

A circular graphic on the left side of the slide features a complex network of nodes and edges. The nodes are represented by small circles in various colors, including red, blue, and white, set against a dark background. The edges are thin lines connecting the nodes, forming a dense web of connections.

OPTIMISATION & OPERATIONS RESEARCH

1 Optimisation & linear programming

1.1 Introduction to Optimisation & Operations Research

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE



Optimisation

- ▶ making the best of something
- ▶ the action or process of making optimal

Operations research

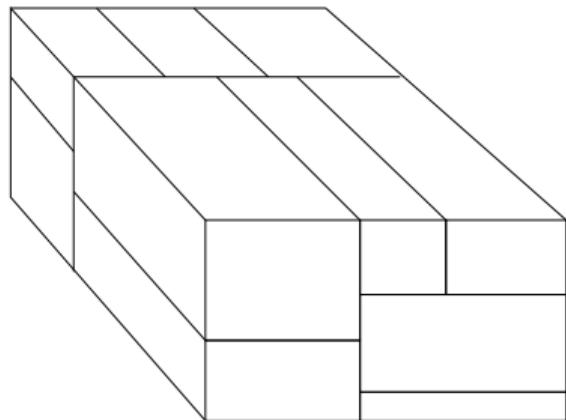
- ▶ interdisciplinary branch of applied mathematics
- ▶ uses maths, statistics, modelling and optimisation to arrive at optimal or near-optimal solutions to complex decision-making problems

Foam cutting

A company makes various grades of foam rubber (used in car seats, beds, etc).

The foam is made in blocks ($3.5\text{m} \times 3.5\text{m} \times 1\text{m}$), and cut into smaller rectangular blocks to order every day.

- ▶ Cuts are made by a guillotine, and so cuts go right through the block being cut.
- ▶ This can lead to waste: some blocks are the wrong shape to be useful.
- ▶ How can we minimise this waste?
- ▶ Research by University of Adelaide mathematicians saved this company \$800,000/year.



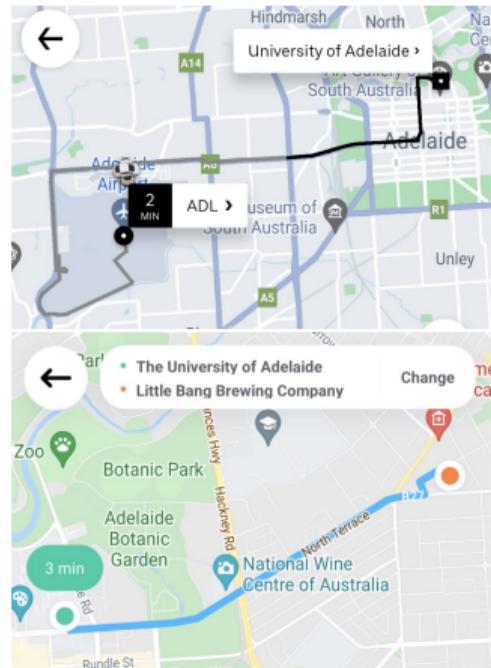
Some real world problems

Ride sharing (Uber/Didi etc.)

Users requests a ride from a specific origin to a specific destination.

- ▶ Design best vehicle routes taking into account:
 - ▶ capacity
 - ▶ duration
 - ▶ time window
 - ▶ pairing
 - ▶ traffic
- ▶ “Best” means to minimise cost (labour/fuel)

Cordeau (2006) *A Branch-and-Cut Algorithm for the Dial-a-Ride Problem*, Operations Research, 54 (3), 573-586.



Electricity distribution

Electricity grids can fail under certain conditions (too much demand, uneven supply, accidents) ... this can lead to power blackouts.

- ▶ How can blackouts be prevented?
 - ▶ Smooth supply with a big battery
 - ▶ What conditions should the battery be used?
- ▶ If a blackout occurs, how best to manage this?
 - ▶ Many options in how a power company can use its resources.
 - ▶ How to do this?
- ▶ Researchers from the University of Adelaide help SA Power Networks to address these questions.



Photos: ABC News (top), University of Adelaide (bottom)

A circular graphic on the left side of the slide features a complex network of nodes (small circles) connected by lines (edges). The colors of the nodes range from white to red, blue, and yellow. The background is a dark blue gradient with blurred red and blue circular bokeh effects.

OPTIMISATION & OPERATIONS RESEARCH

1 Optimisation & linear programming

1.2 Our First Problem

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE



Think of an optimisation problem...

- ▶ What is it that you ideally wish to achieve?
 - ▶ e.g., maximise profit
 - ▶ e.g., minimise risk

This is the *Objective*

- ▶ What is it that you have control over?
 - ▶ e.g., **who** should do W?
 - ▶ e.g., **how** many X should I make?
 - ▶ e.g., **where** should I make Y?
 - ▶ e.g., **when** should I do Z?
 - ▶ e.g., **what** model of V should I choose?

These are the *Variables*

- ▶ Are there any restrictions?
 - ▶ e.g., we can't make negative numbers of items
 - ▶ e.g., we have a limited resources or time

These are the *Constraints*

Our first problem

Example (A scheduling problem)

A manufacturing company makes three types of items: desks, chairs, and bed-frames, using metal and wood.

A single desk requires

- ▶ 2 hours of labour,
- ▶ 1 unit of metal and
- ▶ 3 units of wood.

One chair requires

- ▶ 1 hour of labour,
- ▶ 1 unit of metal and
- ▶ 3 units of wood.

Making one bed frame requires

- ▶ 2 hours of labour,
- ▶ 1 unit of metal and
- ▶ 4 units of wood.

Example (A scheduling problem: (cont))

In a given time period, there are

- ▶ 225 hours of labour available,
- ▶ 117 units of metal, and
- ▶ 420 units of wood.

The profit on

- ▶ one desk is \$13,
- ▶ one chair is \$12, and
- ▶ one bed frame is \$17.

Example (A scheduling problem: (cont))

The company wants a manufacturing schedule designed, which maximises profits without violating any constraint on resource availability during that time period.

So what are the steps in Formulation?

1. Read the question right through!
2. Tabulate any data
3. Identify the variables
4. Identify the objective
5. Formulate the constraints, *i.e.*, define the feasible region
6. Write down the LP (Linear Program)

L[~]O[~]O[~]k carefully.

Our first problem

2. Tabulate the *data*:

	Labour (hrs)	Metal	Wood	Profit
Desk	2	1	3	13
Chair	1	1	3	12
Bedframe	2	1	4	17
Total	225	117	420	

Our first problem

3. Define the *decision variables*:

x_1 to be the number of desks;

x_2 to be the number of chairs; and,

x_3 to be the number of bedframes,
made per time period.

Our first problem

4. Identify (formulate) the *objective function*:

Here we wish to maximise the profit, z , (in dollars), so we have:

$$\max z = 13x_1 + 12x_2 + 17x_3.$$

We might sometimes write this as

$$\underset{x}{\operatorname{argmax}} z = 13x_1 + 12x_2 + 17x_3.$$

which means find the *argument* x that maximises z .

Our first problem

5. Formulate the *constraints*, including any non-negativity constraints

$$\begin{array}{llllll} 2x_1 & + & x_2 & + & 2x_3 & \leq 225 & (\text{labour}) \\ x_1 & + & x_2 & + & x_3 & \leq 117 & (\text{metal}) \\ 3x_1 & + & 3x_2 & + & 4x_3 & \leq 420 & (\text{wood}) \\ x_1 \geq 0, & & x_2 \geq 0, & & x_3 \geq 0 & & (\text{non-negativity}) \end{array}$$

Question:

Should we include any other constraints on the x_i 's?

5. Formulate the *constraints*, including any non-negativity constraints

$$\begin{array}{rclcl} 2x_1 & + & x_2 & + & 2x_3 \leq 225 & (\text{labour}) \\ x_1 & + & x_2 & + & x_3 \leq 117 & (\text{metal}) \\ 3x_1 & + & 3x_2 & + & 4x_3 \leq 420 & (\text{wood}) \\ x_1 \geq 0, & & x_2 \geq 0, & & x_3 \geq 0 & (\text{non-negativity}) \end{array}$$

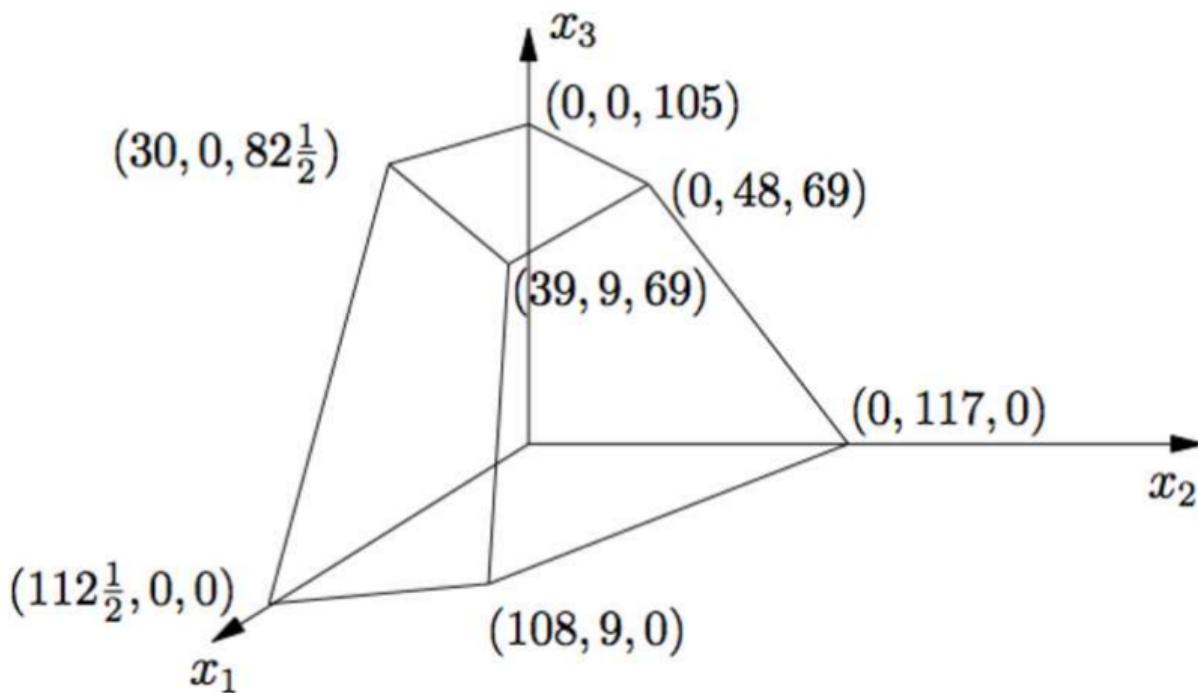
Question:

Should we include any other constraints on the x_i 's?

What about integer constraints?

Feasible region

$$\begin{array}{lclllll} 2x_1 & + & x_2 & + & 2x_3 & \leq & 225 & (\text{labour}) \\ x_1 & + & x_2 & + & x_3 & \leq & 117 & (\text{metal}) \\ 3x_1 & + & 3x_2 & + & 4x_3 & \leq & 420 & (\text{wood}) \\ x_1 \geq 0, & & x_2 \geq 0, & & x_3 \geq 0 & & (\text{non-negativity}) \end{array}$$





Our first problem

6. Write down the *linear program* (LP)

Our LP:

(objective function)

$$\max z = 13x_1 + 12x_2 + 17x_3$$

subject to

$$2x_1 + x_2 + 2x_3 \leq 225$$

$$x_1 + x_2 + x_3 \leq 117$$

$$3x_1 + 3x_2 + 4x_3 \leq 420$$

(non-negativity)

$$x_1 \geq 0, \quad x_2 \geq 0, \quad x_3 \geq 0$$

Note:

If the integer constraints were added, it would be an Integer Linear Program (ILP).

Our first problem

Matrix form of the problem:

$$\begin{aligned} \max \quad z &= \mathbf{c}^T \mathbf{x} + z_0 \\ \text{such that } A\mathbf{x} &\leq \mathbf{b} \\ \text{and } \mathbf{x} &\geq 0 \end{aligned}$$

where vector inequalities mean every element of the vector must satisfy the inequality.

A circular graphic on the left side of the slide features a complex network of nodes (small circles) connected by lines (edges). The colors of the nodes range from white to red, blue, and yellow. The background is a dark purple/blue gradient with blurred circular shapes of the same color palette.

OPTIMISATION & OPERATIONS RESEARCH

1 Optimisation & linear programming

1.3 Optimisation Revision

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

Optimisation is a task all human beings do - we all make decisions.

In any choice between a few options you want to make the “best” decision!

- ▶ minimise the cost of producing an app;
- ▶ shortest route to get hamburger or a drink;
- ▶ getting greatest exam mark, given a limited amount of study time.

Find the best solution to some objective often subject to some constraints.

Natural processes such as evolution are a form of optimisation

- ▶ **genetic algorithms** and **evolutionary computing** are optimisation techniques inspired by biology

Notation and Conventions



THE UNIVERSITY
of ADELAIDE

Throughout these notes we will try to use consistent notation.

- ▶ lower-case letters, e.g., x , will generally denote scalars
- ▶ boldface letters, e.g., \mathbf{x} , will denote (column) vectors, i.e.,

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

and we say $\mathbf{x} \in \mathbb{R}^n$

- ▶ upper case letters, e.g., A , will denote matrices.

There may be exceptions, but we will try to make them clear.

You probably remember that when maximising (or minimising) a real-valued function

$$\max f(x), \quad x \in \mathbb{R},$$

we look for x such that the derivative is zero

$$\frac{df(x)}{dx} = 0.$$

- ▶ we can include more than one variable (set partials to zero)
- ▶ we can include constraints (through Lagrange multipliers)

but the problems we consider in this course don't fit this pattern.

A Standard Linear Programming Problem

Linear programming:

$$\begin{aligned} \max \quad z &= \mathbf{c}^T \mathbf{x} + z_0 \\ \text{such that } A\mathbf{x} &\leq \mathbf{b} \\ \text{and } \mathbf{x} &\geq 0 \end{aligned}$$

Note the use of *program* here. It doesn't mean a computer program, but "a plan, schedule, or procedure."

The usage in optimisation goes back to Danzig in 1948.

We'll be looking at a different class of problems from $\max f(x), x \in \mathbb{R}$.

- ▶ Constraints are a *very* important part
- ▶ The objective function is linear, so $f' \neq 0$
- ▶ Variables might be restricted to be integers
- ▶ The dimension of problem might be very high (1000s)

We will consider new techniques which allow us to address some of these challenges.

What is the background we need to know to get started?

We'll be looking at [Linear Programs](#).

What do we need to know?

- ▶ Constraints define a region
 - ▶ we should understand the shape of that region, and the affect it has on the solution
 - ▶ the constraints are linear inequalities (or equalities), so what sort of space do they define?
- ▶ The objective function is also linear

We will need to use linear algebra!

A circular graphic on the left side of the slide features a complex network of nodes (small circles) connected by lines (edges). The colors of the nodes range from white to red, blue, and yellow. The background is a dark blue gradient with blurred red and white circular bokeh effects.

OPTIMISATION & OPERATIONS RESEARCH

1 Optimisation & linear programming

1.4 The geometry of linear programming

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



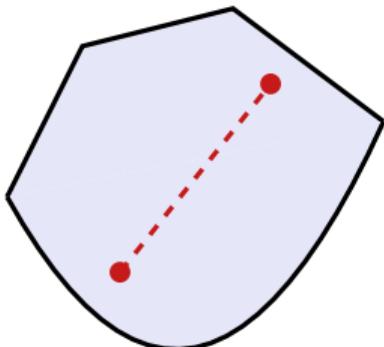
THE UNIVERSITY
*of*ADELAIDE

Definition (A convex set)

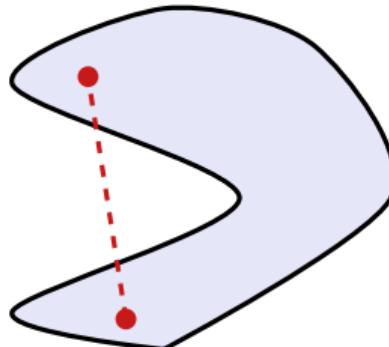
A set $S \subseteq \mathbb{R}^n$ is called a *convex set* if for all $x, y \in S$ and $0 \leq \alpha \leq 1$

$$x, y \in S \implies \alpha x + (1 - \alpha)y \in S,$$

i.e., the line segment joining x and y lies in S .



convex



non-convex

Convex Functions

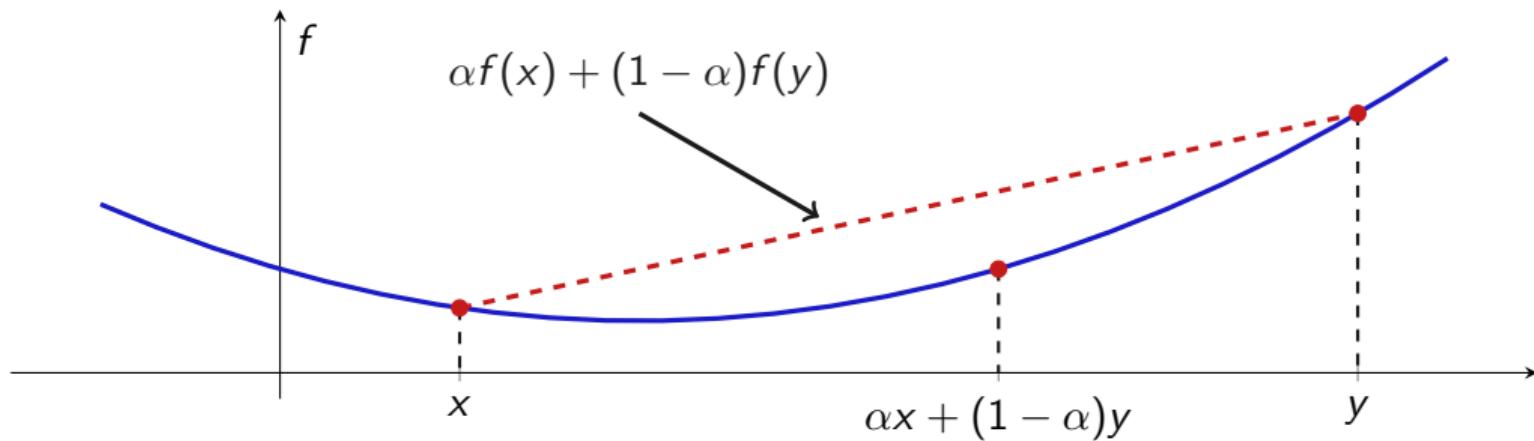


Definition (A convex function)

A function $f : S \rightarrow \mathbb{R}$ is a *convex function* if for all $x, y \in S$ and $0 \leq \alpha \leq 1$

$$\alpha f(x) + (1 - \alpha)f(y) \geq f(\alpha x + (1 - \alpha)y)$$

i.e., chords don't lie below the function.





Lemma

Given m convex functions $g_i(\mathbf{x})$, the set Ω defined by

$$\Omega = \{ \mathbf{x} \in \mathbb{R}^n \mid g_i(\mathbf{x}) \leq 0, \text{ for } i = 1, 2, \dots, m \},$$

is convex.

Proof

Start by considering a *single* constraint.

Take the set

$$\Omega_i = \{ \mathbf{x} \in \mathbb{R}^n \mid g_i(\mathbf{x}) \leq 0 \}.$$

Take two points in this set \mathbf{x} and \mathbf{y} , then consider a point on the chord between them, $\mathbf{z} = \alpha\mathbf{x} + (1 - \alpha)\mathbf{y}$, for $\alpha \in [0, 1]$. As $g_i(\cdot)$ is convex,

$$g_i(\mathbf{z}) = g_i(\alpha\mathbf{x} + (1 - \alpha)\mathbf{y}) \leq \alpha g_i(\mathbf{x}) + (1 - \alpha)g_i(\mathbf{y}),$$

Now $g_i(\mathbf{x}) \leq 0$ and $g_i(\mathbf{y}) \leq 0$ because $\mathbf{x}, \mathbf{y} \in \Omega_i$ and α and $1 - \alpha \geq 0$, so

$$g_i(\alpha\mathbf{x} + (1 - \alpha)\mathbf{y}) \leq 0,$$

and hence $\mathbf{z} \in \Omega_i$, i.e., any point on a chord between two points in the set must also be in the set, so Ω_i is convex.

Proof (continued).

Now we move on to sets with multiple constraints.

$$\Omega = \{ \mathbf{x} \in \mathbb{R}^n \mid g_i(\mathbf{x}) \leq 0, \text{ for } i = 1, 2, \dots, m \} = \cap_{i=1}^m \Omega_i$$

We note two properties of convex sets:

- ▶ The empty set and the set \mathbb{R}^n are convex.
- ▶ The intersection of two convex sets is convex.

Together these mean (e.g., by induction) that $\Omega = \cap_{i=1}^m \Omega_i$ must be convex. □

Lemma

The region defined by a set of linear inequalities $A\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \geq 0$ is convex.

Proof.

- ▶ It is almost self-evident that a straight line is convex.
 - ▶ a chord along the straight line, is just that
 - ▶ they aren't *strictly* convex as the chord doesn't lie above the line, but that doesn't matter here
- ▶ From the previous lemma, the set of linear constraints $a_i x \leq b_i$ and $x_i \geq 0$ defines a convex set.





Theorem

If S is a convex set and f is a convex (concave) function, then any local minimum (maximum) of f is a global minimum (maximum).

Proof.

If x^* is a local minimum, then by definition for some small h

$$f(x^* + h) \geq f(x^*)$$

Now choose any other point $x > x^*$, we can draw a chord between x^* and x , and this chord must lie no lower than $f(\cdot)$. For instance, given $h > 0$, at $x^* + h$ the chord will be $\geq f(x^* + h)$. That implies the chord from $(x^*, f(x^*))$ to $(x, f(x))$ has non-negative slope, and hence $f(x) > f(x^*)$, which implies a global minimum.

We can repeat the argument for $x < x^*$.



Note that the result doesn't require a differentiable function.

