, its rules determine what we can express, thus different theories have different expressibility

For instance, in the theory of 3-CNF we can write things like

$$x_1 \vee x_2 \vee x_3 \qquad x_i \in \{T, F\}$$

In the theory of 2-CNF we are only allowed to use disjunctions over 2 variables

Thus, we can express things in 3-CNF that we cannot express in 2-CNF

For example:
The 3-CNF expression $x_1 \vee x_2 \vee x_3$ is true for all assigments, except $x_1 = F$, $x_2 = F$, $x_3 = F$

But it is impossible to write a 2-CNF expression that is false only for that particular combination

Thm sq since each 2-CNF clause must be false for 2 of the 8 possible assignments

Thus, 2-CNF is strictly less expressive than 3-CNF

On decidability and expressibility

The more expressive a theory is, the harder its decidability becomes

This is why we need different theories, some are easy to decide, some are hard, some are undecidable

A formula in a theory defines a set of satisfiable assignments, think of CP

One theory T_1 is more expressive than another theory T_2, if

$$\forall f_2 \in T_2 \; \exists f_1 \in T_1 : L(f_2) = L(f_1) \; \wedge$$
$$\exists f_1 \in T_1 \; \forall f_2 \in T_2 : L(f_1) \neq L(f_2)$$

where $L(f_i)$ is the satisfying assignments of formula f_i

In words:

If for all formulas in T_2 there exists some formula in T_1 such that these formulae have the same satisfying assignments, and for some formula in T_1 there is no formula in T_2 with the same satisfying assignments

Example:
2-CNF is less expressive than CNF as every 2-CNF formula is a CNF formula, but not the other way around

2-CNF is less expressive than 3-CNF

We already argued that there is no way to write in 2-CNF an expression equivalent to

$$x_1 \vee x_2 \vee x_3$$

But we should also show that all 2-CNF expressions can be written as 3-CNF.

An example will convince us:

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3)$$

Add an extra variable for each clause, write

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge$$
$$(y_1 \vee \neg y_1) \wedge (y_2 \vee \neg y_2)$$

We can write this as an equivalent 3-CNF

$$(x_1 \vee x_2 \vee y_1) \wedge (x_1 \vee x_2 \vee \neg y_1) \wedge$$
$$(x_2 \vee x_3 \vee y_2) \wedge (x_2 \vee x_3 \vee \neg y_2)$$

Note that we need the same number of variables for comparison of the satisfying assignments to make sense

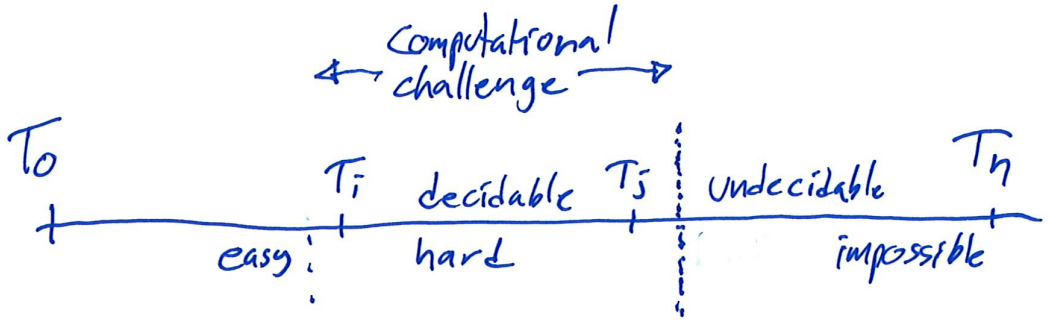
On decidability and expressibility (ctd)

Note that some theories may be incomparable
Some things can be expressed in T_1 that cannot be expressed in T_2, and the other way around.

Assume we have a theory T_n
Restrict T_n in some ways into the theory T_{n-1}
Restrict T_{n-1} in some ways into the theory T_{n-2}
⋮
Restrict T_1 in some ways into the theory T_0

Where "in some ways" means "in such a way that T_i is less expressive than T_{i+1}".