Obviously, the previous result can be generalised to n dimensions, and we see that

- ▶ Linear inequalities define a convex set, and the linear objective function is both convex and concave
- ▶ Hence the local minimum of a linear program will also be its global minimum
- ▶ The objective function here is linear, so $f' \neq 0$ anywhere, so the local minimum (if it exists) will occur on the boundary
- ▶ So the minimum of a linear program (if it exists) is somewhere on the boundary

A quick example

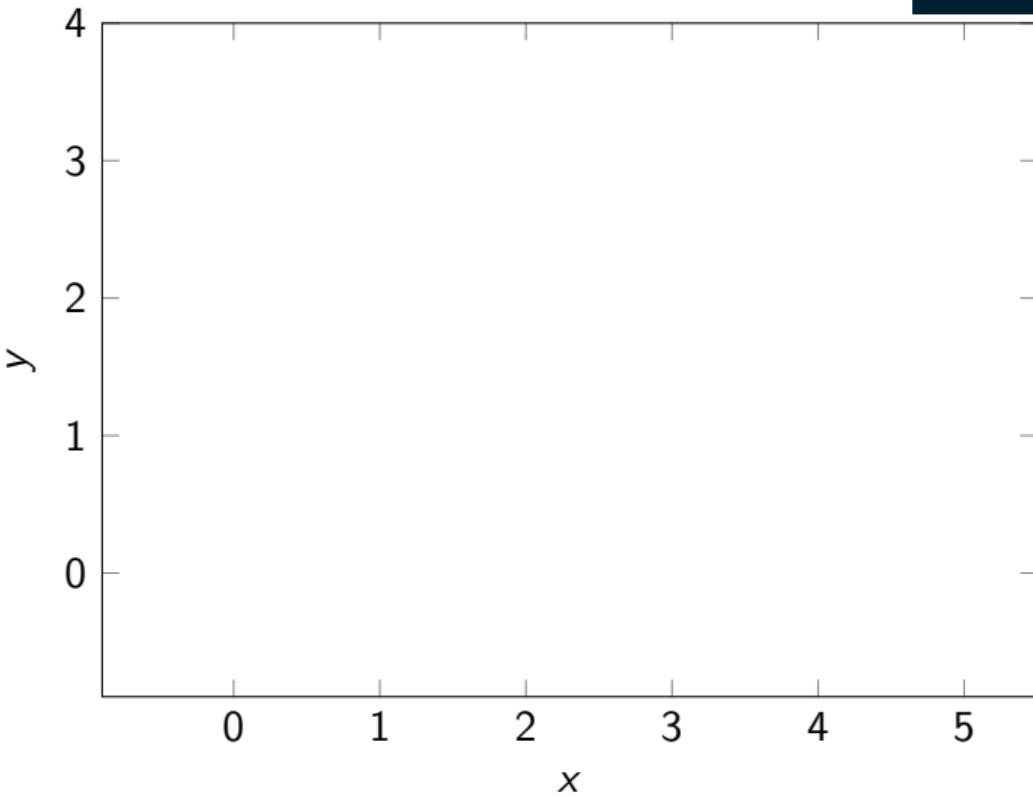
$$\max \quad z = 2x + 3y$$

subject to

$$2x + 4y \leq 12$$

$$x + y \leq 4$$

with $x \geq 0, y \geq 0$.



A quick example

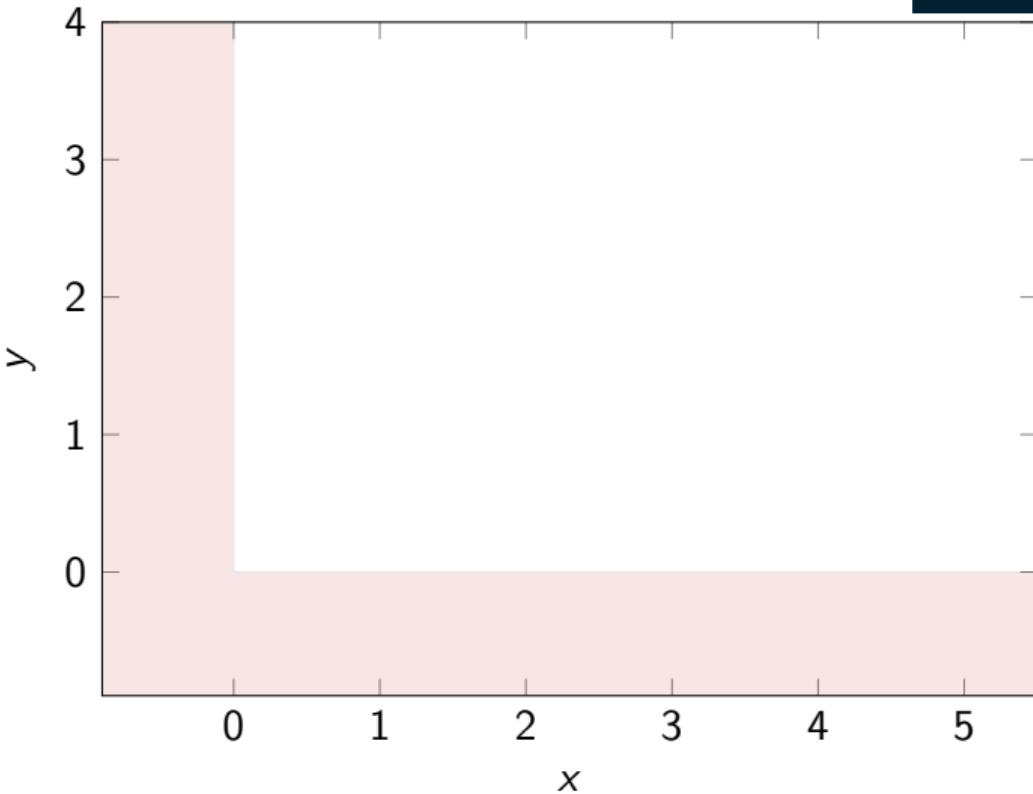
$$\max \quad z = 2x + 3y$$

subject to

$$2x + 4y \leq 12$$

$$x + y \leq 4$$

with $x \geq 0, y \geq 0$.



A quick example

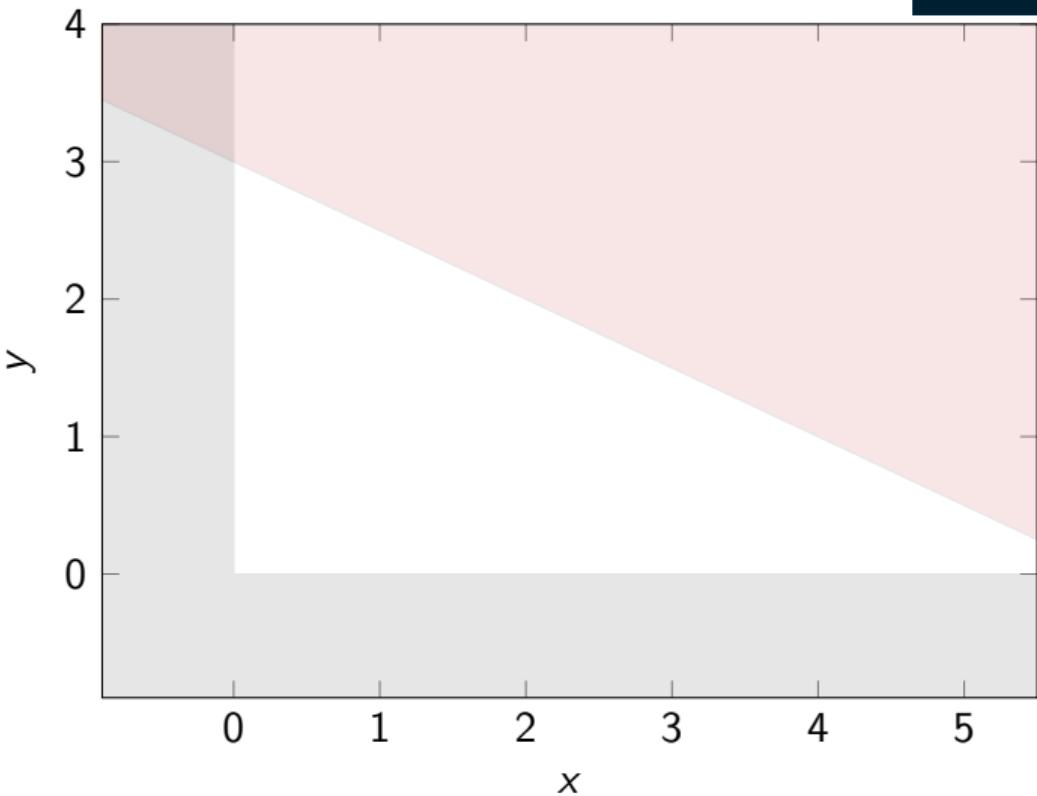
$$\max \quad z = 2x + 3y$$

subject to

$$2x + 4y \leq 12$$

$$x + y \leq 4$$

with $x \geq 0, y \geq 0$.



A quick example

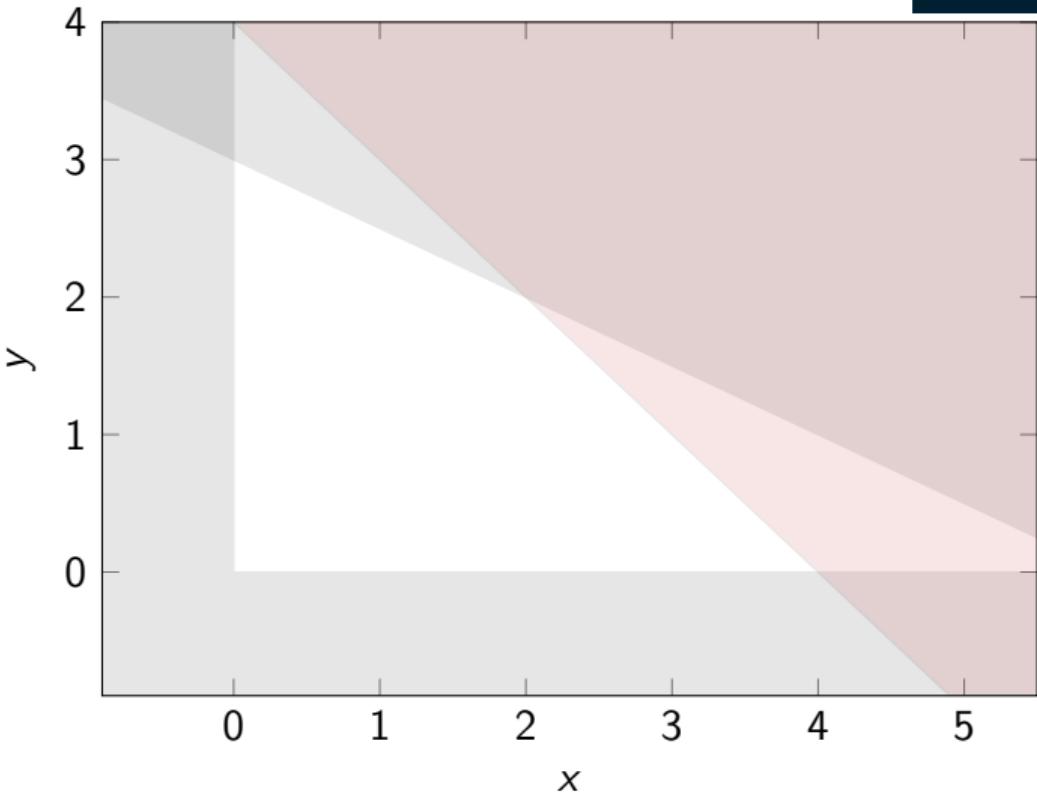
$$\max \quad z = 2x + 3y$$

subject to

$$2x + 4y \leq 12$$

$$x + y \leq 4$$

with $x \geq 0, y \geq 0$.



A quick example

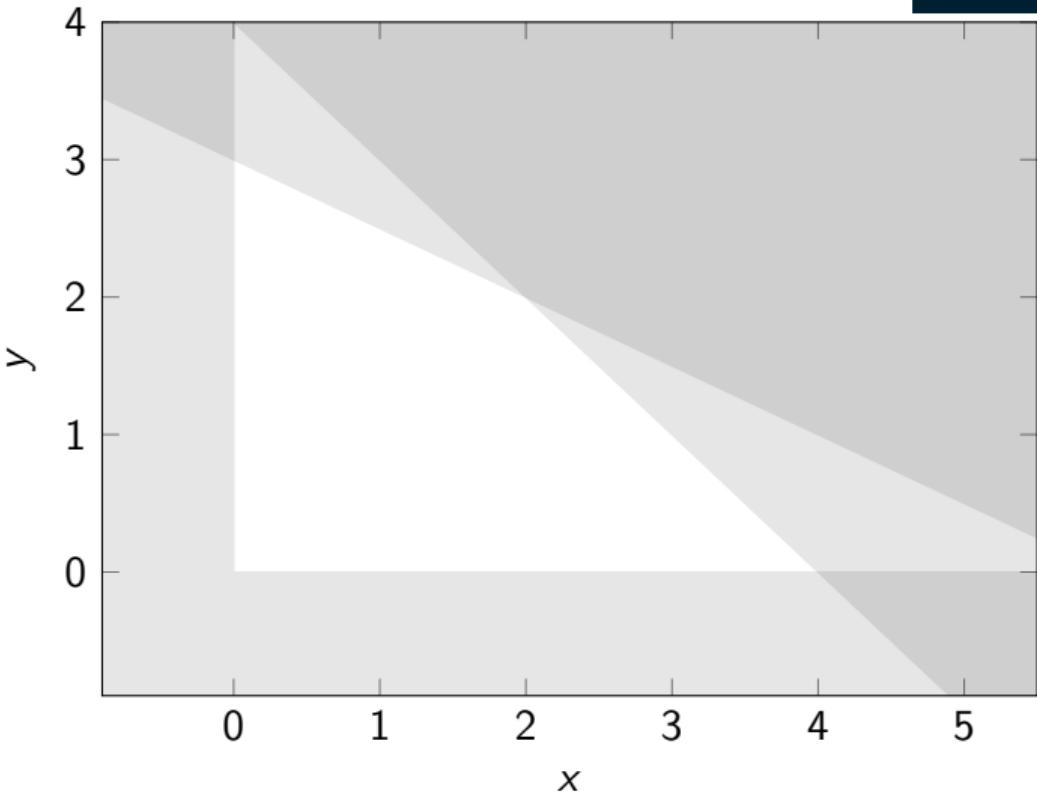
$$\max \quad z = 2x + 3y$$

subject to

$$2x + 4y \leq 12$$

$$x + y \leq 4$$

with $x \geq 0, y \geq 0$.



A quick example

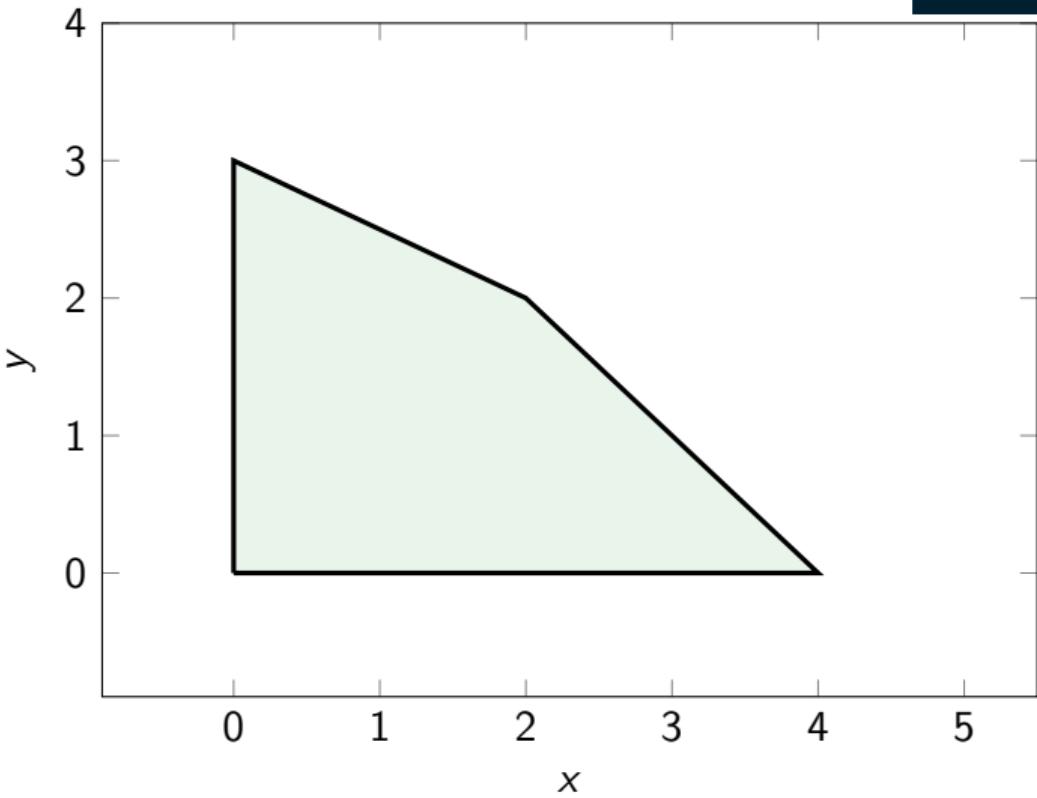
$$\max \quad z = 2x + 3y$$

subject to

$$2x + 4y \leq 12$$

$$x + y \leq 4$$

with $x \geq 0, y \geq 0$.



A quick example

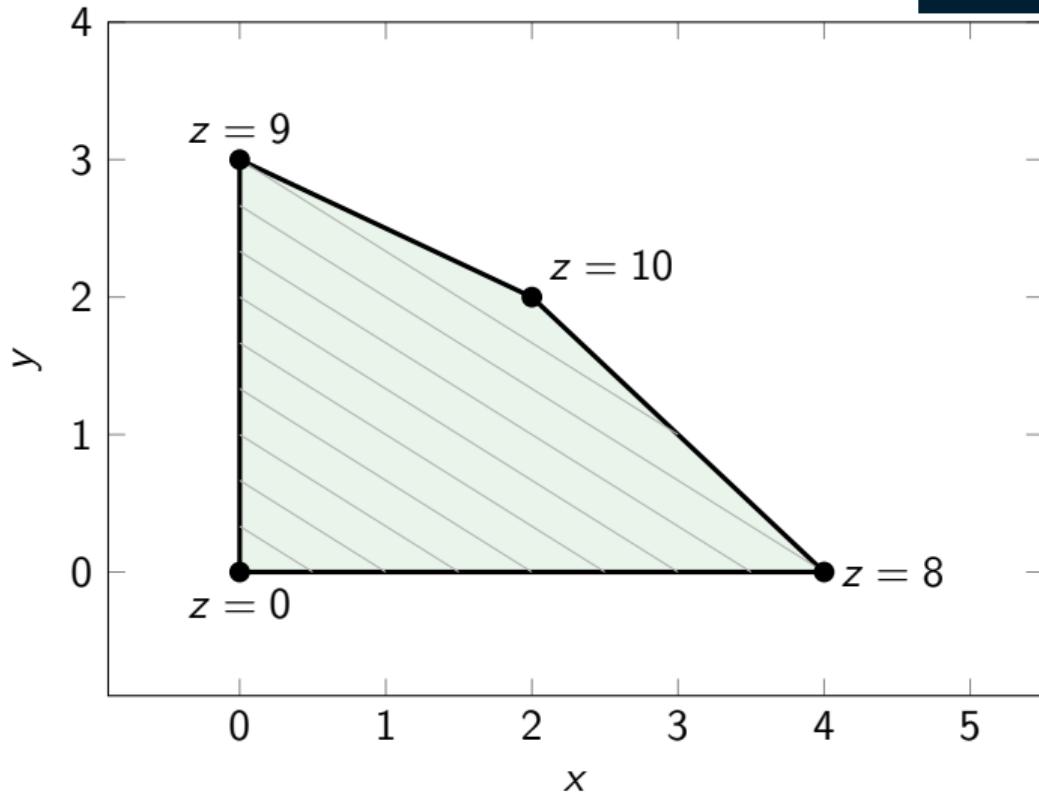
$$\max \quad z = 2x + 3y$$

subject to

$$2x + 4y \leq 12$$

$$x + y \leq 4$$

with $x \geq 0, y \geq 0$.



A circular graphic on the left side of the slide features a complex network of interconnected nodes (dots) and edges (lines). The nodes are colored in shades of red, orange, yellow, and white, set against a dark blue background. The network is dense and organic, representing concepts like connectivity, data flow, or optimization problems.

OPTIMISATION & OPERATIONS RESEARCH

1 Optimisation & linear programming

1.5 Linear equations

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

Given a *system of linear equations* $Ax = \mathbf{b}$ there are three possibilities:

- ▶ They have no solutions
- ▶ They have exactly one solution
- ▶ They have infinitely many solutions

And we know how to test for these cases, and solve when possible.

For a square matrix A we can solve $A\mathbf{x} = \mathbf{b}$ by using the matrix inverse, e.g.,

$$\mathbf{x} = A^{-1}\mathbf{b}$$

But this has issues

- ▶ in our problem, the matrix might not be square
- ▶ even if it were square it might not be invertible (singular)
- ▶ even if it is invertible, this approach is
 - ▶ numerically inefficient
 - ▶ potentially numerically unstable

So we **don't** do this

Gauss-Jordan elimination

Carl Friedrich Gauss (1777–1855)



Though apparently was known to Chinese mathematicians around the birth of Christ.



Gauss-Jordan elimination

- ▶ We augment the matrix A to include the RHS, i.e., $M = [A \mid \mathbf{b}]$
- ▶ We perform an allowed set of operations $M \rightarrow M'$
 1. Interchange two rows
 2. Perform a *pivot* at element (i, j) which combines
 - 2.1 Multiply a row by a nonzero number; and,
 - 2.2 Add a multiple of one row to another row,
such that element $M'_{i,j} = 1$ and $M'_{k,j} = 0$ for $k \neq i$

The allowed operations result in an *equivalent* augmented matrix.

- ▶ The goal is to manipulate it into a form where the solution is obvious. For instance, if we could manipulate it into the form

$$[I \mid \mathbf{b}^*]$$

Then we can read off the solution $\mathbf{x} = \mathbf{b}^*$ because this augmented matrix corresponds to the equations

$$I\mathbf{x} = \mathbf{b}^*$$

Definition

Given a set of equations $Ax = \mathbf{b}$ the

- ▶ coefficient matrix is A
- ▶ augmented matrix is $[A|\mathbf{b}]$
- ▶ The augmented matrix contains all the information about the equations
 - ▶ there is a 1:1 correspondence between the augmented array and a set of equations
- ▶ We work on that to keep all the relevant data together
 - ▶ row operations on the augmented matrix convert to a new augmented matrix, corresponding to a set of equations which have the same solutions



Gauss-Jordan elimination example

Equations	Augmented matrix			
$x_1 + 3x_2 + x_3 = 9$	1	3	1	9
$x_1 + x_2 - x_3 = 1$	1	1	-1	1
$3x_1 + 11x_2 + 5x_3 = 35$	3	11	5	35

Gauss-Jordan elimination example



Gauss-Jordan elimination example

Equations			Augmented matrix		
$x_1 + 3x_2 + x_3 = 9$			1	3	1 9
$x_1 + x_2 - x_3 = 1$			1	1	-1 1
$3x_1 + 11x_2 + 5x_3 = 35$			3	11	5 35
$x_1 + 3x_2 + x_3 = 9$			1	3	1 9
$- 2x_2 - 2x_3 = -8$			0	-2	-2 -8
$2x_2 + 2x_3 = 8$			0	2	2 8
$x_1 - 2x_3 = -3$			1	0	-2 -3
$x_2 + x_3 = 4$			0	1	1 4
$0 = 0$			0	0	0 0



Gauss-Jordan elimination example

Equations			Augmented matrix		
$x_1 + 3x_2 + x_3 = 9$			1	3	1 9
$x_1 + x_2 - x_3 = 1$			1	1	-1 1
$3x_1 + 11x_2 + 5x_3 = 35$			3	11	5 35
$x_1 + 3x_2 + x_3 = 9$			1	3	1 9
$- 2x_2 - 2x_3 = -8$			0	-2	-2 -8
$2x_2 + 2x_3 = 8$			0	2	2 8
$x_1 - 2x_3 = -3$			1	0	-2 -3
$x_2 + x_3 = 4$			0	1	1 4
$0 = 0$			0	0	0 0

We performed 2 pivots

1. `pivot(1,1)` (a pivot on the (1,1) element of the augmented matrix)
2. `pivot(2,2)`

Gauss-Jordan elimination example

In this form

Equations	Augmented array			
$x_1 - 2x_3 = -3$	1	0	-2	-3
$x_2 + x_3 = 4$	0	1	1	4
$0 = 0$	0	0	0	0

we can read off the solution. There is a degree of freedom, because the last equation is always true. This gives us a *free* variable: here we take it to be $x_3 = t$, and the solution will be

$$\mathbf{x} = (-3, 4, 0) + t(2, -1, 1)$$

We could choose to set $x_3 = 0$, and get a solution $(-3, 4, 0)$.



- ▶ Formally we perform *elementary row operations*
 - ▶ Swap two rows.
 - ▶ Multiply a row by a nonzero scalar.
 - ▶ Add a scalar multiple of one row to another.
- ▶ We aim to put the tableau in *reduced row echelon form*
 - ▶ the lower-triangular part of the augmented matrix are all zeros
 - ▶ for every non-zero row, the leading coefficient is to the right of the leading coefficient of the row above
 - ▶ leading coefficients are 1
- ▶ Gauss-Jordan is a numerically unstable procedure in some cases
 - ▶ other approaches, e.g., QR decomposition, are often preferred



For optimisation, we don't want a single solution to the constraints, or the optimisation is trivial.

Take $A\mathbf{x} = \mathbf{b}$, where A is of size $m \times n$

- ▶ n variables
- ▶ m equations

With *full rank* A , and $n > m$:

- ▶ There are ∞ solutions
 - ▶ we don't have enough information to select one
 - ▶ Choose $(n - m)$ variables to be 0. Think of this as either
 - ▶ adding $n - m$ additional equations $x_i = 0$
 - ▶ reducing the number of variables down to m
- then there will (hopefully) be a unique solution for the other m variables

Definition (Basic solution)

A *basic solution* to $Ax = \mathbf{b}$ is a solution with at least $n - m$ zero variables.

Definition (Non-degenerate basic solution)

A basic solution is *non-degenerate* iff **exactly** $n - m$ variables are zero.

It's *degenerate*¹ if there are more than $n - m$ zeros.

Definition (Basic and non-basic variables)

For a non-degenerate basic solution, the m non-zero variables are called *basic*, and the $n - m$ zero variables are *non-basic* or *free*.

¹In mathematics, we call a case degenerate when it is qualitatively different from the other solutions, and thus belongs to a different class of (often simpler) solution. Often its features are lost under small perturbations. For example, a line is a degenerate parabola.

A circular graphic on the left side of the slide features a complex network of nodes (small circles) connected by lines (edges). The colors of the nodes range from white to red, blue, and yellow. The background is a dark purple/blue gradient with blurred circular shapes of the same color palette.

OPTIMISATION & OPERATIONS RESEARCH

1 Optimisation & linear programming

1.6 Linear programming: introduction

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE



We will need some tools to understand the shape of a region defined by linear inequalities

- ▶ We have n variables x_i , so $\mathbf{x} \in \mathbb{R}^n$
- ▶ We have m inequalities defining the shape of the *feasible* region

$$A\mathbf{x} \leq \mathbf{b},$$

where

- ▶ A is an $m \times n$ matrix
- ▶ \mathbf{b} is a length m column vector
- ▶ Also we have n non-negativity constraints $\mathbf{x} \geq \mathbf{0}$

Let's see what we can say/do about this region.

Note that when we use \geq or \leq for vectors in this course, we mean, for each element of the vector.

The Feasible Region



THE UNIVERSITY
of ADELAIDE

There are four possibilities: the feasible region could be

- ▶ equal to \mathbb{R}^n
 - ▶ only if there are no constraints
- ▶ empty
 - ▶ if the constraints leave no feasible points
- ▶ a subset of \mathbb{R}^n
 - ▶ bounded
 - ▶ unbounded

We'll need to think about this a little more, but there is a problem we do know something about: solving linear *equations*.

Converting to standard form

We will always want to put problems into the standard form above, which has

$$Ax \leq b$$

If you have any \geq , then convert them to \leq by multiplying the inequality by -1

Example

Replace

$$2x - 4y \geq 3$$

with

$$-2x + 4y \leq -3$$

Converting to standard form

We will always want to put problems into the standard form above, which has

$$\mathbf{x} \geq 0$$

If any x_i are free, replace by two new variables $x_i^+ \geq 0$ and $x_i^- \geq 0$ such that
 $x_i = x_i^+ - x_i^-$



Converting to standard form

Example

$$\begin{aligned} \max z &= 3x_1 + 2x_2 \\ \text{subject to} \\ 2x_1 + x_2 &\leq 5 \\ x_1 + 3x_2 &\leq 7 \\ x_1 \geq 0, \quad x_2 &\text{ free} \end{aligned}$$

and replace it with

$$\begin{aligned} \max z &= 3x_1 + 2x_2^+ - 2x_2^- \\ \text{subject to} \\ 2x_1 + x_2^+ - x_2^- &\leq 5 \\ x_1 + 3x_2^+ - 3x_2^- &\leq 7 \\ x_1 \geq 0, \quad x_2^+ \geq 0, \quad x_2^- &\geq 0 \end{aligned}$$

and when you finally obtain a solution, go back to $x_2 = x_2^+ - x_2^-$

A circular graphic on the left side of the slide features a complex network of nodes and edges. The nodes are represented by small circles in shades of red, blue, and white, set against a dark background with blurred, glowing circular bokeh effects. The edges are thin lines connecting the nodes, forming a web-like structure.

OPTIMISATION & OPERATIONS RESEARCH

1 Optimisation & linear programming

1.7 Linear programming: inequalities and equations

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

Converting Inequalities to Equations

We want to use 1st year maths – in particular your experience with linear equations so we need to convert the inequalities into equations.

The standard process is to introduce slack variables: e.g.,

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$$

becomes

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n + s_1 = b_1$$

where s_1 is a *slack variable*.

By construction $s_1 \geq 0$.

Inequalities and Equations

In matrix form

$$A' \mathbf{x}' \leq \mathbf{b}$$

becomes

$$A \mathbf{x} = \mathbf{b}$$

where

$$A = [A' \mid I], \quad \mathbf{x} = \begin{pmatrix} x'_1 \\ \vdots \\ x'_{n'} \\ s_{n'+1} \\ \vdots \\ s_{n'+m} \end{pmatrix}$$

So now there are m equations and $n = n' + m$ variables.

$$A' \underline{\mathbf{x}}' \leq \mathbf{b}$$

 \Leftrightarrow

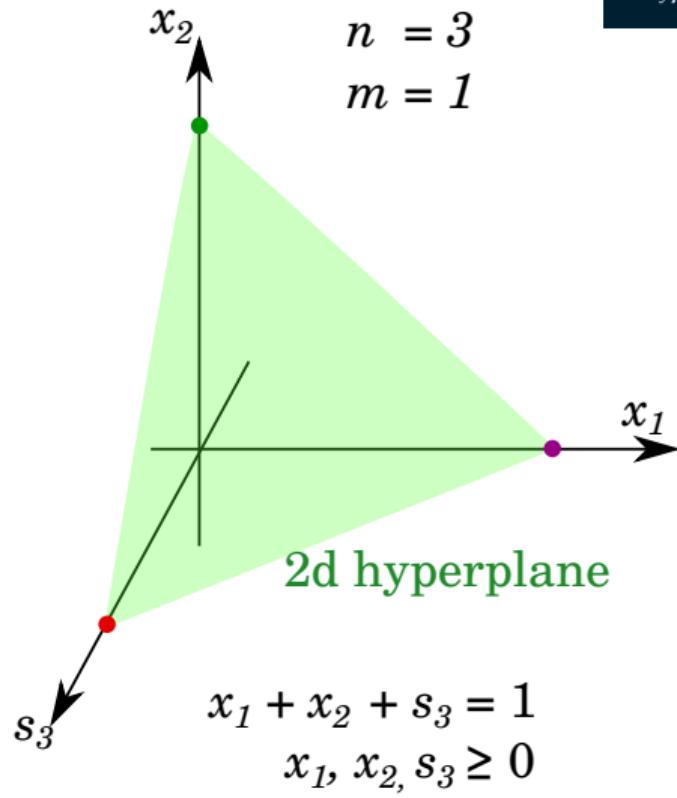
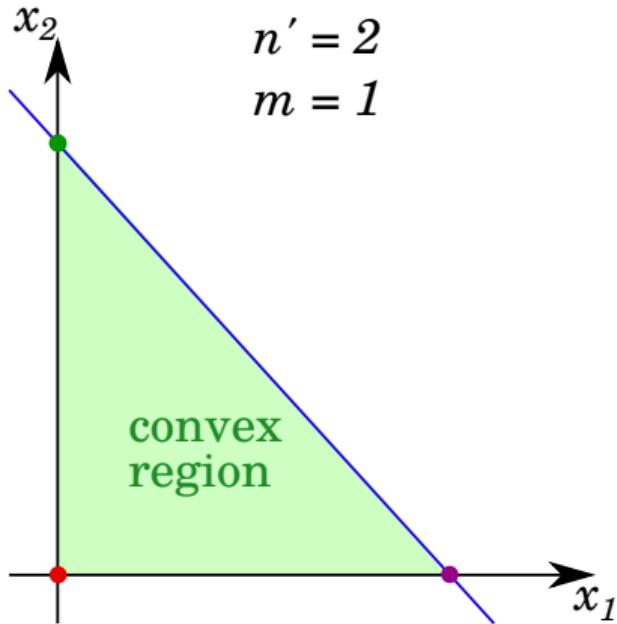
$$A\mathbf{x} = \mathbf{b}$$

- ▶ m constraints (inequalities)
 - ▶ n' variables
 - ▶ A' is an $m \times n'$ matrix
 - ▶ defines a region which is a convex *polytope* of $\mathbb{R}^{n'}$
 - ▶ vertices are a feasible intersection point of m of the boundary planes
 - ▶ non-negativity restricts to positive quadrant
-
- ▶ m constraints (equalities)
 - ▶ $n = n' + m$ variables
 - ▶ A is an $m \times n$ matrix
 - ▶ defines an n' dimensional hyperplane in \mathbb{R}^n
 - ▶ “vertices” are points where the hyperplane intersects axes
 - ▶ $n' = n - m$ of the variables are 0
 - ▶ non-negativity restricts to positive quadrant

Inequalities and Equations



THE UNIVERSITY
of ADELAIDE



- ▶ We need to become expert at converting back and forth between equalities and inequalities, and their interpretation.
 - ▶ when you are on a boundary, it means
 - ▶ either a slack variable corresponding to an inequality is zero

Example

original inequality	$x + y \leq 10$
slack variable equality form	$x + y + s = 10$
on the boundary $s = 0$	$x + y = 10$

- ▶ or one of the $x_i = 0$
- ▶ vertices are intersections of boundaries, so several variables must be zero, one for each boundary that's intersecting
- ▶ We also need to see how to convert a LP back and forth between equality and inequality form.



Converting from equality to inequality

Imagine we want to convert the equality

$$x + y = 10$$

Into an inequality form. We can replace the equality with

$$x + y \leq 10$$

$$x + y \geq 10$$

Or, in standard form

$$x + y \leq 10$$

$$-x - y \leq -10$$

1 Optimisation & linear programming

1.8 Linear programming: standard form

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

There are several alternatives we could end up with in all of these conversions, so we choose one standard, that we will always aim for.

Definition (Standard equality form)

An LP of the form

$$\begin{aligned} \max \quad & z = \mathbf{c}^T \mathbf{x} \\ \text{subject to } & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

is said to be in *standard equality form*.



Standard equality form of a LP

Converting to standard form

1. Convert into standard *inequality* form
 - ▶ this isn't strictly necessary, but makes everything more consistent
2. Convert inequalities into equalities
 - ▶ introduce slack variables
 - ▶ coefficients of **c** corresponding to slack variables are 0
3. Convert to a *max*²
 - ▶ easy to convert by taking $\min z = \max(-z)$

Example

$$\max z = 2x_1 + 4x_2$$

is equivalent to

$$\min w = -2x_1 - 4x_2$$

²MATLAB's `linprog` works with *min* problems, so you need to know how to convert, backwards and forwards.



Converting a LP

Example

Let us consider the LP:

$$\begin{aligned} \max z &= x_1 + x_2 \\ \text{subject to} \\ x_1 + 2x_2 &\leq 6 \\ x_1 - x_2 &\leq 3 \\ x_1 \geq 0, \quad x_2 \geq 0. \end{aligned}$$

We write this as a system of linear equations, by introducing 2 slack variables x_3 and x_4 , and leaving the objective unchanged

$$\begin{aligned} \max z &= x_1 + x_2 \\ \text{subject to} \quad x_1 + 2x_2 + x_3 &= 6 \\ x_1 - x_2 + x_4 &= 3 \\ x_1 \geq 0, \quad x_2 \geq 0, \quad x_3 \geq 0, \quad x_4 \geq 0. \end{aligned}$$

Note we often don't use special notation for the slack variables.

A circular graphic on the left side of the slide features a complex network of nodes (small circles) connected by lines (edges). The colors of the nodes range from white to red, blue, and yellow. The background is a dark purple/blue gradient with blurred circular shapes of the same color palette.

OPTIMISATION & OPERATIONS RESEARCH

1 Optimisation & linear programming

1.9 Linear programming: solutions

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE



Recall the following:

Definition (Basic solution)

A *basic solution* to $Ax = \mathbf{b}$ is a solution with at least $n - m$ zero variables.

We can add to this:

Definition (Basic feasible solution)

If a basic solution satisfies $x \geq \mathbf{0}$, then it is called a *basic feasible solution*.

That is, the solution is feasible if it also satisfies the non-negativity requirements (for the original variables, and the slack variables).



We already know that the feasible set for a LP (in inequality form) is a convex set. We can add to this

Definition (Extreme points)

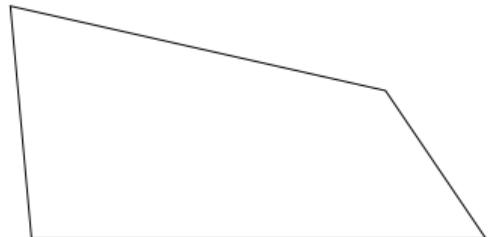
We say $x \in S$ is an *extreme point* of convex set S if

$$x = \lambda y + (1 - \lambda)z$$

for $y, z \in S$ and $0 < \lambda < 1$ implies $x = y = z$.

Intuitively this means that extreme points are not in the interior of any line segment inside the set.

For a convex polytope (such as defined by $A'x' \leq \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$) the extreme points are the vertices.



Theorem

If an LP has a finite optimum, it has an optimum at an extreme point of the feasible set.

For LP problems the feasible set will always have a finite number of extreme points (vertices). This suggests a **naïve** algorithm:

1. Find all the vertices of the feasible set
2. Choose the optimum.

How do we find vertices?



THE UNIVERSITY
of ADELAIDE

Theorem

The basic feasible solutions of $Ax = b$ are the extreme points of the feasible set $A'x' \leq b$, $x \geq 0$.

Unfortunately, we can not typically identify which basic solutions will be feasible *a priori* (i.e., which are vertices)

New naïve algorithm

1. Find all the basic solutions
2. Test to see if they are feasible
3. Choose the optimum of the basic feasible solutions

Somehow we also need to check for boundedness at the same time.

Example (again)

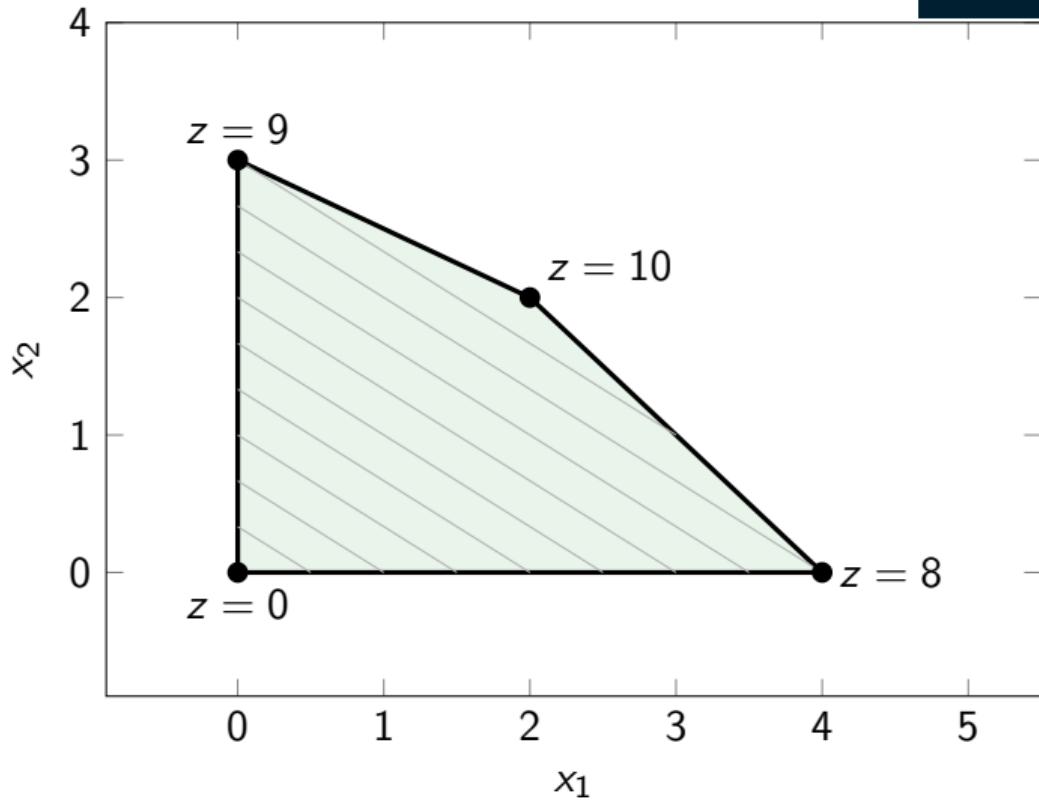
$$\max \quad z = 2x_1 + 3x_2$$

subject to

$$2x_1 + 4x_2 \leq 12$$

$$x_1 + x_2 \leq 4$$

with $x \geq 0$.



A circular graphic on the left side of the slide features a complex network of interconnected nodes (dots) and edges (lines). The nodes are colored in shades of red, orange, yellow, and white, set against a dark blue background. The network is dense and organic, representing concepts like connectivity or data flow.

OPTIMISATION & OPERATIONS RESEARCH

1 Optimisation & linear programming

1.10 Rank

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

Linear equations and redundancy

Given $A\mathbf{x} = \mathbf{b}$, where A is of size $m \times n$

- ▶ n variables (unknowns)
- ▶ m equations (pieces of information)

Naïvely we have m pieces of information, and n variables, so we might expect a unique solution for $m \geq n$. However, some pieces of information (equations) might be redundant.

- ▶ examples
 - ▶ if two rows are the same, then they don't provide any extra information
 - ▶ if one row is a scalar multiple of another, then it doesn't provide any extra information, e.g.,

$$\begin{aligned}x_1 + x_2 &= 3 \\2x_1 + 2x_2 &= 6\end{aligned}$$

- ▶ how do we assess redundancy in general?

Definition (Rank)

The *rank* of a matrix A is the number of *linearly-independent* rows³.

A matrix is *full-rank* if it has the maximum rank for its size (the minimum of n and m for an $m \times n$ matrix).

- ▶ Note this is defined for any matrix: so we can use it for
 - ▶ $\text{rank}(A)$ – the rank of the constraint coefficient matrix
 - ▶ $\text{rank}(M)$ – the rank of the augmented matrix $M = [A \mid \mathbf{b}]$
- ▶ they won't necessarily have the same rank, but

$$\text{rank}(A) \leq \text{rank}(M)$$

because when we add a column the rank either increases by 1, or stays the same.

³There are actually several equivalent definitions of rank.