Then we may have the following situation:



Polynomial time solving and verifying

Given a list of integers, n long:

1. Find the max

   We have to walk the list from start to end and look at each integer and compare it to the currently seen largest one

   The amount of work is therefore proportional to the length n of the list, $O(n)$

2. Given value x, verify that this is the max

   Same job as 1, $O(n)$

3. Sort the list

   Now we have to do several rounds through the list. We can look at pairs of integers and put them in correct order. Doing this again and again, until nothing changes, will sort the list. (Bubble sort)

   The amount of work is $O(n^2)$

4. Determine if the list is sorted or not

   This is the same job as 1. and 2.!

   $O(n)$

```
int find_max(List<int> list)     // find the max of a give list
{
    assert(not list.isEmpty()) : "Empty list has no max!";

    int max = list.firstElement():
    foreach(val in list)
    {
        if(val > max)
            max = val;
    }
    return max;
}


bool check_max(List<int> list, int max) // check that a given max is the max of a given list
{
    return max == find_max(list);
}

//////////////////////////////////////

bool sort_list(List<int> list)
{
    bool was_already_sorted = true;

    if(list.isEmpty())  // An empty list is always sorted
        return was_already_sorted;

    int topmost = 0;
    do
    {
        bool swapped = false;
        for(int i = 1; i < list.length()-topmost; i++)
        {
            if(list[i-1] > list[i]) // Never true for a sorted list
            {
                swap(list[i-1], list[i]);
                swapped = true;
                was_already_sorted = false;
            }
        }
        topmost += 1; // on each iteration the topmost sorted
                      // elements increase by one
    } while(swapped);

    return was_already_sorted;
}

bool check_list(List<int> list)
{                               // For a sorted list, sort_list
    return sort_list(list);     // makes only one pass through
                                // its code
}
```

Sorting as an Assignment Problem

Given a list of numbers, $L = \langle 4, 11, 42, 7, 99, 32, 4, \ldots \rangle$ of length n, assign values to n variables x_i (for $i = 1 \ldots n$) so that $x_i \leq x_{i+1}$ (for $i = 1 \ldots n-1$).

Thus, we have an assignment problem $\langle X, C, D \rangle$ with;

$$\begin{aligned} X &= \{x_1, x_2, \ldots, x_n\} \\ C &= \{x_1 \leq x_2,\ x_2 \leq x_3,\ \ldots,\ x_{n-1} \leq x_n\} \\ D_{x_1} &= L \\ D_{x_2} &= L \\ &\vdots \\ D_{x_n} &= L \end{aligned}$$

The domains are the same for all variables, namely the list of given values L.

The constraint graph looks like:

$$x_1 \overset{\leq}{\text{———}} x_2 \overset{\leq}{\text{———}} x_3 \ \cdots\ x_{n-1} \overset{\leq}{\text{———}} x_n$$

This is beneficial, as it means that we never have to backtrack.

We assign x_1 the smallest value in the domain, remove that value and propagate that removal to all domains (in practice we need only a single domain shared between all variables). Then we assign x_2 the smallest value from the remaining domain, etc

The result will be an assignment to all all variables that fulfills all constraints.

How much work?

Assigning x_1, we need to go through all n elements in the list.

Assigning x_2, we need to go through all $n-1$ elements in the list

Assigning x_3, we need to go through all $n-2$ elements in the list

...

Assigning x_{n-1}, we need to go through all 2 elements in the list

Assigning x_n, there is only a single element left in the list

So... the total amount of work is:

$$n + (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n\,(n+1)}{2}$$

Thus, the complexity is $\mathcal{O}(n^2)$

Circuit satisfiability verification and solving

Given a Boolean expression of n variables

1. Verify if a given combination of variable values make the expression true.

   Simply walk the "list" from start to end, assigning the given values to the variables and evaluating the connectives

   This is done in linear time, $O(n)$

2. Find a combination of variable values that make the expression true

   In principle, you have to evaluate <u>all</u> of the 2^n combinations and try them one by one

   In the worst case none makes the expression true, so you may in fact have to test all

   $O(2^n)$

Now consider an optimization problem

An algorithm returns what it claims is the optimal solution

In the worst case it has to evaluate *all possible* solutions to compute this, which we know to be non-polynomial

Now we are to check that the returned result is indeed the optimal solution

To do that we have to evaluate *all possible* solutions, otherwise there is no way we can be sure

This is again non-polynomial!