Rank example

Example

$$A = \begin{bmatrix} 1 & 2 & 1 \\ -2 & -3 & 1 \\ 3 & 5 & 0 \end{bmatrix}$$

The $\text{rank}(A) = 2$

- ▶ $R_3 = R_1 - R_2$ so the 3 rows are linearly dependent
- ▶ any pair of two rows is linearly independent

Now add columns $M = [A | \mathbf{b}]$

$$M_1 = \left[\begin{array}{cccc} 1 & 2 & 1 & 2 \\ -2 & -3 & 1 & 4 \\ 3 & 5 & 0 & -1 \end{array} \right] \text{ and } M_2 = \left[\begin{array}{cccc} 1 & 2 & 1 & 2 \\ -2 & -3 & 1 & 3 \\ 3 & 5 & 0 & -1 \end{array} \right]$$

Then $\text{rank}(M_1) = 3$ and $\text{rank}(M_2) = 2$



Theorem (Rouché-Capelli)

A system of linear equations $Ax = \mathbf{b}$ has a solution iff the ranks of its coefficient matrix A and its augmented matrix $M = [A|\mathbf{b}]$ are equal.

If there are solutions, they form an affine subspace⁴ of \mathbb{R}^n of dimension $n - \text{rank}(A)$ where there are n variables.

Examples:

- ▶ $\text{rank}(A) = \text{rank}([A|\mathbf{b}]) = n$, there will be a unique solution
- ▶ $\text{rank}(A) = \text{rank}([A|\mathbf{b}]) < n$, infinite solutions
- ▶ $\text{rank}(A) \neq \text{rank}([A|\mathbf{b}])$, no solutions

⁴Here this is just a linear subspace translated away from the origin.



- ▶ If the matrix A is less than full rank, and the equations are consistent (there is at least one solution) we can reduce the set of equations down to a new set of full rank
 - ▶ some LP pre-processors do just this
- ▶ When we go from $A'\mathbf{x}' \leq \mathbf{b}$ to $A\mathbf{x} = \mathbf{b}$ we add m slack variables, and an identity matrix to A , i.e., $A = [A' \mid I_m]$. Even if all of the rows of A' were linearly dependent (i.e., $\text{rank}(A') = 1$), adding the identity would lead to $\text{rank}(A) = m$, i.e., A is full rank.
- ▶ For what follows, we shall usually assume we start with a full rank coefficient matrix A
 - ▶ if not, algorithms often still work, but may be inefficient

A circular graphic on the left side of the slide features a complex network of interconnected nodes. The nodes are represented by small circles in various colors, including white, red, and blue. They are connected by a web of thin lines of the same colors, forming a dense web of triangles and other polygons. The background is a dark blue gradient, with the network appearing more prominent against a lighter blue area at the top.

OPTIMISATION & OPERATIONS RESEARCH

1 Optimisation & linear programming

1.11 Naïve solution

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE



Take $A\mathbf{x} = \mathbf{b}$, where A is of size $m \times n$

- ▶ n variables
- ▶ m equations
- ▶ $n \leq m$

with full-rank A

- ▶ $\text{rank}(A) = \min(m, n) = n$
- ▶ $\text{rank}([A|\mathbf{b}]) = n$ or $n + 1$
 - ▶ could be 1 or 0 solutions
- ▶ 0 or 1 solutions doesn't leave much room for optimisation, so we will usually consider the case $n > m$
 - ▶ this always happens when we start with $A'\mathbf{x} \leq \mathbf{b}$, because we add m slack variables



Solutions

Take $A\mathbf{x} = \mathbf{b}$, where A is of size $m \times n$

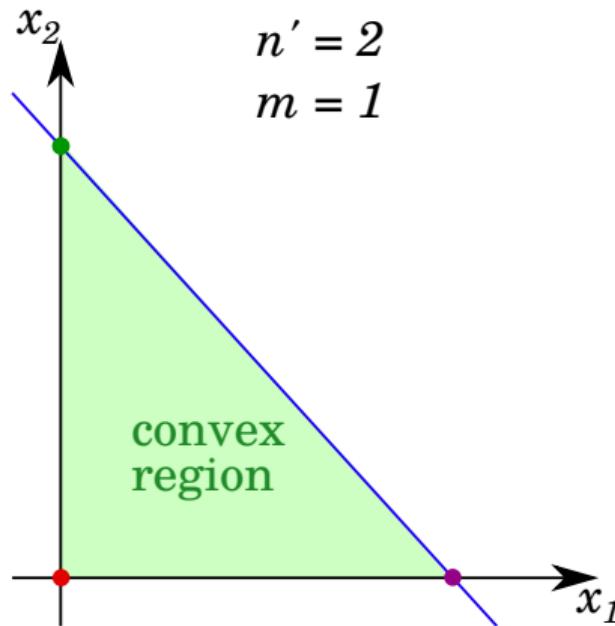
- ▶ n variables
- ▶ m equations

with full rank A , and $n > m$:

- ▶ The dimension of the solution subspace is $n - m$
 - ▶ the solution is a $n - m$ dimensional hyperplane
 - ▶ we can describe the hyperplane by where it intersects the axes (*i.e.*, places where $x_i = 0$ for some set of i)
- ▶ Choose $(n - m)$ variables to be 0
 - ▶ then there will be a unique solution for the other m variables
 - ▶ think of it as adding $n - m$ additional equations
 - ▶ or we reduce the number of variables down to m
 - ▶ the possible number of choices is

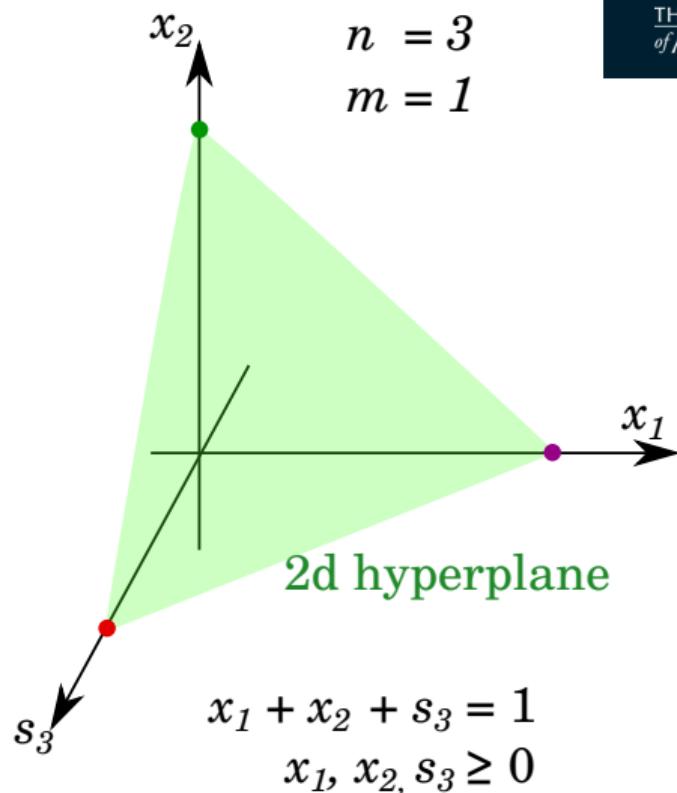
$$\binom{n}{n-m} = \binom{n}{m}$$

Numbers of solutions



$$\begin{aligned}x_1 + x_2 &\leq 1 \\x_1, x_2 &\geq 0\end{aligned}$$

3 vertices

 \Leftrightarrow $\binom{n}{m} = 3$ basic solutions

$$\begin{aligned}x_1 + x_2 + s_3 &= 1 \\x_1, x_2, s_3 &\geq 0\end{aligned}$$

$$\begin{aligned}n &= 3 \\m &= 1\end{aligned}$$

- ▶ The above suggests that we could perform optimisation via an exhaustive search
 - ▶ look for all the vertices
 - ▶ pick the best
- ▶ This would be very computationally challenging

Explore all possible vertices

1. construct all vertices:
 - 1.1 each construct requires at least one pivot: $O(nm)$ operations
 - 1.2 loose bound on basic solutions $\binom{n}{m}$
 - ▶ Stirling' approximation makes it $O(m^n)$
 - 1.3 check basic solution is feasible
 - ▶ requires comparison of m values
2. for each vertex you have to compute z which is $O(n)$, but this is small compared to performing the pivot.

So the total (worst-case) complexity is $O(nmm^n)$

This grows VERY VERY VERY quickly, e.g.,

- ▶ $n = 10, m = 10$, this would be around 10^{12}
- ▶ $n = 20, m = 20$, this would be around 10^{28}

- ▶ We think we can do MUCH better
Hirsch conjecture says that any two vertices of the polytope must be connected to each other by a path of length at most $m - n$.
- ▶ Maybe we could get the optimal value in $m - n$ steps (pivots), if we were really smart?
 - ▶ actually the conjecture is wrong, but doesn't rule out $O(m)$ performance

More on computational complexity later



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

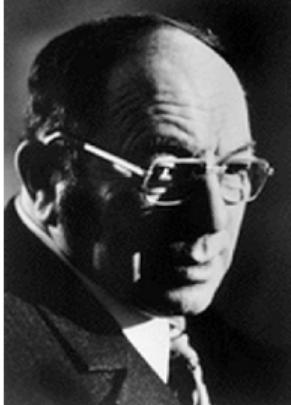
2.1 Simplex overview

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

A little history



Leonid Vitaliyevich Kantorovich; (19 January 1912 – 7 April 1986) was a Soviet mathematician and economist, known for his theory and development of techniques for the optimal allocation of resources. He was the winner of the Nobel Prize in Economics in 1975 and the only winner of this prize from the USSR.

George Bernard Dantzig:

(November 8, 1914 – May 13, 2005) was an American mathematical scientist who made important contributions to operations research, computer science, economics, and statistics. He is particularly well known for his development of the simplex algorithm and his work with linear programming, some years after it was pioneered by the Soviet mathematician and economist Leonid Kantorovich.



Problem Recap



THE UNIVERSITY
of ADELAIDE

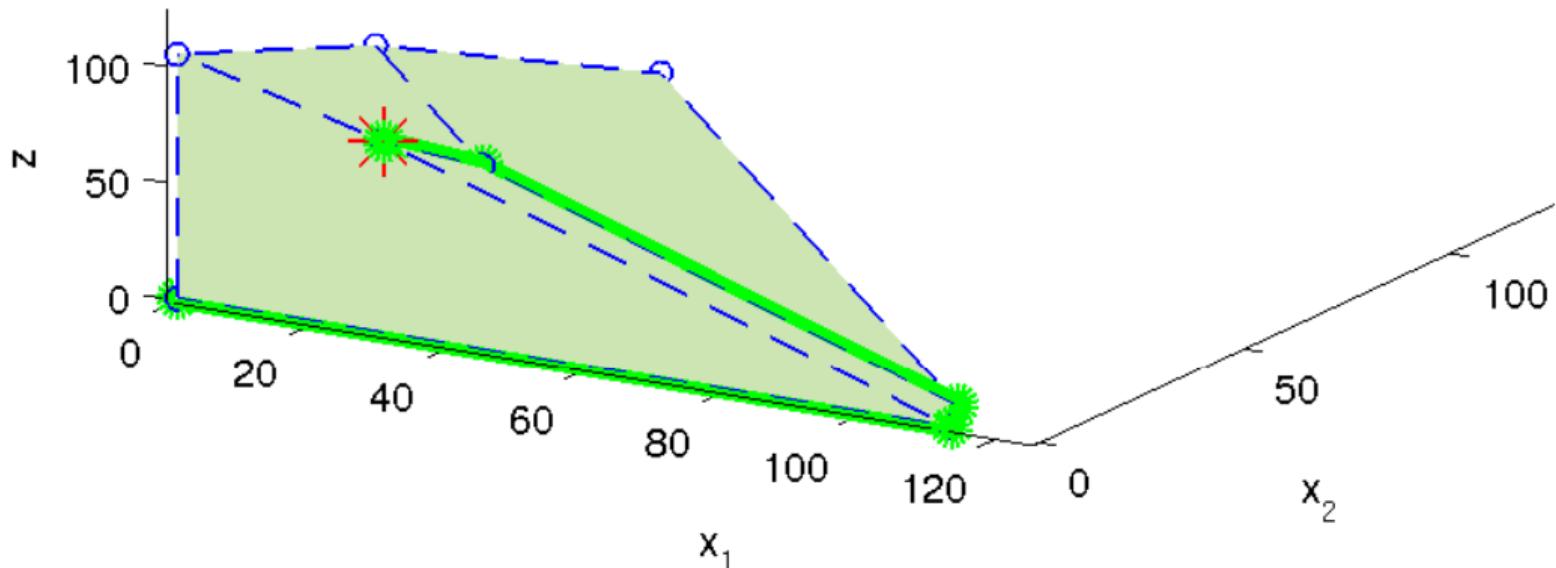
We will start with a problem in *standard equality form*, find \mathbf{x} that solves

$$\begin{aligned} \max \quad z &= \mathbf{c}^T \mathbf{x} + z_0 \\ \text{such that } A\mathbf{x} &= \mathbf{b} \\ \text{and } \mathbf{x} &\geq 0 \end{aligned}$$

or in abbreviated form

$$\underset{\mathbf{x}}{\operatorname{argmax}} \left\{ z = \mathbf{c}^T \mathbf{x} + z_0 \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0 \right\}$$

- ▶ Remember that often this has been converted from inequality form: so the vector x could include slack variables from inequalities.
- ▶ Intuitively, we want to search the vertices (efficiently) of the original feasible region for the optimal vertex.
- ▶ Vertices (of the original inequality problem) are feasible basic solutions to the equality
 - ▶ We will organise our search so as to force the constraints towards being true, and then staying true
 - ▶ Search always improves the objective



Our First Problem

$$\begin{array}{lll} \max & z = 13x_1 + 12x_2 + 17x_3 \\ \text{subject to} & 2x_1 + x_2 + 2x_3 \leq 225 \\ & x_1 + x_2 + x_3 \leq 117 \\ & 3x_1 + 3x_2 + 4x_3 \leq 420 \\ & x_i \geq 0, \quad \text{for } i = 1, 2, 3 \end{array}$$

1. Form the Simplex Tableau

- ▶ it's convenient to put all the information in one large matrix

2. Phase I

- ▶ look for a starting point
- ▶ either we get
 - ▶ a feasible starting point
 - ▶ we show the problem is infeasible

3. Phase II

- ▶ find the optimal point
- ▶ or show that the problem is unbounded

We teach this in the order

1. Tableau
2. Phase II
3. Phase I

because

- ▶ sometimes Simplex Phase I isn't needed
- ▶ phase I can be written in terms of Phase II

2 Simplex algorithm

2.2 The Simplex Tableau

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



The Simplex Tableau

The Simplex Tableau M for

$$\operatorname{argmax}_{\mathbf{x}} \left\{ z = \mathbf{c}^T \mathbf{x} + z_0 \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0 \right\}$$

is constructed by

	z col.	b col.
A	$\mathbf{0}$	\mathbf{b}
$-\mathbf{c}^T$	1	z_0

z row

- ▶ Assuming there are m constraints, and n variables, the Tableau M has size $(m + 1) \times (n + 2)$.
- ▶ Note the z column never changes, so we don't need to actually have this, but its useful to understand the algorithm.



The Simplex Tableau Example

Example

The LP from our first problem in standard equality form

$$\max z = 13x_1 + 12x_2 + 17x_3 + 0$$

subject to

$$2x_1 + x_2 + 2x_3 + 1x_4 = 225$$

$$x_1 + x_2 + x_3 + 1x_5 = 117$$

$$3x_1 + 3x_2 + 4x_3 + 1x_6 = 420$$

$$x_i \geq 0 \quad \forall i$$

can be written in Tableau form (with $z_0 = 0$) as

x_1	x_2	x_3	x_4	x_5	x_6	z	b
2	1	2	1	0	0	0	225
1	1	1	0	1	0	0	117
3	3	4	0	0	1	0	420
-13	-12	-17	0	0	0	1	0

Why use the Simplex Tableau?

The Simplex Tableau isn't really a necessary part of the algorithm

- ▶ e.g., sometimes people put the various bits in different parts of the Tableaux – it doesn't matter

So why do it?

- ▶ The Simplex Tableau puts all the data in one place
 - ▶ useful back in the days of crappy memory management
- ▶ Does it in a way that means we can use a single operation
 - ▶ pivot does the bulk of the work
 - ▶ makes the code simpler
- ▶ It's useful for teaching
 - ▶ we can understand what the algorithm is doing in a very simple way



The Simplex Tableau and solutions

Imagine the Simplex Tableau

$$M = \begin{array}{|c|c|c|} \hline & A & \mathbf{0} & \mathbf{b} \\ \hline -\mathbf{c}^T & 1 & z_0 \\ \hline \end{array}$$

can be (re)written in the form

$$M' = \begin{array}{|c|c|c|} \hline Q & I & \mathbf{b}' \\ \hline \end{array}$$

Then we can read off the solution immediately:

- ▶ set the free variables $x_i = 0$ for $i = 1, \dots, n$
- ▶ then just read off the basic variables $x_{n+i} = b'_i$ for $i = 1, \dots, m$

We want to generalise and exploit this idea



The Simplex Tableau and Solutions

Example

x_1	x_2	x_3	x_4	x_5	x_6	z	b
2	1	2	1	0	0	0	225
1	1	1	0	1	0	0	117
3	3	4	0	0	1	0	420
-13	-12	-17	0	0	0	1	0

Solution:

$$\mathbf{x} = (0, 0, 0, 225, 117, 420) \text{ and } z = 0$$

We want to generalise and exploit this idea:

- ▶ we are looking for *unit* columns, i.e., columns with m entries zero and the remaining element is 1.
- ▶ we need a complete set of these unit columns



Basic columns

- ▶ If x_j is a basic variable, then the j th column $M(:,j)$ will be a unit column which we call a *basic column*
- ▶ A solution should have m basic variables, so we need m basic columns, plus 1 (the z column).
- ▶ The basic columns need to have their '1's in different rows, or we don't really have independent basic variables.

Let's construct a vector of the indexes of the basic variables:

$$\ell_B = [\ell_1, \ell_2, \dots, \ell_m]$$

then we want to get M in the form such that when we take the columns corresponding to these variables, we get an identity matrix:

$$\begin{bmatrix} M_{\ell_1} & M_{\ell_2} & \dots & M_{\ell_m} & \textcolor{green}{M_z} \end{bmatrix} = I_{m+1}$$

where I_{m+1} is the $(m + 1) \times (m + 1)$ identity matrix.

Canonical form



THE UNIVERSITY
of ADELAIDE

Definition (Canonical form)

The tableau M is said to be in *canonical form* if there is a complete set of $(m+1)$ unit columns:

$$\begin{bmatrix} M_{\ell_1} & M_{\ell_2} & \dots & M_{\ell_m} & M_z \end{bmatrix} = I_{m+1}$$

and $\ell_B = [\ell_1, \ell_2, \dots, \ell_m]$ is the canonical form of the list of basic variables.

Example

Note that the basic variables don't have to be in order

$$M = \begin{array}{|c|c|c|c|c|c|c|} \hline x_1 & x_2 & x_3 & x_4 & x_5 & z & b \\ \hline 0 & 1 & 0 & 6 & 1 & 0 & 2 \\ \hline 2 & -3 & 1 & 1 & 0 & 0 & 5 \\ \hline 0 & -1 & 0 & 0 & 0 & 1 & 4 \\ \hline \end{array}$$

is in canonical form, with $\ell_B = [5, 3]$



Example

$$M = \begin{array}{|c|c|c|c|c|} \hline & x_1 & x_2 & x_3 & z & b \\ \hline 1 & 1 & 2 & 0 & 0 & 2 \\ \hline 0 & 0 & 3 & 0 & 0 & 3 \\ \hline 0 & 0 & 0 & 1 & 0 & 4 \\ \hline 0 & 0 & 4 & 0 & 1 & 4 \\ \hline \end{array}$$

is **not** in canonical form, because there is no unit column with 1 in the second row.



Definition (Feasible Canonical form)

A tableau M in canonical form is said to be in *feasible canonical form* iff its solution is feasible.

- ▶ We can detect a feasible canonical form by noting that the solution where we read off the x_i values as above will have x_i values that are either 0 or b_j . These automatically satisfy $Ax = b$, but must also satisfy $x \geq 0$, so
*a canonical form is feasible iff the b column is non-negative!*⁵
- ▶ A feasible canonical form corresponds to a vertex of the original inequality feasible region.

⁵Ignoring the z_0 element



Getting to feasible canonical form

- ▶ Simplex Phase II starts with a matrix in feasible canonical form
 - ▶ how do we get there?
- ▶ If we start with the problem $\max_{\mathbf{x}} \{z = \mathbf{c}^T \mathbf{x} + z_0 \mid A' \mathbf{x}' \leq \mathbf{b}, \mathbf{x}' \geq 0\}$, with $\mathbf{b} \geq 0$, then conversion to standard equality form will introduce slack variables that create the required identity matrix.

Example

x_1	x_2	x_3	s_4	s_5	s_6	z	b
2	1	2	1	0	0	0	225
1	1	1	0	1	0	0	117
3	3	4	0	0	1	0	420
-13	-12	-17	0	0	0	1	0

- ▶ this canonical form corresponds to a starting point $\mathbf{x}' = \underline{0}$ (with positive slack variables), i.e., the origin
- ▶ Otherwise we use Simplex Phase I to get into feasible canonical form before we perform Phase II.

- ▶ The Tableau M encapsulates all of the information we need to keep track of
 - ▶ we need to know how to construct it
- ▶ The form we aim for or want to keep is *feasible canonical form*
 - ▶ complete set of unit columns
 - ▶ **b** column is non-negative
- ▶ In this form we can read off a basic feasible solution



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

2.3 Simplex Phase II

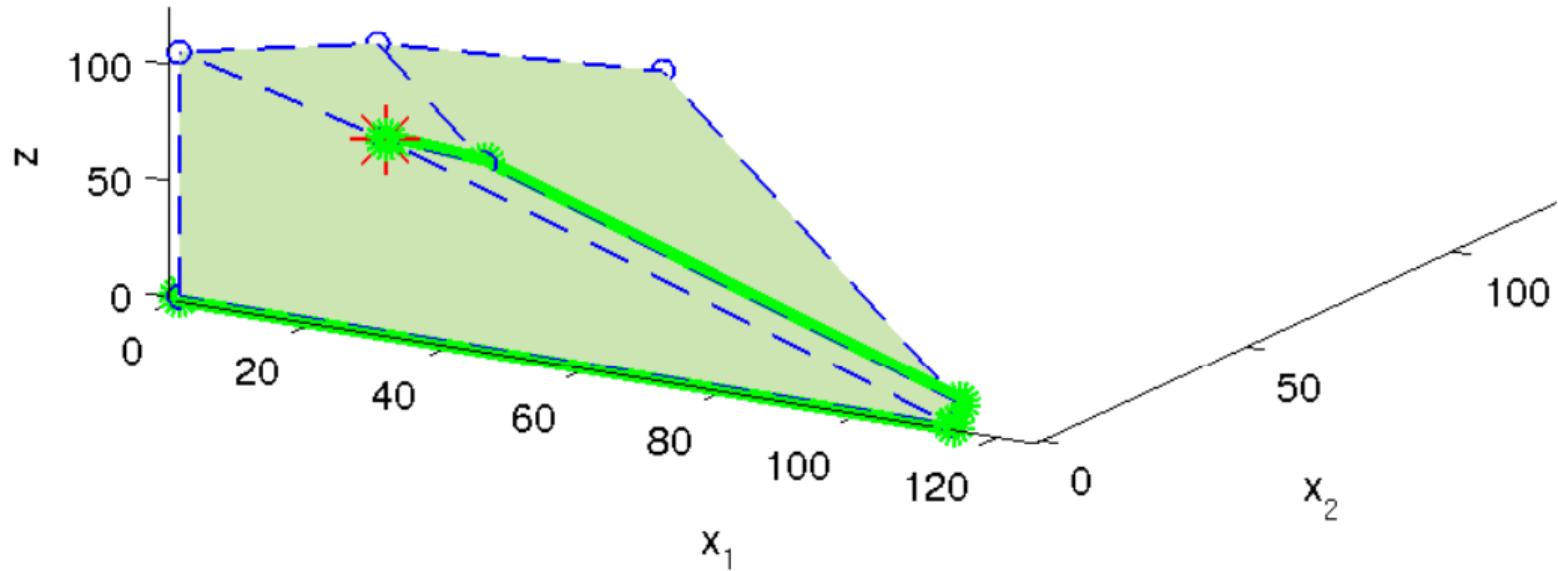
Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



- ▶ Assume we start in feasible, canonical form
 - ▶ Corresponds to a feasible vertex in original inequality form
 - ▶ Often starts this way
 - ▶ Simplex Phase I will fix this if it isn't already
- ▶ Test to see if it is optimal (if so then stop)
- ▶ Move to an *adjacent* vertex such that
 - ▶ the solution remains feasible
 - ▶ the solution gets better (or at least no worse)
- ▶ And repeat until its optimal, or we can't go any further



$$\begin{aligned} \max \quad & z = 13x_1 + 12x_2 + 17x_3 \\ \text{subject to} \quad & 2x_1 + x_2 + 2x_3 \leq 225 \\ & x_1 + x_2 + x_3 \leq 117 \\ & 3x_1 + 3x_2 + 4x_3 \leq 420 \\ & x_i \geq 0, \quad \text{for } i = 1, 2, 3 \end{aligned}$$



Operating on a tableau

Imagine the Simplex Tableau

$$M = \begin{array}{|c|c|c|} \hline & A & \mathbf{0} & \mathbf{b} \\ \hline -\mathbf{c}^T & 1 & z_0 \\ \hline \end{array}$$

If we perform the same row operations we were allowed in Gauss-Jordan,

$$M \xrightarrow{\text{rowops}} M'$$

then M' will be an *equivalent tableau*

- ▶ it expresses exactly the same LP
- ▶ for exactly the same reasons that G-J works



- ▶ We can perform the same row operations we were allowed in Gauss-Jordan.
- ▶ We want to change vertices:
 - ▶ change which constraints are active (in inequality form)
 - ▶ change which variables are basic (in equality form)
- ▶ We do this with *pivot* operations on (i, j) where
 - ▶ i means the i th row of A
 - ▶ j means the j th column of A

We don't pivot in the z column because it represents the objective.

One pivot (plus associated bits) is one *iteration* of Simplex

$$M^{(i)} \rightarrow M^{(i+1)}$$

When we perform a pivot, one variable *leaves* the set of basic variables, and another *enters* the basic set.

Definition (Entering Variable)

The entering variable is the variable that enters the basic set after a pivot operation. It corresponds to the column j of the pivot.

Definition (Leaving Variable)

The leaving variable is the variable that leaves the basic set after the pivot operation. The row i of the pivot determines the leaving variable, by selecting constraint i , and the current basic variable k associated with it.

Rules for Choosing Entering and Leaving Variables

There are different rules for choosing the entering and leaving variables, but these rules have two components:

- ▶ *mandatory*: all choices must satisfy these in order for Simplex to work
- ▶ *discretionary*: choices can pick and choose between these.

They matter, because may affect the performance of the overall algorithm, and sometimes will have a big impact, but there are alternatives.

Choosing the Entering Variable (the column)

- *Mandatory:* the choice of column j must satisfy

$$-c_j < 0,$$

i.e., the current j th value in the z -row must be negative.

- *Discretionary:* when there is a choice between multiple possible negative values, we can implement several different approaches.
 - e.g., choose the most negative

I will tell you the mandatory rules, but I *want you to work out what discretionary rules makes the most sense.*

Choosing the Entering Variable: Example

Example

						pivot column	
x_1	x_2	x_3	x_4	x_5	x_6	z	b
1	3	-2	1	0	0	0	3
-1	2	1	0	1	0	0	1
0	3	1	0	0	1	0	2
2	1	-2	0	0	0	1	-2

We choose $j = 3$ because this is the only A -column with $-c_j < 0$

Choosing the Leaving Variable (pivot row)

- ▶ *Mandatory:* given column j , the choice of row i must
 - ▶ $a_{ij} > 0$
 - ▶ of the non-negative possibilities, choose smallest possible b_i/a_{ij}
- ▶ *Discretionary:* when there is more than one equal value of b_i/a_{ij} we get to choose one.

I want you to work out what discretionary rule makes the most sense.



Choosing the Leaving Variable: Example

Example

x_1	x_2	x_3	x_4	x_5	x_6	z	b
1	3	-2	1	0	0	0	3
-1	2	1	0	1	0	0	1
0	3	1	0	0	1	0	2
2	1	-2	0	0	0	1	-2

$a_{ij} < 0$
 $b_i/a_{ij} = 1$ pivot row
 $b_i/a_{ij} = 2$

We already chose $j = 3$, so now examine possible rows,

- ▶ we can eliminate row 1, because $a_{13} \leq 0$
- ▶ we have

$$b_2/a_{23} = 1$$

$$b_3/a_{33} = 2$$

we choose the smallest, so $i = 2$

- ▶ $\ell_B = \{4, 5, 6\}$, and the i th gives us the leaving variable $x_{\ell_2} = x_5$

- ▶ If you can't satisfy a mandatory rule, then Simplex stops
- ▶ Output state:
 - ▶ if you can't find a new entering variable, *i.e.*, no column has $-c_j < 0$ then you have the optimal value
 - ▶ if you can't find a new leaving variable, then the problem is *unbounded*
- ▶ Why do the choice rules work?
 - ▶ we have two imperatives
 1. stay feasible
 2. improve the objective function (or at least make it no worse)
 - ▶ the mandatory rules are aimed at these imperatives

2 Simplex algorithm

2.4 Simplex Phase II (continued)

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



Why Have Mandatory Choice Rules: column

Why do we require $-c_j < 0$ for the pivot column j ?

- ▶ Simplex keeps track of z -value in z_0 (bottom right corner)
- ▶ When we pivot at (i,j)

$$z'_0 = z_0 + \frac{c_j}{a_{ij}} b_i$$

- ▶ Feasible canonical form has $b_i \geq 0$
- ▶ We choose a pivot so $a_{ij} > 0$
- ▶ If $-c_j < 0$, then the above means that

$$z'_0 \geq z_0$$

- ▶ This rule ensures that the *objective is non-decreasing*
 - ▶ it only stays the same if $b_i = 0$
- ▶ We stop when we can't find such a column, because the objective can't be increased any more, *i.e.*, we have found the optimal solution



Why Have Mandatory Choice Rules: row

Why do we require $a_{ij} > 0$ and smallest possible b_i/a_{ij} for row i ?

- ▶ (1) so objective remains non-decreasing
- ▶ (2) when we pivot at (i, j) the new **b** column is

$$b'_k = b_k - \frac{a_{kj}}{a_{ij}} b_i, \quad \text{for } k \neq i$$

for **feasible** canonical form $b_k \geq 0$, and we choose $a_{ij} > 0$ so

- ▶ if $a_{kj} \leq 0$ then $b'_k \geq b_k \geq 0$
- ▶ if $a_{kj} > 0$ for all rows $k \neq i$

$$b'_k = a_{kj} \left(\frac{b_k}{a_{kj}} - \frac{b_i}{a_{ij}} \right) \geq 0$$

so $b'_k \geq 0$ we choose i such that for all rows $k \neq i$

$$\frac{b_k}{a_{kj}} \geq \frac{b_i}{a_{ij}}$$

- ▶ So this rule maintains **feasibility**

Why Have Mandatory Choice Rules: row

What if we can't find a row?

- ▶ That means all of $a_{ij} \leq 0$.
- ▶ Consider the following solution
 - ▶ set entering variable $x_j = t > 0$
 - ▶ set other free variables to 0
 - ▶ solve for the basic variables ℓ_B

$$x_{\ell_i} = b_i - a_{ij}t$$

- ▶ normally, we would have had $t = 0$, and $x_{\ell_i} = b_i$
- ▶ note $x_{\ell_i} \geq 0$ as $b_i \geq 0$, $t > 0$ and $a_{ij} < 0$
- ▶ so this solution is feasible for any value $t > 0$
- ▶ Hence we have an *arbitrarily large* feasible solution
- ▶ The problem is *unbounded*

I want you to find the *discretionary* part of your choice rules.

- ▶ research them – you may want to Google “Bland’s rule”
- ▶ you need to write a Simplex program using functions so that you can change choice rules easily and experiment (some class exercises may require you to be able to try different choice rules).

You will be examined on choice rules, so this part of your work is not optional!!!

Finally, if we found a feasible, optimal value, just read off the solution

1. All of the non-basic variables are zero.
2. For the basic variables in the list ℓ_B , just read off the values

$$x_{\ell_B(i)}^* \leftarrow b_i$$

3. z^* , the optimal value of the objective function, is given by the bottom value of the **b** column (where you originally placed z_0)



Simplex Phase II pseudo-code

Assumes a problem in standard equality form: $\max_{\mathbf{x}} \{z = \mathbf{c}^T \mathbf{x} + z_0 \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \underline{0}\}$
such that when we construct the Tableau M it will be in feasible canonical form

```
Input:  $A, \mathbf{b}, \mathbf{c}, z_0, \ell_B$  // implicitly n variables, m constraints
Output:  $\mathbf{x}^*, z^*$ 

1 Construct Simplex Tableau  $M$  // an  $(m+1) \times (n+2)$  matrix
2 while at least one  $-c_j < 0$  do
3   Select Entering Variable  $x_j$  (column  $j$ )
4   Select Leaving Variable  $x_k$  (row  $i$ )
5   if there is no possible leaving variable then
6     | break // problem is unbounded
7   else
8     |  $\ell_B(i) \leftarrow j$ 
9     |  $M \leftarrow \text{Pivot}(M, i, j)$ 
10  end
11 end
12 Set  $\mathbf{x}^* = \underline{0}$ 
13 for  $i = 1 \dots m$  do
14   |  $x_{\ell_B(i)}^* \leftarrow b_i = M(i, \text{b-col})$ 
15 end
16  $z^* \leftarrow M(\text{z-row}, \text{b-col})$ 
```

Algorithm 1: Simplex Phase II



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

2.5 Simplex Phase II example

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



Simplex Phase II Example: iteration 1

Starts at $\mathbf{x}_0 = (0, 0, 0, 225, 117, 420)$, $z_0 = 0$

x_1	x_2	x_3	s_4	s_5	s_6	z	b
2	1	2	1	0	0	0	225
1	1	1	0	1	0	0	117
3	3	4	0	0	1	0	420
-13	-12	-17	0	0	0	1	0



1	1/2	1	1/2	0	0	0	225/2
0	1/2	0	-1/2	1	0	0	9/2
0	3/2	1	-3/2	0	1	0	165/2
0	-11/2	-4	13/2	0	0	1	2925/2

After the pivot $\mathbf{x}_1 = (225/2, 0, 0, 0, 9/2, 165/2)$, $z_1 = 1462.5$



Simplex Phase II Example: iteration 2

Starts at $\mathbf{x}_1 = (225/2, 0, 0, 0, 9/2, 165/2)$, $z_1 = 1462.5$

x_1	x_2	x_3	s_4	s_5	s_6	z	b
1	1/2	1	1/2	0	0	0	225/2
0	1/2	0	-1/2	1	0	0	9/2
0	3/2	1	-3/2	0	1	0	165/2
0	-11/2	-4	13/2	0	0	1	2925/2



1	0	1	1	-1	0	0	108
0	1	0	-1	2	0	0	9
0	0	1	0	-3	1	0	69
0	0	-4	1	11	0	1	1512

After the pivot $\mathbf{x}_2 = (108, 9, 0, 0, 0, 69)$, $z_2 = 1512$



Simplex Phase II Example: iteration 3

Starts at $\mathbf{x}_2 = (108, 9, 0, 0, 0, 0, 69)$, $z_2 = 1512$

x_1	x_2	x_3	s_4	s_5	s_6	z	b
1	0	1	1	-1	0	0	108
0	1	0	-1	2	0	0	9
0	0	1	0	-3	1	0	69
0	0	-4	1	11	0	1	1512



1	0	0	1	2	-1	0	39
0	1	0	-1	2	0	0	9
0	0	1	0	-3	1	0	69
0	0	0	1	-1	4	1	1788

After the pivot $\mathbf{x}_3 = (39, 9, 69, 0, 0, 0)$, $z_3 = 1788$



Simplex Phase II Example: iteration 4

Starts at $x_3 = (39, 9, 69, 0, 0, 0)$, $z_3 = 1788$

x_1	x_2	x_3	s_4	s_5	s_6	z	b
1	0	0	1	2	-1	0	39
0	1	0	-1	2	0	0	9
0	0	1	0	-3	1	0	69
0	0	0	1	-1	4	1	1788

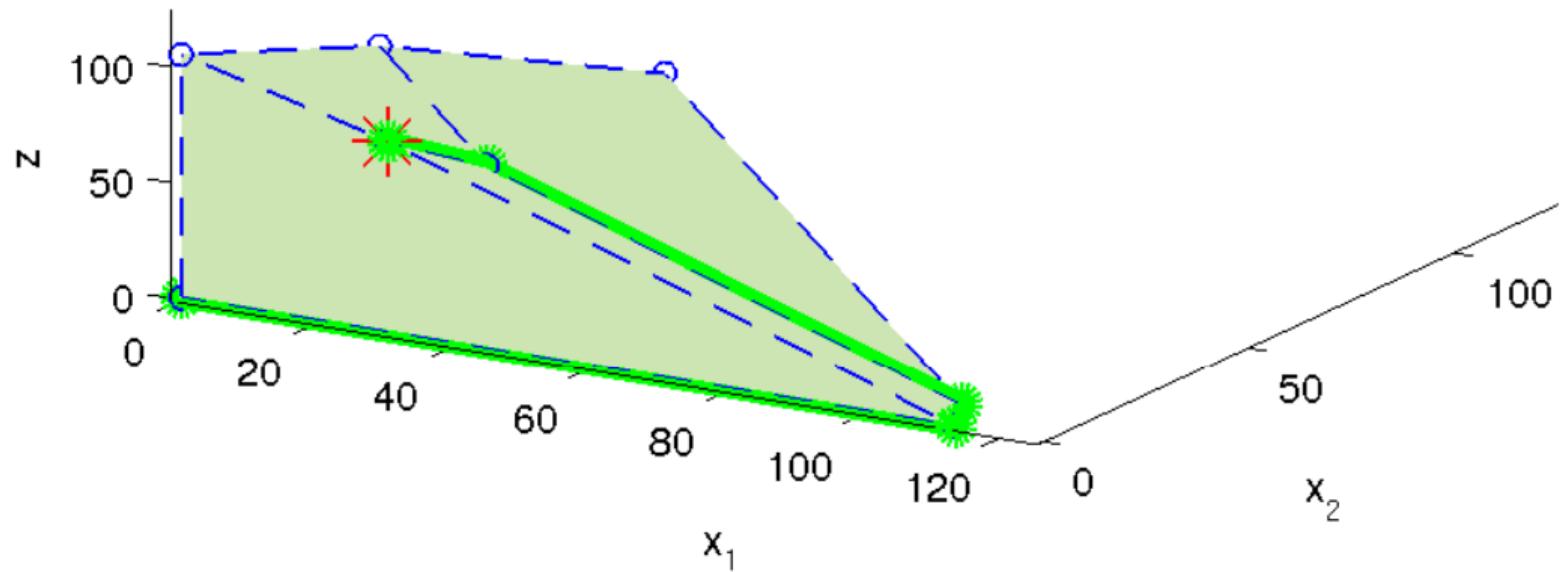


1	-1	0	2	0	-1	0	30
0	1/2	0	-1/2	1	0	0	9/2
0	3/2	1	-3/2	0	1	0	165/2
0	1/2	0	1/2	0	4	1	3585/2

After the pivot $x_4 = (30, 0, 165/2, 0, 9/2, 0)$, $z_4 = 1792.5$

- ▶ there are no more possible pivot columns, so we stop, and this is the optimal solution

Sim



- ▶ Simplex Phase II
 - ▶ starts in feasible canonical form
 - ▶ operates by choosing pivots (rules)
 - ▶ each pivot
 - ▶ keeps it feasible
 - ▶ makes objective no worse
 - ▶ ends
 - ▶ unbounded
 - ▶ optimal solution



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

2.6 Simplex Phase I

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

Problem Recap



We will start with a problem in *standard equality form*.

$$\operatorname{argmax}_{\mathbf{x}} \left\{ z = \mathbf{c}^T \mathbf{x} + z_0 \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0 \right\}$$

or in longer form, find \mathbf{x} that solves

$$\begin{aligned} \max \quad & z = \mathbf{c}^T \mathbf{x} + z_0 \\ \text{such that } & A\mathbf{x} = \mathbf{b} \\ \text{and } & \mathbf{x} \geq 0 \end{aligned}$$

However, the problem might not start in feasible canonical form:

- ▶ could be **infeasible**, e.g., $b_i < 0$
- ▶ could be "**uncanonical**," e.g., lacks one or more unit columns



The Simplex Tableau for Phase I Example

Example (uncanonical and infeasible standard equality form)

$$\begin{array}{rclclcl} \text{max } z & = & 2x_2 & + & x_3 & + & 6x_4 & + & 2 \\ \text{such that} & & x_1 & - & x_2 & - & 2x_4 & = & 0 \\ & & x_1 & + & x_2 & + & 2x_3 & = & 4 \text{ becomes} \\ & & & & x_2 & - & x_4 & - & x_5 = -2 \\ & & & & & & x_i \geq 0, \forall i \end{array}$$

the Tableau

x_1	x_2	x_3	x_4	x_5	z	b
1	-1	0	-2	0	0	0
1	1	2	0	0	0	4
0	1	0	-1	-1	0	-2
0	-2	-1	-6	0	1	2

z row

There is only one basic column (not a full set) and one of the $b_i < 0$

- ▶ Construct an *artificial problem*, which we specifically create so that
 - ▶ it has a *guaranteed* feasible starting point
 - ▶ maximising its objective *creates a feasible solution* to the original problem
- ▶ Use Simplex Phase II on the artificial problem
 - Each step (roughly)
 - ▶ adds a basic variable to the original problem
 - ▶ moves towards a feasible solution
- ▶ When we stop, we know we will either
 - ▶ have a feasible canonical tableau
 - ▶ have proved that the original problem was infeasible



- ▶ *Relaxation* means giving up on one or more constraints
- ▶ Here we want to solve for $Ax = b$, but we could relax this to the constraint $Ax \leq b$.
- ▶ The idea is that
 - ▶ we know how to put this in a suitable form to solve: just add *artificial* slack variables a_i to get

$$Ax + a = b$$

- ▶ artificial slack variables measure how far away we are from feasibility

Creating an Artificial Problem via Relaxation

1. Multiply any row with a $b_i < 0$ by -1
2. Relax $A\mathbf{x} = \mathbf{b}$ $\rightarrow A\mathbf{x} \leq \mathbf{b}$.
3. Add in *artificial* slack variables a_i to get

$$A\mathbf{x} + \mathbf{a} = \mathbf{b}$$

The relaxed problem is *guaranteed* to be in feasible canonical form

- ▶ it is feasible, because we ensured that all $b_i \geq 0$ before we started
- ▶ it is canonical because we can write it as

$$B\mathbf{y} = \mathbf{b}$$

where $B = [A \mid I_m]$ and $\mathbf{y}^T = (\mathbf{x}^T, \mathbf{a}^T)$.

We still need an artificial objective function.

How do we measure infeasibility?

We don't (for the moment) care about the original objective, we instead create an artificial objective to pull us back to the original equations

- ▶ the artificial variables a_i measure how far we are from feasible
- ▶ maximise new objective

$$u = - \sum_{i=1}^m a_i = \sum_{i=1}^m [A\mathbf{x} - \mathbf{b}]_i$$

- ▶ we created the a_i such that $a_i \geq 0$, so $u \leq 0$
- ▶ when $u = 0$ we have $A\mathbf{x} = \mathbf{b}$, i.e., feasibility
- ▶ So we want to maximise u , with the aim of getting 0



How do we build this objective into the problem?

$$\begin{aligned} u &= \sum_{i=1}^m \left(\sum_{j=1}^n a_{ij}x_j - b_i \right) \\ &= \sum_{i=1}^m \sum_{j=1}^n a_{ij}x_j - \sum_{i=1}^m b_i \\ &= \sum_{j=1}^n x_j \sum_{i=1}^m a_{ij} - \sum_{i=1}^m b_i \\ &= \sum_{j=1}^n x_j d_j + u_0 \end{aligned}$$

where we define

$$d_j = \sum_{i=1}^m a_{ij} \quad \text{and} \quad u_0 = -\sum_{i=1}^m b_i$$

So

- ▶ We multiply any constraint row with $b_i < 0$ by -1
- ▶ We introduce artificial variables a_i so that the system is in feasible canonical form
- ▶ We maximise $u = \sum_{i=1}^n x_j d_j + u_0$
- ▶ We also note that

$$z = \mathbf{c}^T \mathbf{x} + z_0$$

which we can rewrite as a constraint

$$-\mathbf{c}^T \mathbf{x} + z = z_0$$

where z looks like another variable

- ▶ we don't know the optimal value of z yet, its just another variable for the purpose of this optimisation.