So in this case, bot computing and checking is non-polynomial.

This is not good news...

Solving and verifying

Some computational problems can be solved in polynomial time

These same problems can also be verified in polynomial time

Some problems cannot be solved in polynomial time, but takes much longer, say exp

↑

Some of these problems can be verified in polynomial time though.

The set of problems that can be verified in polynomial time is called NP

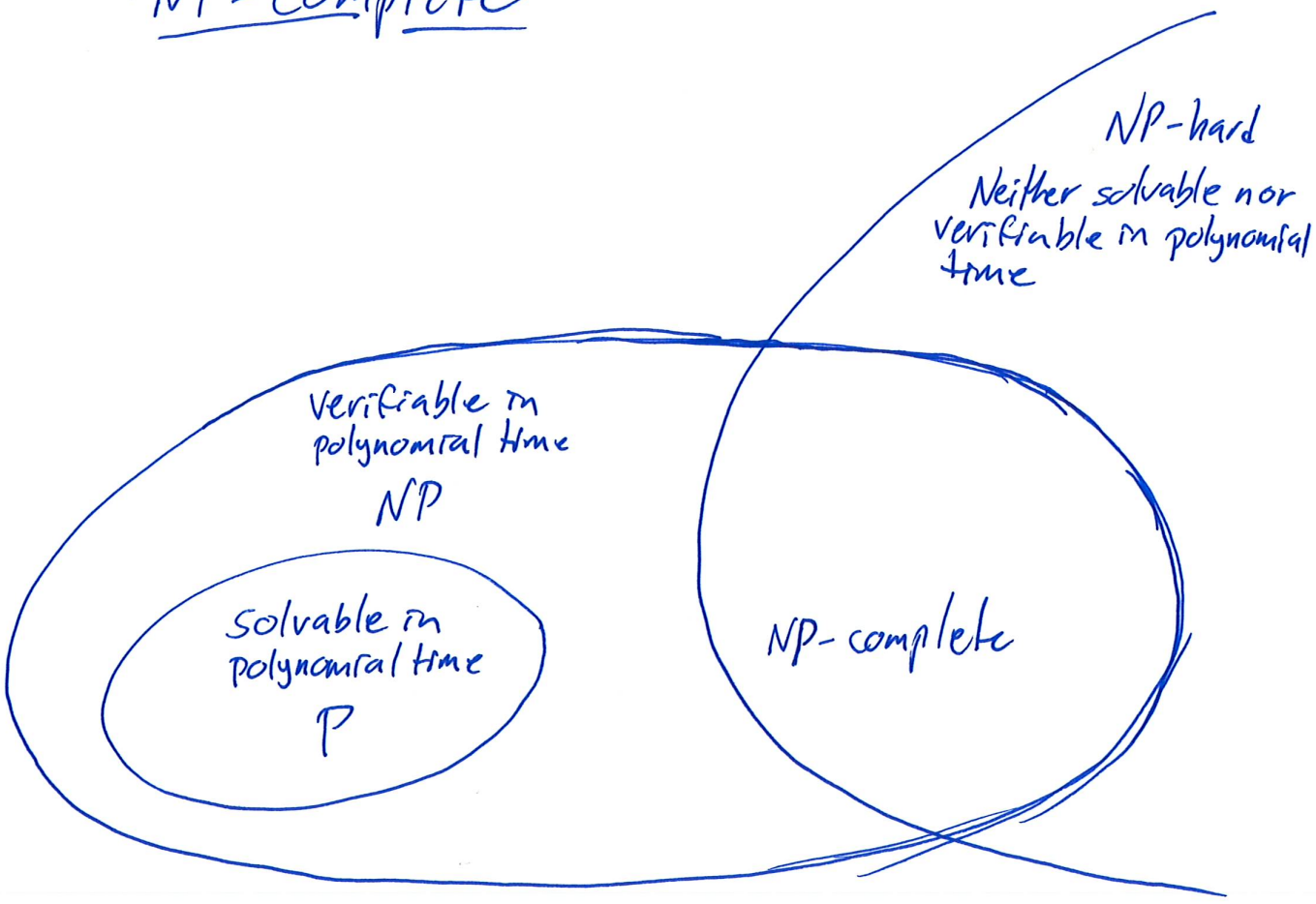
The set of problems that can be solved in polynomial time is called P