Now we have a new LP

$$\begin{array}{ll} \max & u = \mathbf{d}^T \mathbf{x} + u_0 \\ \text{such that} & B\mathbf{y} = \mathbf{b} \\ & -\mathbf{c}^T \mathbf{x} + z = z_0 \\ \text{and} & \mathbf{y} \geq 0 \end{array}$$

- ▶ This is a standard equality form LP with $n+1$ variables and $m+1$ constraints
 - ▶ it starts in feasible canonical form
 - ▶ so we can solve it using the Simplex Phase II algorithm
- ▶ But there are some differences
 1. we don't pivot in the z row (see below)
 2. the final tableau might have some elements with $b_i = 0$, on which we perform a special step
 3. this problem is guaranteed to be bounded, by construction
- ▶ Once we finish, remove the u row, and continue with Phase II



The Simplex Tableau for Phase I

The Simplex Phase I Tableau M for

$$\underset{\mathbf{x}}{\operatorname{argmax}} \left\{ \mathbf{u} = \mathbf{d}^T \mathbf{x} + u_0 \mid A\mathbf{x} = \mathbf{b}, -\mathbf{c}^T \mathbf{x} + z = z_0, \mathbf{x} \geq 0 \right\}$$

is

a_i 's	z col.	\mathbf{u} col.	\mathbf{b} col.	
A	I	$\mathbf{0}$	$\mathbf{0}$	\mathbf{b}
$-\mathbf{c}^T$	$\mathbf{0}$	1	0	z_0
$-\mathbf{d}^T$	$\mathbf{0}$	0	1	u_0

z row

u row

- ▶ Pre-multiply any row with a negative b_i by -1 before construction
- ▶ $d_j = \sum_{i=1}^m a_{ij}$
- ▶ Start with artificial variables as basic
- ▶ We don't really need to keep track of the artificial variables because they will be zero in the end anyway



The Simplex Tableau for Phase I Example

Example

max $z = 2x_2 + x_3 + 6x_4 + 2$

such that

$$\begin{array}{rclclclclclcl} x_1 & - & x_2 & & & & - & 2x_4 & = & 0 \\ x_1 & + & x_2 & + & 2x_3 & & & & = & 4 \text{ becomes} \\ & & x_2 & & & - & x_4 & - & x_5 & = & -2 \\ & & & & & & & & x_i & \geq & 0, \forall i \end{array}$$

the Tableau

x_1	x_2	x_3	x_4	x_5	a_1	a_2	a_3	z	u	b
1	-1	0	-2	0	1	0	0	0	0	0
1	1	2	0	0	0	1	0	0	0	4
0	-1	0	1	1	0	0	1	0	0	2
0	-2	-1	-6	0	0	0	0	1	0	2
-2	1	-2	1	-1	0	0	0	0	1	-6

$\times -1$
 z row
 u row

In calculations, we can drop the artificial a_i and u columns (as we could with the z column earlier).

The Simplex Phase I Pivot Selection

- ▶ Simplex Phase I then proceeds using the Phase II algorithm
 - ▶ series of pivots
- ▶ However, pivot selection is slightly different
 - ▶ don't pivot in the z -row
 - ▶ we might use different discretionary rules

The Simplex Phase I Termination



THE UNIVERSITY
of ADELAIDE

- ▶ Simplex Phase I then proceeds using the Phase II algorithm
 - ▶ series of pivots
 - ▶ terminates when we can't find a new pivot
- ▶ The artificial problem
 - ▶ is guaranteed to be feasible (we set it up that way)
 - ▶ is guaranteed to be bounded because we know $u \leq 0$

Hence we will find an optimal solution u^* to the artificial problem

- ▶ There are two cases
 - ▶ $u^* < 0 \Rightarrow$ the original problem is infeasible
 - ▶ $u^* = 0 \Rightarrow$ the original problem is feasible, and the solution to the artificial problem (minus the artificial variables) is a feasible starting point for the original problem
- ▶ It's possible we won't have all the required unit columns
 - ▶ we can multiply rows with $b_i = 0$ by -1 to get them



Simplex Phase I pseudo-code

Problem in standard equality form: $\max_{\mathbf{x}} \{ z = \mathbf{c}^T \mathbf{x} + z_0 \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0 \}$

```
Input:  $A, \mathbf{b}, \mathbf{c}, z_0$                                 // implicitly n variables, m constraints
Output:  $\mathbf{x}^*, z^*$ 
1 Construct Simplex Phase I Tableau  $M$ 
2 while at least one  $-d_i < 0$  do
3   Select Entering Variable  $x_j$  (column  $j$ )
4   Select Leaving Variable  $x_k$  (row  $i$ )
5    $\ell_B(i) \leftarrow j$ 
6    $M \leftarrow \text{Pivot}(M, i, j)$ 
7 end
8  $u^* \leftarrow M(u\text{-row}, b\text{-col})$ 
9 if  $u^* < 0$  then
10  problem is infeasible
11 else
12  remove any all-zero rows
13  for row  $i$  without a basic column do
14    |  $M \leftarrow \text{Pivot}(M, i, j)$ , s.t.  $a_{ij} \neq 0$           //  $b_i = 0$  for such rows
15  end
16  remove  $u$ -row and col, and artificial variables
17  continue to Simplex Phase II
18 end
```

Algorithm 2: Simplex Phase I



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

2.7 Simplex Phase I example

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



Simplex Method Phase I Example: iteration 1

Example

x_1	x_2	x_3	x_4	x_5	z	u	b
1	-1	0	-2	0	0	0	0
1	1	2	0	0	0	0	4
0	-1	0	1	1	0	0	2
0	-2	-1	-6	0	1	0	2
-2	1	-2	1	-1	0	1	-6



z row
 u row

1	-1	0	-2	0	0	0	0
0	2	2	2	0	0	0	4
0	-1	0	1	1	0	0	2
0	-2	-1	-6	0	1	0	2
0	-1	-2	-3	-1	0	1	-6

z row
 u row



Simplex Method Phase I Example: iteration 2

Example

x_1	x_2	x_3	x_4	x_5	z	u	b
1	-1	0	-2	0	0	0	0
0	2	2	2	0	0	0	4
0	-1	0	1	1	0	0	2
0	-2	-1	-6	0	1	0	2
0	-1	-2	-3	-1	0	1	-6

 z row
 u row

1	0	1	-1	0	0	0	2
0	1	1	1	0	0	0	2
0	0	1	2	1	0	0	4
0	0	1	-4	0	1	0	6
0	0	-1	-2	-1	0	1	-4

 z row
 u row

Simplex Method Phase I Example: iteration 3

Example

1	0	1	-1	0	0	0	2
0	1	1	1	0	0	0	2
0	0	1	2	1	0	0	4
0	0	1	-4	0	1	0	6
0	0	-1	-2	-1	0	1	-4

z row

u row



1	0	3/2	0	1/2	0	0	4
0	1	1/2	0	-1/2	0	0	0
0	0	1/2	1	1/2	0	0	2
0	0	3	0	2	1	0	14
0	0	0	0	0	0	1	0

z row

u row



Simplex Method Phase I Example: stop

Example

x_1	x_2	x_3	x_4	x_5	z	u	b
1	0	3/2	0	1/2	0	0	4
0	1	1/2	0	-1/2	0	0	0
0	0	1/2	1	1/2	0	0	2
0	0	3	0	2	1	0	14
0	0	0	0	0	0	1	0

z row
 u row

- ▶ There are no more places with $-d_j < 0$, so Simplex Phase I stops
- ▶ The Tableau is in feasible canonical form

$$\ell_B = (1, 2, 4)$$

- ▶ Also $u^* = 0$, so we have found a feasible starting point for Phase II
- ▶ Drop the u -row and proceed to Phase II (but that will finish straight away because the resulting Tableau is already finished)

How does Phase I guarantee to get us to a feasible canonical form?

- ▶ We never pivot on the columns corresponding to the a_i ;
 - ▶ in fact, we can leave these columns out of calculations
 - ▶ so these are never entering variables
- ▶ Take R to be the set of rows without a corresponding basic variable (outside the artificial variables)
 - ▶ Start with all rows (except u and z row) in R .
 - ▶ If we ever pivot in row i , this row is removed from R and never returns
- ▶ We know we can always select a new row (as long as we have a column), because the problem is bounded
- ▶ When we finish, if $u_0 = 0$, then either
 - ▶ we did a pivot in every row, and $R = \emptyset$
 - ▶ or we need to think a little more ...



Simplex Method Phase I Properties

Row k of M is denoted \mathbf{r}_k , but write \mathbf{u} for the special u -row.

Theorem

For each tableau in Phase I

$$\mathbf{u} = - \sum_{i \in R} \mathbf{r}_i, \text{ i.e., } d_j = \sum_{k \in R} a_{kj}, \text{ and } u_0 = - \sum_{k \in R} b_k$$

Proof : (by induction):

The equation is true for the initial tableau by construction.

Let's suppose that the result is true for some tableau M

Perform a pivot at (i, j) (satisfying Simplex pivot rules) to go from

$$M \rightarrow M'$$

then proof by induction requires us to show the result is true for M' .



Simplex Method Phase I Properties

Perform a pivot at (i, j) to go from $M \rightarrow M'$

$$\begin{aligned}\mathbf{u}' &= \mathbf{u} + \frac{d_j}{a_{ij}} \mathbf{r}_i \\ &= - \sum_{k \in R} \mathbf{r}_k + \frac{\sum_{k \in R} a_{kj}}{a_{ij}} \mathbf{r}_i \\ &= - \sum_{k \in R, k \neq i} \left(\mathbf{r}_k - \frac{a_{kj}}{a_{ij}} \mathbf{r}_i \right) - \left(\mathbf{r}_i - \frac{a_{ij}}{a_{ij}} \mathbf{r}_i \right) \\ &= - \sum_{k \in R, k \neq i} \left(\mathbf{r}_k - \frac{a_{kj}}{a_{ij}} \mathbf{r}_i \right) \\ &= - \sum_{k \in R'} \left(\mathbf{r}_k - \frac{a_{kj}}{a_{ij}} \mathbf{r}_i \right)\end{aligned}$$

Q.E.D.

Simplex Method, Phase I Properties



THE UNIVERSITY
of ADELAIDE

Now consider the case $u_0 = 0$ but R is non-empty

- ▶ we have reached the end
- ▶ but haven't got a complete set of basic columns

Then

- ▶ the solution must be feasible $b_i \geq 0$
- ▶ from previous theorem $\sum_{i \in R} b_i = -u_0 = 0$
- ▶ so $b_i = 0$ for all $i \in R$
- ▶ if all elements in the row are equal to 0 then this row is a linear combination of the other rows and is therefore redundant and can be removed.
- ▶ otherwise we can pivot on any element $a_{ij} \neq 0$ in such a row without changing the last column of the tableau
 - ▶ we do these pivots to create our necessary basic variables



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

2.8 Putting it together

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

The Simplex Algorithm (Phase I and II) always stops in one of the following ways:

- ▶ *stop 1*: (at end of in Phase 1) there are no feasible solutions.
 - ▶ because $u^* < 0$
- ▶ *stop 2*: (during Phase 2) there are feasible solutions with arbitrarily large z -values, and therefore no optimal solutions;
 - ▶ because we couldn't find a valid pivot row
- ▶ *stop 3*: (at the end of Phase 2) feasible optimal solution has been found;
 - ▶ because we couldn't find a valid pivot column



Mix and Match

Sometimes we start with a problem that has inequalities, and equalities. We could convert to standard form, and continue, but the initial inequalities already introduce slack variables (and hence basic columns), so we don't really need artificial variables for these, so for

$$\underset{\mathbf{x}}{\operatorname{argmax}} \left\{ z = \mathbf{c}^T \mathbf{x} + z_0 \mid B\mathbf{x} \leq \mathbf{b}, D\mathbf{x} = \mathbf{d}, \mathbf{x} \geq 0 \right\}$$

the Simplex Phase I Tableau would be

	s_i 's	a_i 's	z col.	u col.	b col.
B	I	0	0	0	\mathbf{b}
D	0	I	0	0	\mathbf{d}
$-\mathbf{c}^T$	0	0	1	0	z_0
$-\mathbf{d}^T$	0	0	0	1	u_0

z row

u row

Mix and Match Example

Example

$$\begin{array}{rclclclcl} \max z & = & x_2 & - & 3x_4 & + & 2 \\ & & x_1 & + & x_3 & - & x_4 & = & 2 \\ & & x_2 & + & x_3 & + & x_4 & = & 2 \\ & & x_2 & & & - & x_4 & \leq & 2 \\ & & & & & & x_i & \geq & 0 \end{array}$$

The Phase I Tableau is

x_1	x_2	x_3	x_4	x_5	z	u	b
1	0	1	-1	0	0	0	2
0	1	1	1	0	0	0	2
0	1	0	-1	1	0	0	2
0	1	0	-3	0	1	0	2
-1	-2	-2	1	1	0	1	-6

Why do we teach Simplex?

- ▶ because it shows one of the greatest ideas in algorithm development
 - ▶ construct a new problem that is simpler to solve, but helps you with your original problem
- ▶ this idea is reused in solution of many mathematical problems



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

2.9 Duality

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

Problem Recap



THE UNIVERSITY
of ADELAIDE

We will start with a problem in *standard equality form*.

$$\begin{aligned} \max \quad z &= \mathbf{c}^T \mathbf{x} + z_0 \\ \text{such that } A\mathbf{x} &= \mathbf{b} \\ \text{and } \mathbf{x} &\geq 0 \end{aligned}$$

We call this the *primal* LP

Primal (*P*)

$$\begin{aligned} \max \quad z &= \mathbf{c}^T \mathbf{x} + z_0 \\ \text{such that } A\mathbf{x} &= \mathbf{b} \\ \text{and } \mathbf{x} &\geq 0 \end{aligned}$$

Consider a new LP called the *dual* problem (*D*)

$$\begin{aligned} \min \quad w &= \mathbf{b}^T \mathbf{y} + z_0 \\ \text{such that } A^T \mathbf{y} &\geq \mathbf{c} \\ \text{and } \mathbf{y} &\text{ free} \end{aligned}$$



Origin of the dual

Suppose there is an optimal solution (\mathbf{x}^*, z^*) to the primal LP

$$\begin{array}{lll} \max & z & = \mathbf{c}^T \mathbf{x} + z_0 \\ \text{such that} & A\mathbf{x} & = \mathbf{b} \\ \text{and} & \mathbf{x} & \geq \mathbf{0} \end{array}$$

with $z^* = \mathbf{c}^T \mathbf{x}^* + z_0$.

We might have obtained this via Simplex

Initial Tableau

A	\mathbf{b}
$-\mathbf{c}^T$	z_0

Simplex



Final Tableau

\hat{A}	$\hat{\mathbf{b}}$
$-\hat{\mathbf{c}}^T$	\hat{z}_0

where $z^* = \hat{z}_0$



Origin of the dual

The final tableau comes from pivots, so there are some numbers y_1^*, \dots, y_m^* such that

$$\begin{aligned}-\hat{c}_j &= \sum_{i=1}^m y_i^* a_{ij} - c_j, \quad j = 1, \dots, n \\ \hat{z}_0 &= \sum_{i=1}^m y_i^* b_i + z_0.\end{aligned}$$

At the end of Simplex $-\hat{c}_j \geq 0$ so

$$\sum_{i=1}^m y_i^* a_{ij} \geq c_j \quad \text{for } j = 1, \dots, n.$$

So let's consider any variables y_1, \dots, y_m which satisfy

$$\sum_{i=1}^m y_i a_{ij} \geq c_j, \quad \text{for } j = 1, \dots, n.$$

We could look for an optimisation to find the \mathbf{y}^*



Origin of the dual

Let's build a new objective function with

$$w = \sum_{i=1}^m y_i b_i + z_0.$$

From the Primal, $b_i = \sum_{j=1}^n a_{ij}x_j$ so

$$\begin{aligned} w &= \sum_{i=1}^m y_i \left(\sum_{j=1}^n a_{ij}x_j \right) + z_0 \\ &= \sum_{j=1}^n \left(\sum_{i=1}^m y_i a_{ij} \right) x_j + z_0 \\ &\geq \sum_{j=1}^n c_j x_j + z_0 \end{aligned}$$

Hence $w \geq z$, but also the above is true for any feasible \mathbf{x} , so $w \geq \hat{z}_0$, the optimal value of the primal. But we defined w so that $w(\mathbf{y}^*) = \hat{z}_0$, so \hat{z}_0 is the minimum of w .



The Dual of an LP (Summary)

In the dual, variables become constraints, and visa versa

$$(P) \quad \max z = \sum_{j=1}^n c_j x_j + z_0$$

$$\sum_{j=1}^n a_{ij} x_j = b_i, \quad i = 1, \dots, m$$

$$x_j \geq 0, \quad j = 1, \dots, n$$

or

$$\max z = \mathbf{c}^T \mathbf{x} + z_0,$$

$$A\mathbf{x} = \mathbf{b},$$

$$\mathbf{x} \geq 0.$$

$$(D) \quad \min w = \sum_{i=1}^m y_i b_i + z_0$$

$$y_i \text{ free, } \quad i = 1, \dots, m$$

$$\sum_{i=1}^m y_i a_{ij} \geq c_j, \quad j = 1, \dots, n$$

or

$$\min w = \mathbf{y}^T \mathbf{b} + z_0,$$

$$\mathbf{y}^T A \geq \mathbf{c}^T,$$

$$\mathbf{y} \text{ free.}$$

Note how the lines are paired up

Dual Example



THE UNIVERSITY
of ADELAIDE

Example (Dual)

$$(P) \quad \begin{aligned} \max z &= 3x_1 - x_2 + x_3 \\ \text{s.t.} \quad & x_1 + 2x_2 = 5 \\ & x_1 - x_2 + x_3 = 7 \\ & x_2 + 6x_3 = 11 \end{aligned}$$

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0.$$

$$(D) \quad \begin{aligned} \min w &= 5y_1 + 7y_2 + 11y_3 \\ \text{s.t.} \quad & y_1 + y_2 \geq 3 \\ & 2y_1 - y_2 + y_3 \geq -1 \\ & y_2 + 6y_3 \geq 1 \end{aligned}$$

y_1, y_2 and y_3 are all free variables.



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

2.10 Duality proofs

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



Why do we call it the dual?

Theorem

The Dual of the Dual is the Primal.

Proof: The dual is

$$\begin{array}{ll} \min & w = \mathbf{b}^T \mathbf{y} + z_0 \\ \text{such that} & A^T \mathbf{y} \geq \mathbf{c} \\ & \text{and} \quad \mathbf{y} \text{ free} \end{array}$$

We convert to standard equality form by

- ▶ multiplying the objective by -1 (to make it a max)
- ▶ multiplying the constraints by -1
- ▶ adding slack variables
- ▶ replacing free variables y_i with non-negative variables $y_i^+ \geq 0$ and $y_i^- \geq 0$ such that $y_i = y_i^+ - y_i^-$,



Why do we call it the dual?

Proof: (continued) The dual, written in standard equality form is (D')

$$\begin{array}{lll} \max & -w & = \mathbf{b}^T \mathbf{y}^- - \mathbf{b}^T \mathbf{y}^+ - z_0 \\ \text{such that} & A^T \mathbf{y}^- - A^T \mathbf{y}^+ + \mathbf{s} & = -\mathbf{c} \\ \text{and} & \mathbf{y}^+, \mathbf{y}^-, \mathbf{s} & \geq 0 \end{array}$$

Now we can take the dual of (D') which we denote (DD) by creating a variable x_i for each constraint, and a constraint for each variable y_j and s_j .

$$\begin{array}{lll} \min & u & = -\mathbf{c}^T \mathbf{x} - z_0 \\ \text{such that} & A\mathbf{x} & \geq \mathbf{b}^T \quad \text{from } \mathbf{y}^- \\ & -A\mathbf{x} & \geq -\mathbf{b}^T \quad \text{from } \mathbf{y}^+ \\ \text{and} & \mathbf{x} & \geq 0 \quad \text{from } \mathbf{s} \end{array}$$



Why do we call it the dual?

Proof: (continued) We have (DD)

$$\begin{array}{llll} \min & u & = & -\mathbf{c}^T \mathbf{x} - z_0 \\ \text{such that} & A\mathbf{x} & \geq & \mathbf{b}^T & \text{from } \mathbf{y}^- \\ & -A\mathbf{x} & \geq & -\mathbf{b}^T & \text{from } \mathbf{y}^+ \\ \text{and} & \mathbf{x} & \geq & 0 & \text{from } \mathbf{s} \end{array}$$

Note that

- ▶ we can multiply the objective by -1 (to have a max)
- ▶ The constraints $A\mathbf{x} \geq \mathbf{b}^T$ and $A\mathbf{x} \leq \mathbf{b}^T$ together imply that $A\mathbf{x} = \mathbf{b}$.

So (DD) \equiv (P).

Q.E.D.

Primal	Dual
$\max z = \mathbf{c}^T \mathbf{x} + z_0$	$\min w = \mathbf{b}^T \mathbf{y} + z_0$
$A\mathbf{x} = \mathbf{b}$	$A^T \mathbf{y} \geq \mathbf{c}$
$\mathbf{x} \geq 0$	\mathbf{y} free

Theorem

Given primal and dual problems as above have feasible solutions \mathbf{x} and \mathbf{y} , then $z \leq w$.



Weak Duality Proof

Proof: We essentially showed this earlier:

$$\begin{aligned} w &= \sum_{i=1}^m y_i b_i + z_0 \\ &= \sum_{i=1}^m y_i \left(\sum_{j=1}^n a_{ij} x_j \right) + z_0 \quad \text{because (P) requires } b_i = \sum_{j=1}^n a_{ij} x_j \\ &= \sum_{j=1}^n \left(\sum_{i=1}^m y_i a_{ij} \right) x_j + z_0 \quad \text{swapping order of summation} \\ &\geq \sum_{j=1}^n c_j x_j + z_0 \quad \text{from constraints of (D)} \\ &= z \end{aligned}$$

Hence $w \geq z$

Corollary

If we have feasible solutions \mathbf{x} and \mathbf{y} to the primal and dual respectively and $w = z$, then these are optimal solutions to their respective problems.

Proof: w is an upper bound on z (and visa versa), so if $z = w$ for a feasible solution, it has achieved its upper bound, and hence we have an optimal solution.

Q.E.D.

Corollary

If the primal (dual) problem is unbounded then the dual (primal) problem is infeasible.

Proof: If the primal were feasible and unbounded, then that means there can be no upper bound on z , so we cannot have a feasible solution the dual.

Similarly if the dual is unbounded.

Q.E.D.

Theorem

If the primal (dual) problem has a finite optimal solution, then so does the dual (primal) problem, and these two values are equal.

Proof: see later as a result of complementary slackness.



The Dual of an LP

Summary of Results for Primal/Dual pair (P) and (D)

1. For a feasible solution x_1, \dots, x_n of (P), with value z , and a feasible solution y_1, \dots, y_m of (D), with value w , we have
 $w \geq z$ (**Weak Duality**)
2. If (P) has an optimal solution then (D) has an optimal solution and
 $\max z = \min w$ (**Strong Duality**)
3. Because the dual of the dual is the primal, if (D) has an optimal solution then (P) has an optimal solution and
 $\max z = \min w$ (**Strong Duality**)
4. If (P) has no optimal solution ($z \rightarrow \infty$) then (D) cannot have a feasible solution (as $w \geq \max z$), and if (D) has no optimal solution ($w \rightarrow -\infty$) then (P) cannot have a feasible solution (as $z \leq \min w$).

The Dual of an LP



THE UNIVERSITY
of ADELAIDE

		Primal (P)		
		stop 1 optimal solution	stop 2 feasible sol^n , no opt. sol^n	stop 3 no feasible solution
Dual (D)	stop 1 optimal solution	possible ‡	impossible	impossible
	stop 2 feas. sol^n , no opt. sol^n	impossible	impossible	possible
	stop 3 no feasible solution	impossible	possible	possible

‡ ($\max z = \min w$)



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

2.11 Complementary Slackness

Ashley Dennis-Henderson

School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

Theorem (Complementary slackness)

Given primal problem P and dual problem D with feasible solutions \mathbf{x} and \mathbf{y} , respectively, then \mathbf{x} is an optimal solution of (P) and \mathbf{y} an optimal solution of (D) if and only if

$$x_j \left(\sum_{i=1}^m y_i a_{ij} - c_j \right) = 0, \quad \text{for } j = 1, \dots, n.$$

Implicit in this theorem is the Strong Duality Theorem, which we prove now, and the second part is called the *complementary slackness* relations.

Definition (Complementary Slackness Relations)

The relations $x_j \left(\sum_{i=1}^m y_i a_{ij} - c_j \right) = 0$, for each $j = 1, \dots, n$ are called the *Complementary Slackness Relations* (CSR).



Complementary slackness proof

Proof: We have already seen the argument:

$$z = \sum_{j=1}^n c_j x_j + z_0 \leq \sum_{j=1}^n \sum_{i=1}^m y_i a_{ij} x_j + z_0 \quad \left(\text{as } \sum_{i=1}^m y_i a_{ij} \geq c_j, x_j \geq 0 \right)$$

$$= \sum_{i=1}^m \sum_{j=1}^n a_{ij} x_j y_i + z_0$$

$$= \sum_{i=1}^m y_i b_i + z_0 = w \quad \left(\text{as } \sum_{j=1}^n a_{ij} x_j = b_i \right)$$

So $z \leq w$ for all feasible solutions

Proof : (cont)

From Strong Duality, if we have an optimal solution, then $z^* = w^*$. Reversing the above logic, we get

$$\sum_{j=1}^n c_j x_j + z_0 = \sum_{j=1}^n \left(\sum_{i=1}^m y_i a_{ij} x_j \right) + z_0$$

which, when we group the x_i terms together gives

$$\sum_{j=1}^n \left(\sum_{i=1}^m y_i a_{ij} - c_j \right) x_j = 0$$



Complementary slackness proof

Proof : (cont)

If we know that $n_j \geq 0$, and we have

$$\sum_j n_j = 0$$

then we must have $n_j = 0$.

This applies here because we know from the LP dual and primals that

$$\sum_{i=1}^m y_i a_{ij} - c_j \geq 0 \text{ and } x_j \geq 0.$$

Hence

$$\left(\sum_{i=1}^m y_i a_{ij} - c_j \right) x_j = 0, \text{ for each } j = 1, \dots, n.$$





Complementary Slackness: Example

Primal Tableax (P)

x_1	x_2	x_3	x_4	x_5	b
2	-1	-1	1	0	1
2	2	1	-2	0	3
1	1	1	1	1	2
-1	1	1	1	0	0

Simplex Phase I and II

1	0	$-\frac{1}{6}$	0	0	$\frac{5}{6}$
0	1	$\frac{2}{3}$	-1	0	$\frac{2}{3}$
0	0	$\frac{1}{2}$	2	1	$\frac{1}{2}$
0	0	$\frac{1}{6}$	2	0	$\frac{1}{6}$

Resulting solution $\mathbf{x}^* = (5/6, 2/3, 0, 0, 1/2)$, with $z^* = 1/6$.

Complementary Slackness: Example



THE UNIVERSITY
of ADELAIDE

Primal Tableax (P)

x_1	x_2	x_3	x_4	x_5	b
2	-1	-1	1	0	1
2	2	1	-2	0	3
1	1	1	1	1	2
-1	1	1	1	0	0

Dual Tableax (D)
(note that these represent \geq)

\Rightarrow

y_1	y_2	y_3	c
2	2	1	1
-1	2	1	-1
-1	1	1	-1
1	-2	1	-1
0	0	1	0
-1	-3	-2	0



Complementary slackness

Write down the dual (D) of (P).

$$\begin{aligned}(D) \quad \min w &= y_1 + 3y_2 + 2y_3 \\ 2y_1 + 2y_2 + y_3 &\geq 1 && \text{(iv)} \\ -y_1 + 2y_2 + y_3 &\geq -1 && \text{(v)} \\ -y_1 + y_2 + y_3 &\geq -1 && \text{(vi)} \\ y_1 - 2y_2 + y_3 &\geq -1 && \text{(vii)} \\ y_3 &\geq 0 && \text{(viii)}\end{aligned}$$

y_1, y_2 and y_3 are all free variables⁶. (i)-(iii)

Writing down the CSR, we see that

- ▶ From (iv), we get $(2y_1^* + 2y_2^* + y_3^* - 1)x_1^* = 0$ and since $x_1^* > 0$, we have $2y_1^* + 2y_2^* + y_3^* = 1$.
- ▶ From (v), since $x_2^* > 0$, we have $-y_1^* + 2y_2^* + y_3^* = -1$.
- ▶ Similarly, from (viii), since $x_5^* > 0$, we have $y_3^* = 0$.

⁶Note the last constraint tightens y_3 , but this is OK.



Complementary slackness

Thus from the CSR for (iv),(v) and (viii) we see that

$$\begin{aligned}2y_1^* + 2y_2^* + y_3^* &= 1 \\ -y_1^* + 2y_2^* + y_3^* &= -1 \\ y_3^* &= 0\end{aligned}$$

Solving these equalities for y_1^*, y_2^*, y_3^* , we get $y_1^* = \frac{2}{3}$, $y_2^* = -\frac{1}{6}$, $y_3^* = 0$, which need to be checked for feasibility in the other constraints:

That is,

$$(vi) \quad -\frac{2}{3} - \frac{1}{6} + 0 > -1$$

$$(vii) \quad \frac{2}{3} + \frac{1}{3} + 0 > -1.$$

Also note that
$$\boxed{z^* = \frac{5}{6} - \frac{2}{3} = \frac{1}{6}} \quad = \quad \boxed{w^* = \frac{2}{3} - 3 \times \frac{1}{6} = \frac{2}{3} - \frac{1}{2} = \frac{1}{6}}.$$



- ▶ Optimisation problems have a Dual
 - ▶ this is a very general concept, and holds beyond LPs
- ▶ Complementary slackness relates dual solutions to primal
 - ▶ these give us an edge in knowing when we have found optimal solutions
 - ▶ we'll use these later!



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

2.12 Sensitivity Analysis

Ashley Dennis-Henderson

School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

Errors in Linear Program Formulation

- ▶ All data has errors, or *noise*
 - ▶ artefacts of measurement
 - ▶ we might only have estimates of hard-to-measure quantities
- ▶ Optimisation often considers the future
 - ▶ we base objectives and constraints on *predictions*
- ▶ Formulating a LP often involves *approximation*
 - ▶ quadratic cost approximated as linear
 - ▶ complex boundary approximated by linear segments

All of these factors mean that the LP that we solve today, might be different from the *real* problem we aim to solve

Errors in Linear Program Formulation

- ▶ All LPs have errors in their formulation
- ▶ Do these errors matter?
 - ▶ maybe a small error doesn't really change the result?
 - ▶ maybe even a large error only has a small effect?
- ▶ *Sensitivity analysis* is the process of learning about how *sensitive* or *robust* our LP is to such errors



Types of errors

We can have errors in several components of the problem

- ▶ The objective function coefficients **c**
- ▶ The constraint coefficients **A**
- ▶ The constraint coefficients **b**
- ▶ We might want to add a constraint
- ▶ We might want to add a variable



1. changes in **c** can
 - 1.1 change the vertex of the optimal solution
 - 1.2 keep the same vertex, but change the objective function value
 - 1.3 have no effect at all
2. changes in **A** and **b** perturb the shape of the feasible region
 - 2.1 this could toggle feasibility of the problem
 - 2.2 it could change the vertex of the optimal solution
 - 2.3 it could change the location of the optimal vertex
 - 2.4 it might have no effect at all

Formal Sensitivity Analysis

- ▶ We can calculate these things formally
 - ▶ Use clever matrix analysis
 - ▶ Avoids costly recalculations
 - ▶ See lectures 20 and 21
- ▶ We'll start a bit simpler

2 Simplex algorithm

2.13 Empirical Sensitivity Analysis

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

Fundamental idea

- ▶ if you are worried about the effect of an error
- ▶ try it out!



An error in the objective coefficients \mathbf{c}

Example $z = [c_1, c_2]^T \mathbf{x}$

- ▶ Consider alternative objectives, e.g.,

- ▶ $[c_1 + \delta, c_2 - \delta]$
- ▶ $[c_1, c_2 + \delta]$

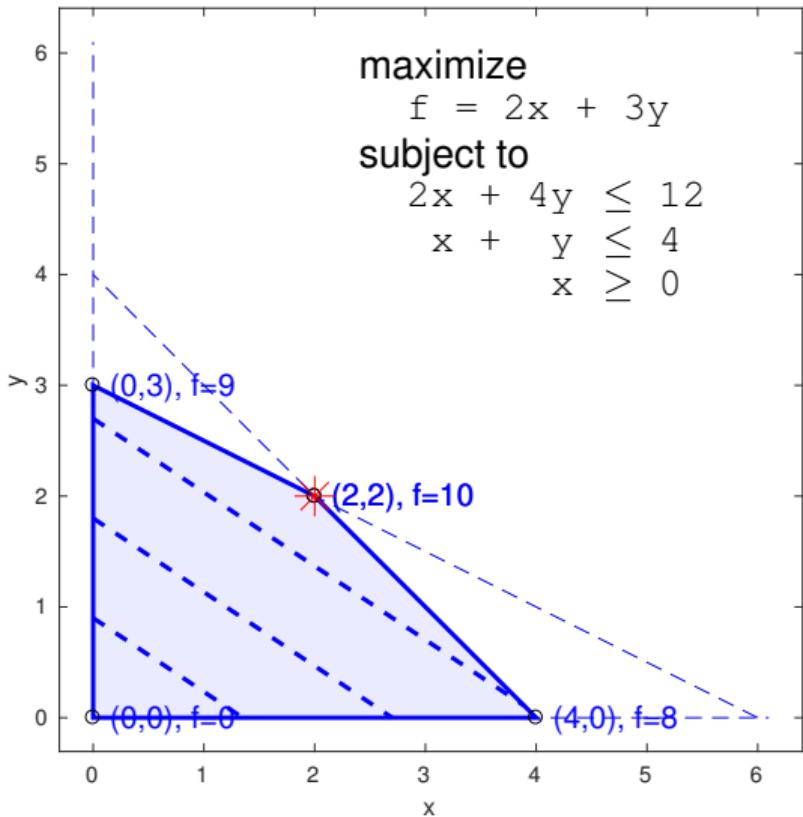
your choice depends on what you know about the errors, or how you want to examine their effect

- ▶ Now solve the new problem, e.g.,

$$\begin{aligned} \max \quad & z = [c_1 + \delta, c_2 - \delta]^T \mathbf{x} \\ \text{subject to } & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

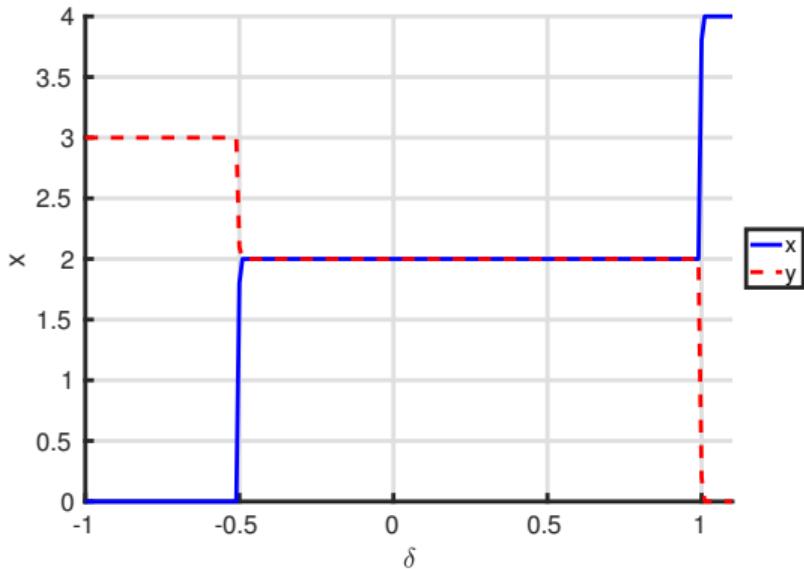
for a range of values of δ and plot the results

Example



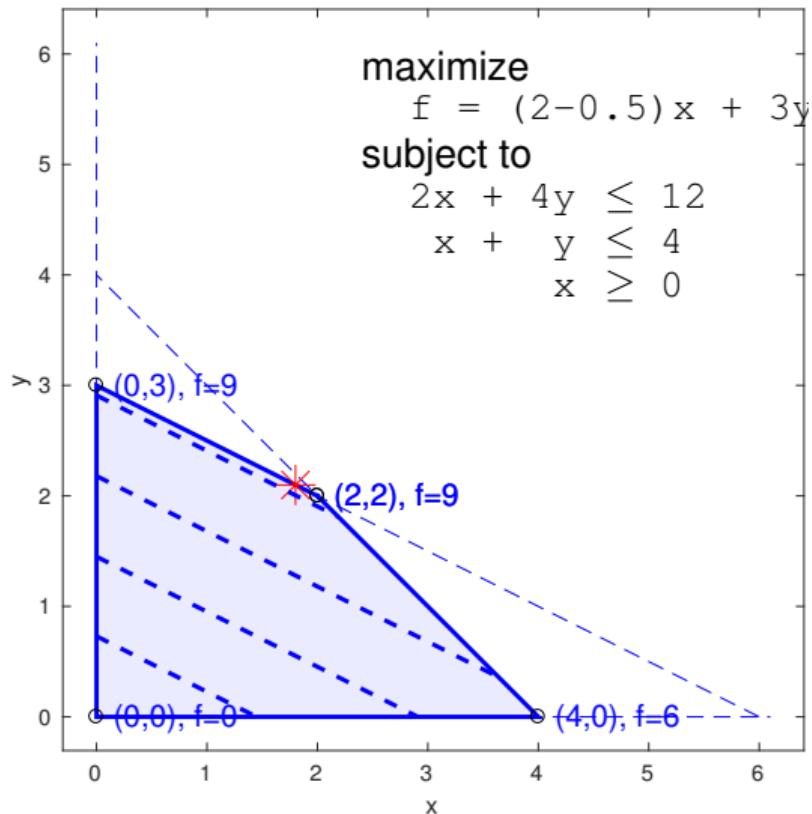
Take $f = (2 + \delta)x + 3y$

Example



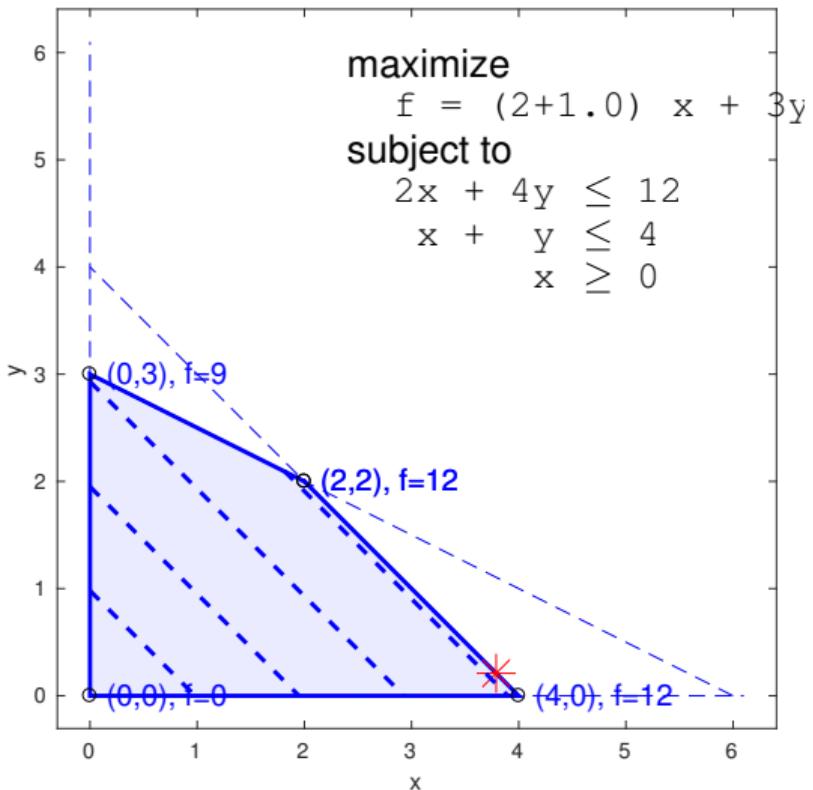
Take $f = (2 + \delta)x + 3y$

Example



Take $f = (2 + \delta)x + 3y$

Example



Take $f = (2 + \delta)x + 3y$

An error in the constraints

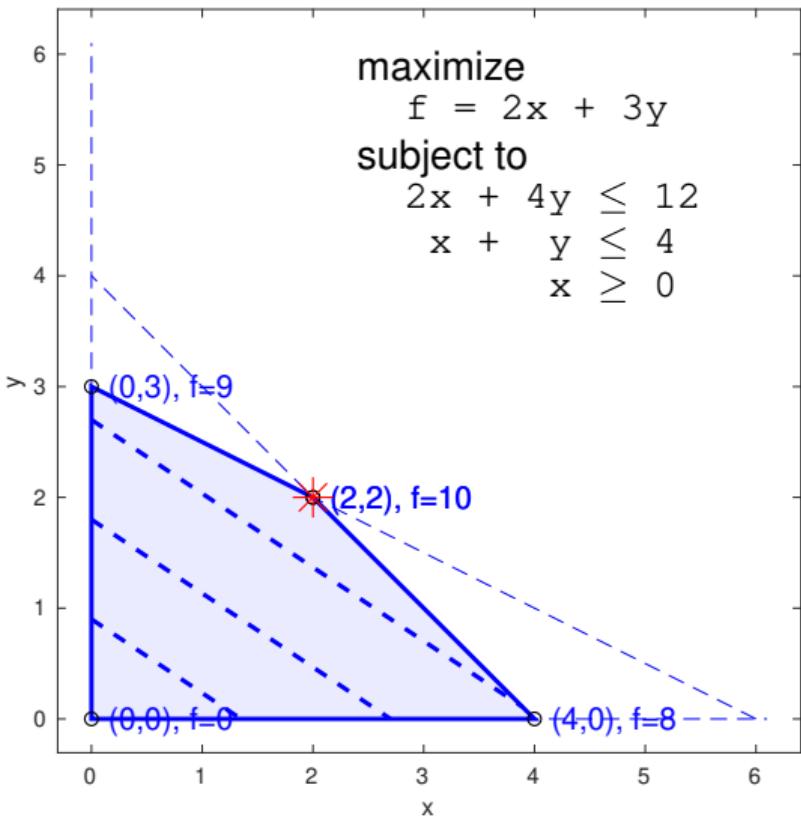


THE UNIVERSITY
of ADELAIDE

- ▶ As before, formulate the error model
- ▶ Substitute the new values in the problem and solve
 - ▶ e.g., take \mathbf{b}_δ as the constraint values plus error of size δ and solve

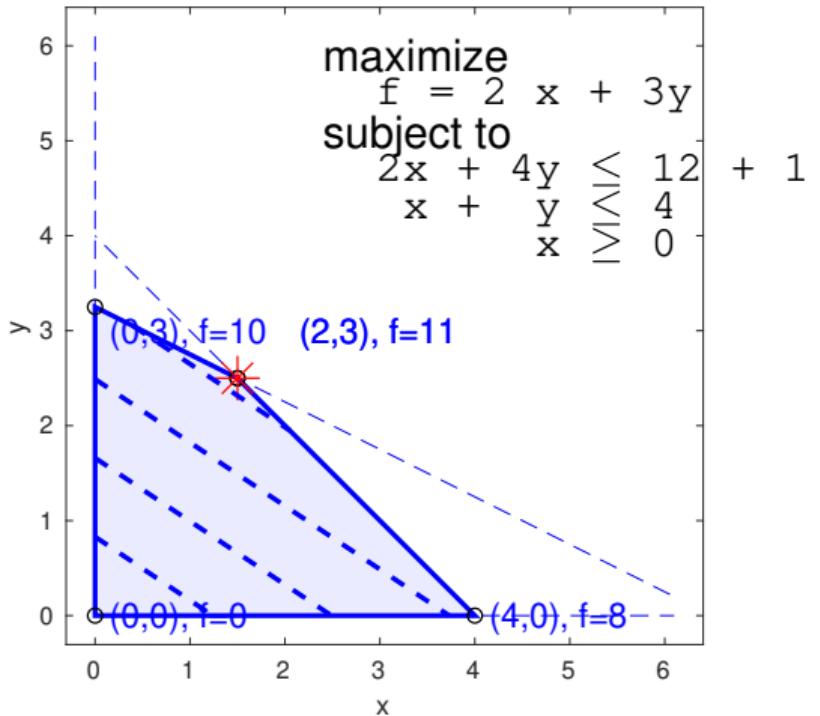
$$\begin{aligned} & \max \quad z = \mathbf{c}\mathbf{x} \\ \text{subject to } & \mathbf{A}\mathbf{x} = \mathbf{b}_\delta \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

Example



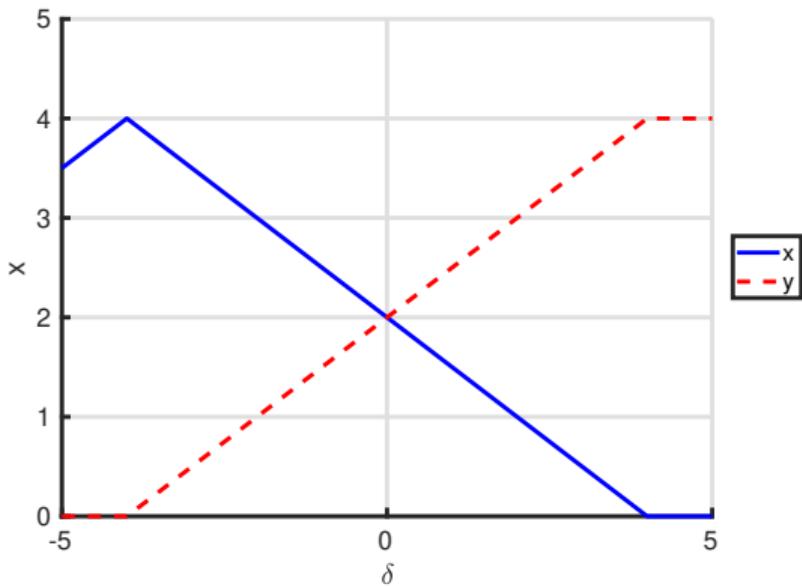
Use constraint $2x + 4y \leq 12 + \delta$

Example



Use constraint $2x + 4y \leq 12 + \delta$

Example



Use constraint $2x + 4y \leq 12 + \delta$



The hard part is choosing an error model:

1. As above, choose a set of simple changes to the coefficient vectors, and explore the effect of a range of values
 - 1.1 use simple scenarios to explore the space
2. Alternatively, add random noise to coefficients
 - 2.1 relatively easy
 - 2.2 scale/size of errors can be controlled by standard deviation

but this might be unrealistic: e.g., might end up with negative b_i

The key is in understanding your problem well.