$$P \subseteq NP$$

NP-hard, NP-complete

A problem H is NP-hard if every problem in NP can be reduced in polynomial time to H

Some of the NP-hard problems can be verified in polynomial time, these we call NP-complete



Polynomial time reduction?

A problem A is NP-complete, if an NP problem B can be reduced to A in polynomial time

If there is no polynomial time reduction from any NP-problem, then A is not NP-complete

Why polynomial time reduction?

Because, if we find polynomial time solution procedure for A, then because of the polynomial time reduction, we have polynomial time solution for all NP-problems!

If reduction to A is exponential time, then a polynomial time solution of A will not help for polynomial time solution of other problems

Reductions are only interesting if they are easier than solving

Complexity

Basic backtracking algorithm: $O(nd^n)$

Forward checking: $O(nd^n)$

d: domain size
c: number of constraints

Arc Consistency algorithm, AC-3: $O(n^2d^3)$
$O(cd^3)$, Rem

Rina Dechter

FC cd^2

PLA cd^3 the same as AC-LA, FLA

FC often 100 times faster than plain BT
FC + MRV often 10000 times faster

~~AC-3 $O(n^2d^3)$~~

The halting problem

A program like:

```
print("Hello world!");
```

prints a string of characters on the screen and then halts

A program like:

```
while (true)
{
  //...
}
```

may or may not halt depending on what happens inside the while-loop, and probably also what inputs the program is given

Can we write a program that takes another program with some inputs, and determines whether that program halts or not?

The halting problem (ctd)

Assume now that we can write a program h, which takes an arbitrary program p, and that program's inputs i, and determines whether p halts or not:

$$h(p,i) = \begin{cases} \text{true} & \text{if } p \text{ halts on input } i \\ \text{false} & \text{if } p \text{ loops forever on input } i \end{cases}$$

If such a program h exists, then we could use it to create a program g, such that:

```
if (h(g,i))
    loop forever;
```

If you say I do, I don't.
If you say I don't, I do.

This is a major contradiction!

* If h determines that g halts on input i, then g will not halt, but loop forever.
* If h determines that g does not halt on input i ($h(g,i)$ returns false), g halts

Thus, the program h cannot exist!

This is one of CS's most fundamental results

IP posed as Presburger Arithmetic formula

On Presburger and stuff

$$\min \; p(x) = x_1 + x_2 + \dots$$
$$\text{s.t.} \quad x_1 + x_2 - 3x_3 \leq 42$$
$$x_1 + x_4 = 11$$
$$\dots$$

The constraints are one big conjunction

$$C(x) := (x_1 + x_2 - 3x_3 \leq 42) \wedge (x_1 + x_4 = 11) \wedge \dots$$

Then we pose the following claim:

$$\exists x^* \in D : C(x^*) \wedge \forall x \in D : C(x) \; p(x^*) \leq p(x)$$

where D is the domain of the variables,
$D := D_1 \times D_2 \times \dots$

So we have the problem

$$\langle X, D, C, p \rangle$$

and we are looking for assignments to the decision variables x_i such that the conjunction is satisfied

In addition, with IP (and MILP) we look for a satisfying assignment that is optimal given the obj fun

But this last part is not always the case. Sometimes we are only interested in finding a satisfying assignment!

Regard an LP/IP/MILP problem

$$\min \quad p = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n$$

$$\text{s.t} \quad b_{11} x_1 + b_{12} x_2 + \ldots + b_{1n} x_n = b_1$$
$$b_{21} x_1 + b_{22} x_2 + \ldots + b_{2n} x_n = b_2$$
$$\vdots$$
$$b_{m1} x_1 + b_{m2} x_2 + \ldots + b_{mn} x_n = b_m$$

$$0 \leq x_i \leq u_i$$

With rational coefficients, we can regard only IP problems, without loss of generality

We can formulate this as

$$\langle X, D, C, p \rangle$$

where $X = \{x_1, x_2 \ldots, x_n\}$ the set of variables

$D = D_1 \times D_2 \times \ldots \times D_n$ the domain given by $0 \leq x_i \leq u_i$

C the set of constraints

p the objective function to minimize

Given this, let us claim the following

$$\exists x^* \in D : x^* \text{ sat } C \wedge \forall v \in D : v \text{ sat } C, \quad p(x^*) \leq p(v)$$

This is a logical claim, it is either true or false

Our MILP solver can prove (or disprove) this, It can tell us:

* TRUE, and here is x^*
* FALSE, the problem is infeasible
* don't know yet, but let me work on it

From the decidability of Presburger arithmetic, we know that there can always be given a definitive TRUE/FALSE answer

Williams talks about problems that have NO obj fcn, 3.2.5 (p.29) "many practical problems involve no optimization"

Given constraints, finding a solution that satisfies all constraints may be by no means trivial.

Solving a mathematical programming model of the situation will at least find a feasible solution (if it exists)

May reveal "corner cases" that can show that the model is inaccurate or ill-defined.

Show infeasibility/non-satisfaction

One example is compiler optimization verification (program verification in general)

```
for(i=1; i<=10; i++)
  a[j+i]=a[j];
```

This fragment is intended to replicate the value of `a[j]` into the locations `a[j+1]` to `a[j+10]`. A compiler might generate the assembly code for the body of the loop as follows. Suppose variable `i` is stored in register `R1`, and variable `j` is stored in register `R2`:

```
R4 ⟵ mem[a+R2]     /* set R4 to a[j] */
R5 ⟵ R2+R1         /* set R5 to j+i */
mem[a+R5] ⟵ R4     /* set a[j+i] to a[j] */
R1 ⟵ R1+1          /* i++ */
```

Code that requires memory access is typically very slow compared with code that operates only on the internal registers of the CPU. Thus, it is highly desirable to avoid load and store instructions. A potential optimization for the code above is to move the load instruction for `a[j]`, i.e., the first statement above, out of the loop body. After this transformation, the load instruction is executed only once at the beginning of the loop, instead of 10 times. However, the correctness of this transformation relies on the fact that the value of `a[j]` does not change within the loop body. We can check this condition by comparing the index of `a[j+i]` with the index of `a[j]` together with the constraint that `i` is between 1 and 10:

$$i \geq 1 \wedge i \leq 10 \wedge j + i = j\,. \tag{5.2}$$

This formula has no satisfying assignment, and thus, the memory accesses cannot overlap. The compiler can safely perform the read access to `a[j]` only once. ⌟

5.1.1 Solvers for Linear Arithmetic

The **Simplex** method is one of the oldest algorithms for numerical optimization. It is used to find an optimal value for an objective function given a conjunction of linear constraints over real variables. The objective function and the constraints together are called a **linear program** (LP). However, since we are interested in the decision problem rather than the optimization problem, we cover in this chapter a variant of the Simplex method called **general Simplex** that takes as input a conjunction of linear constraints over the reals *without* an objective function, and decides whether this set is satisfiable.

Integer linear programming, or ILP, is the same problem for constraints over integers. Section 5.3 covers BRANCH AND BOUND, an algorithm for deciding such problems.

These two algorithms can solve conjunctions of a large number of constraints efficiently. We shall also describe two other methods that are considered less efficient, but can still be competitive for solving small problems.