OPTIMISATION & OPERATIONS RESEARCH

2 Simplex algorithm

2.14 Interrogating a problem

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



One of the hardest things in mathematics is transcribing a problem into mathematics in the first place.

- ▶ Customers and managers don't have the terminology to tell you what you want
 - ▶ they speak a different language, literally
- ▶ They sometimes don't know what the problem is
 - ▶ because they don't know what is possible
- ▶ The data they have is usually a mess
 - ▶ The age "Big Data" didn't change that, it just meant there was more mess

Interrogating the Problem

We'll start simple, with *interrogating the problem*

- ▶ Assume the problem is known, and has been expressed in words
- ▶ We need to learn how to extract a mathematical description of the problem
- ▶ It will require
 - ▶ a bit of linguistics
 - ▶ some puzzle solving
 - ▶ some approximations
 - ▶ a clear understanding of where we are trying to get to

Interrogating the Problem

We've seen simpler examples

A manufacturing company makes three types of items: desks, chairs, and bed-frames, using metal and wood. A single desk requires 2 hours of labour, 1 unit of metal and 3 units of wood, ... In a given time period, there are 225 hours of labour available, 117 units of metal and 420 units of wood. The profit on one desk is \$13, Choose the right number of items to produce to maximise the company's profit.

Interrogating the Problem



THE UNIVERSITY
of ADELAIDE

1. Read the question right through!
2. Identify the variables
3. Identify the objective
4. Formulate the constraints

L[~]O[~]O[~]k carefully.

Interrogating the Problem

A manufacturing company makes three types of items: desks, chairs, and bed-frames, using metal and wood. A single desk requires 2 hours of labour, 1 unit of metal and 3 units of wood, ... In a given time period, there are 225 hours of labour available, 117 units of metal and 420 units of wood. The profit on one desk is \$13, Choose the right number of items to produce to maximise the company's profit.

Variables: look for

- ▶ words like “choose” or “decide”
- ▶ repeated words:
 - ▶ these *might* be related to variables
- ▶ we are looking for something numerical that we can control
- ▶ identify *units*
 - ▶ these are *required* but also give clues

Interrogating the Problem

A manufacturing company makes three types of items: desks, chairs, and bed-frames, using metal and wood. A single desk requires 2 hours of labour, 1 unit of metal and 3 units of wood, ... In a given time period, there are 225 hours of labour available, 117 units of metal and 420 units of wood. The profit on one desk is \$13, Choose the right number of items to produce to **maximise** the company's **profit**.

Objective: look for

- ▶ words like “maximise” or “minimise” or “profit” or “cost”
- ▶ we are looking for something numerical that we will optimise
- ▶ it should be written in terms of the variables
 - ▶ so this is another clue about variables
- ▶ then find the coefficients **c**
 - ▶ if its profit or loss, units (of **c**) should be **\$s per unit variable**
 - ▶ otherwise, look for the values/numbers related to the objective

Interrogating the Problem



THE UNIVERSITY
ofADELAIDE

A manufacturing company makes three types of items: desks, chairs, and bed-frames, using **metal** and wood. A single desk requires 2 hours of labour, 1 unit of **metal** and 3 units of wood, ... In a given time period, there are 225 hours of labour available, 117 units of **metal** and 420 units of wood. The profit on one desk is \$13, Choose the right number of items to produce to maximise the company's profit.

Constraints: look for

- ▶ words like “less than” or “no more” or “at least”
- ▶ repeated words:
 - ▶ these **might** be related to constraints
- ▶ the find the coefficients **A** and **b**
 - ▶ look for the values/numbers related to each constraints

Interrogating the Problem

The hardest parts are often *implicit* constraints

- ▶ No-one states that we can't have "negative" chairs
 - ▶ Likewise, there can be other constraints that seem so obvious (to the person setting the problem) that they don't state them
- ▶ Other constraints are in the problem statement, but spread out, and never explicitly stated (see following example)
- ▶ Often constraints require some reasoning
 - ▶ think about the meaning behind the words
 - ▶ think about "physics"
 - ▶ use common sense
- ▶ One BIG clue is that we are doing *Linear Programming*, so all of your constraints will be either linear inequalities, or linear equations

You have \$1 million to spend on a new coin collection. There are a variety of coins available. Each has characteristics of interest: rarity, age, and condition. You want a balance in your collection. That is, you want a certain number of coins that are rare, and a number that are old, a number in good condition, and so on. And you wish to maximise the total number of coins in the collection.

(actual numbers omitted for brevity)

- ▶ Variables: whether or not to purchase each possible coin.
 - ▶ notice that these are *binary* variables
- ▶ Objective: maximise the number of coins
- ▶ Constraints
 - ▶ the obvious, explicit constraints concern rarity, age, and condition of the overall collection
 - ▶ implicitly, each coin has a cost, and you can't spend more than \$1 million.

Takeaways

- ▶ All LPs contain errors
- ▶ Sensitivity analysis is used to see the effect of these errors
- ▶ Interrogating a problem
 - ▶ this is the HARDEST bit of mathematics



Algorithm Analysis

Matthew Roughan

<matthew.roughan@adelaide.edu.au>

<http://roughan.info/>

School of Mathematical Sciences,
University of Adelaide

January 7, 2022

Algorithm Analysis

- We would like to estimate how long our program will take to run
- More generally, how many resources will it take?
 - ▶ time
 - ▶ memory (space)
 - ▶ unicorns
- And often, we would like to make it better

Empirical Measures

- Run the program and test it
 - ▶ see how long it takes
 - ★ in MATLAB use tic and toc
 - ★ there are lots of tools, e.g., see *profilers*
 - ▶ see how much memory it uses
- This is simple, but
 - ▶ how do we *anticipate* performance for a new problem?
 - ▶ how do we determine the practical limits of our program?
 - ▶ how will our program run on a different computer?

Memory Analysis

- What takes up memory?
 - ▶ the program itself
 - ★ this is usually fairly constrained, so we mostly ignore it
 - ▶ the variables
 - ★ in MATLAB, most variables are double precision floating point
 - ★ effectively they take *8 bytes* each
- Memory analysis is just a matter of counting the number of variables
 - ▶ a vector of length n , takes $8n$ bytes
 - ▶ an $n \times m$ matrix takes $8nm$ bytes
- In general, there are many other issues to consider, but simple counting is the starting point

- Time analysis is more complicated
- What takes time?
 - ▶ each *operation* takes time, but they aren't all the same
 - ★ + might be faster than ×
 - ★ ×2 is often very fast in binary
 - ▶ the time an operation takes depends on the particular computer
 - ★ so often we don't work out actual time, we look directly at the number of operations
- So once again it is just counting, but
 - ▶ there are different types of operations to count
 - ▶ there are some extra complexities we will consider below

Time Analysis: simple examples

calculation	operations		notes
	+	×	
$(x_1 + x_2) \times x_3$	1	1	
$(x_1 \times x_3) + (x_2 \times x_3)$	1	2	same output as previous calc
x^6		5	assumes naïve multiplication
$A + B$	nm		for $n \times m$ matrices

It's just counting, but details matter!



Time Analysis: operations

There are lots of operations you could count

- *arithmetic*: e.g., \times , $+$, $-$, $/$, ...
- *relational*: e.g., comparisons $x > 0$, $y == 2$
- *logic*: if true ...
- *bitwise*: we don't use these much in MATLAB
- *set*: e.g., union, intersection, ...
- *memory access*: e.g., creating a variable (memory allocation), setting a variable, reading from an array, ...
- functions you call contain multiple operations: e.g., $\sin(x)$
- in MATLAB vector operations are actually made up of lots of smaller operations, e.g., $A + B$ would need to add all of the elements
- Input/Output (to screen or disk) – be aware this is *SLOW*

Time Analysis: operations

- There are lots of operations you need to consider
- We aim to break it down to *primitive operations*
 - ① e.g., arithmetic, relational, and logic
 - ② Separate I/O from the algorithm
 - ★ make sure MATLAB lines end with a ;'
- Take *uniform-cost model*
 - ① assume all primitive operations take the same time

Time Analysis: loops

```
for i=1:n
    % do some stuff that takes k operations
    ...
end
```

- The cost of a loop is the internal cost (assume k) multiplied by the number of times the loop runs (here n)
- So the cost here is nk operations

Time Analysis: loops example

```
for i=1:m
    for j=1:n
        x = x + (i*j)
    end
end
```

- The inner loop does **2** primitive ops
 - ▶ Note that in *this* example, it doesn't depend on the values of x , i or j
- The inner “ j ” loop performs this n times, so **$2n$** ops
- The outer “ i ” loop repeats this m times, so the final count is

$$2nm$$

operations.

- Break the program into blocks
 - ▶ we can just add up the cost of each block
- Look for loops
 - ▶ the cost of the “stuff” inside the block is *multiplied* by the number of times the loop runs
- Some cases require some thought

Example: evaluating x^n

- Naïvely we do this with $n - 1$ multiplications, but that is slow
- Actually we use tricks like $x^n = \exp(n \log x)$
- Simple functions like $\exp(\cdot)$ and $\log(\cdot)$ are $O(1)$ (for fixed precision)
- So $x^n = O(1)$

Time Analysis: indeterminacy

The big problem for analysing cost is indeterminacy, i.e., sometimes the code's behaviour changes depending on the values

```
if (x > 0)
    y = x + 1
else
    y = x + z + w
end
```

Often we don't know what value of x to expect (that's might be the whole point of the program) so how should we analyse this?

We'll look at this a little later.

- ➊ Modern computers really mess all this up
 - ➊ CPU can perform multiple operations per clock cycle, under certain (complex) conditions
 - ➋ Multiple levels of cache change speed to access (and hence operate) on variables
 - ➌ ...
- ➋ So what do we do?
 - ➊ Big-O notation (see next section) abstracts away some details
 - ➋ Complexity analysis looks at the *class* of the algorithm rather than the details, but we will look at this later

Conclusion

- Complexity analysis is just counting
- Break your code into blocks, and look for loops
- It gets hard, though, in real programs, so we need some more tricks – see the next video.



Big-O Notation and Friends

Matthew Roughan

<matthew.roughan@adelaide.edu.au>

<http://roughan.info/>

School of Mathematical Sciences,
University of Adelaide

January 7, 2022

Computational complexity

- Often, we don't care about the time for a particular problem, we care about the practical bounds for problems we might consider in the future
- We would like to estimate how long our program will take to run
 - ▶ as a function of the *size* of the problem
 - ★ e.g., n equals the number of variables
 - ★ e.g., m equals the number of constraints
 - ▶ could also include the size of the variables in memory
 - ★ e.g., k bit floating point numbers
- often interested in BIG problems, so look at asymptotic behaviour
 - ▶ e.g., large m and n
 - ▶ use big-O notation



Definition

$$f(x) = O(g(x))$$

means (i.e., iff) there exists constant c and x_0 such that

$$|f(x)| \leq c|g(x)|$$

for all x such that $x_i \geq x_0$.

Usage:

- describes asymptotic limiting behaviour: implicit that $x \rightarrow \infty$
- the function $g(x)$ is chosen to be as simple as possible
- a common **mistake** is to think that it means $f(x)/g(x) \rightarrow k$

Big-O notation properties

- Multiplication: $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then

$$f_1 \times f_2 = O(g_1 \times g_2)$$

- Multiplication by a constant: $f = O(g)$

$$kf = O(g)$$

- Summation: $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then we can write a general expression, but usually either $g_1 = g_2$, or WLOG g_1 grows faster than g_2 and in these cases

$$f_1 + f_2 = O(g_1)$$

These properties mean that we can simplify using a simple set of rules

Big-O notation rules

When we use Big-O notation, we use the following rules:

- ① if $f(x)$ is a sum drop everything except the term with the largest growth rate
- ② if $f(x)$ is a product any constants are ignored

Assume these rules have been applied, when you see Big-O.



Example of RULE-1

Example

$f(x) = x^7 - 200x^4 + 10$ is dominated (for large x) by the x^7 term, so

$$f(x) = O(x^7)$$

We dropped the terms $-200x^4 + 10$ because they grow slower than x^7 .

Example

We can reduce $O(n^2 + \log n)$ to $O(n^2)$.

The $\log()$ function grows more slowly than n (or any polynomial).



Example of RULE-2

Example

$f(n) = 3n^2$, which is a product, so we ignore constants, and

$$f(n) = O(n^2)$$

We ignored the constant 3.

Example

If k is a constant, we can rewrite $O(kn \log n)$ as $O(n \log n)$.

Whether k is a constant depends on the context.

Stirling's approximation

Stirling's approximation is both an example of use of the notation, and also a useful tool in some analysis:

$$\ln n! = n \ln n - n + O(\ln n)$$

We use Big-O notation here

We will use Big-O notation to count operations in an algorithm



Classic examples

problem	complexity	notes
$\sum_{i=1}^n x_i$	$O(n)$	
$A \times B$	$O(n^3)$	naïve algorithm
	$O(n^{2.373})$	clever algorithm
A^{-1}	$O(n^3)$	naïve algorithm
	$O(n^{2.373})$	clever algorithm
$\det(A)$	$O(n!)$	naïve algorithm
	$O(n^3)$	clever algorithm

Where A and B are $n \times n$ matrices



Example of a more complicated function

Example

Calculate the complexity of computing $f(x) = \exp(x)$.

- This depends on how you compute $\exp(x)$.
- A simple approach is Taylor series
 - ▶ assume you want n digits of precision
 - ▶ that determines how many terms you need in the Taylor series
 - ▶ so computation is $O(nM(n))$, where $M(n)$ is the cost of a multiplication with n digits
- Assuming fixed precision (e.g., in MATLAB, double precision)

$$\exp(x) = O(1)$$

That is, its computational time doesn't depend on how big x is

- There are faster approaches, but this suffices for today
- Other elementary functions, e.g., \sin , \cos , \arctan , \log , are similar

In order, we describe classes of algorithms as ???-time (e.g., constant-time)

complexity	name	example algorithms
$O(1)$	constant	calculate simple functions
$O(\log n)$	logarithmic	binary search
$O(n)$	linear	adding arrays of length n
$O(n \log n)$	log linear	Fast Fourier Transform (FFT)
$O(n^2)$	quadratic	adding up all elements of a matrix
$O(n^d)$	polynomial	naïve matrix multiplication
$O(c^n)$	exponential	Simplex
$O(n!)$	factorial	brute force search for TSP



Example

$$x = O(x^2) \quad \text{but} \quad x^2 \neq O(x)$$

so using $=$ is slightly weird, as there is an asymmetry.
Sometimes we use \in instead.

e.g.,

$$x \in O(x^2)$$



Often the symbols are used more generally

Sometimes we use these symbols in a type of algebra

Example

$$(n + O(n^{1/2})) (n + O(\log n))^2 = n^3 + O(n^{5/2})$$

Meaning: for any functions which satisfy each $O(\dots)$ on the LHS, there are some functions satisfying each $O(\dots)$ on the RHS, such that substituting all these functions into the equation makes the two sides equal.

It can get confusing, as variables and constants sometimes are inferred from context.

For instance

$$\begin{aligned}f(n) &= O(n^m) \\g(m) &= O(n^m)\end{aligned}$$

mean quite different things, even though the RHSs are the same.

Big-O limitations

Big-O has advantages:

- it gets to the nub of the question – what is the *shape* of the performance of our algorithm for large problems

However it has limitations

- it doesn't tell us about constants, and lower-order terms
 - ▶ these are important, particular for small to moderate sized problems
 - ▶ Big-O is only for asymptotic performance
- it doesn't tell us actual computation times
- *it's only an upper bound*

Two forms of Big-Omega notation

- Hardy-Littlewood (used in math)
- Knuth (used in computational complexity)

Definition (Big Omega)

$$f(x) = \Omega(g(x)) \Leftrightarrow g(x) = O(f(x))$$

More succinctly: $f(x) \geq kg(x)$ for some k

- Similar to Big-O, but gives a lower bound

Definition (Big Theta)

$$f(x) = \Theta(g(x))$$

means that $f(\cdot)$ is bounded above and below by $g(\cdot)$, i.e.,

$$k_1 g(x) \leq f(x) \leq k_2 g(x)$$

for positive constants k_1 and k_2 , for all $x > x_0$.

So Big- Θ notation means the function $f(x)$ grows as fast as $g(x)$.

Big-? notations are used to provide asymptotic descriptions of algorithm performance

- Big-O notation means the function grows no faster than
- Big- Ω means the function grows faster than, and
- Big- Θ notation means the function grows as fast as.

The notation has some big advantages, but also some big gotcha's.



Worst Case Algorithm Analysis

Matthew Roughan

<matthew.roughan@adelaide.edu.au>

<http://roughan.info/>

School of Mathematical Sciences,
University of Adelaide

January 7, 2022

Strategy for counting operations

- Break the program into blocks
 - ▶ we can just add up the cost of each block
- Look for loops
 - ▶ the cost of the “stuff” inside the block is *multiplied* by the number of times the loop runs
- Using Big-O notation simplifies our work
 - ▶ Big-O with respect to asymptotic problem *size*
 - ▶ We only have to keep track of “biggest” parts of an algorithm
 - ▶ Uniform-cost model works
 - ★ constant factors drop out, so
 - ★ we can count all operation types as $O(1)$
 - ▶ Elementary functions are also $O(1)$

Time Analysis: indeterminacy

The big problem for analysing cost is indeterminacy, i.e., sometimes the code's behaviour changes depending on the values

```
if (x > 0)
    y = x + 1
else
    y = very_complicated_function(x)
end
```

Often we don't know what value of x to expect (that's might be the whole point of the program) so how should we analyse this?



Worst case analysis

- Usually look at worst case performance
 - ▶ because we are conservative
 - ▶ because this is often easier to calculate
- In the example above, the worst case is the `very_complicated_function` so we use its complexity
- The hard cases involve things like
 - ▶ a loop where the number of iterations is variable
 - ▶ complicated logic and calculations that depend on inputs e.g., Simplex
- Let's have a bit more of a look
- Remember we will use Big-O notation, which makes our lives easier

Any (interesting) algorithm can solve many *instances* of a given problem

- e.g., Simplex can solve lots of different LPs
 - ▶ any given set of objective and constraints defines an instance
- Computation time depends on the particular instance
- We usually parameterise instances by *size*
 - ▶ the size of a LP is the number of variables and constraints
- The *worst case* is the instance of a given size that takes the longest to solve
 - ▶ note this is a very specific sense of “worst”



Example 2

Evaluating a degree n polynomial

$$a_n x^n + \cdots + a_1 x + a_0$$

- Assume we calculate it as written
- The instance *size* is the degree n
- The worst case instance: all of the coefficients a_i are non-zero
- In the worst case, we have to do at least n additions
- As noted in a previous video $x^n = O(1)$

So evaluating a degree- n polynomial directly is $O(n)$



Example 2b

Evaluating a degree n polynomial

$$a_n x^n + \cdots + a_1 x + a_0$$

- Naïve way to evaluate is to calculate the sum as it is written
- There is a better way – Horner's algorithm

$$\left(\left((a_n x + a_{n-1}) x + a_{n-2} \right) x \cdots + a_1 \right) x + a_0$$

- Note that this is still $O(n)$, but it will typically be faster
 - ▶ Big-O notation hides constant factors

Example 3

```
if (condition)
    % sequence of statements 1 which is O(n)
else
    % sequence of statements 2 which is O(n^2)
end
```

- Assume worst case, so take the worst branch
- Algorithm is $O(n^2)$

Example 4

```
while (n > 0)
    % make n smaller by some amount x >= 1
    % steps take O(n^2) to calculate
end
```

- “Size” is n
- Assume work inside the loop takes $O(n^2)$
- The worst case is that $x = 1$ for all loops, so we have to go through the loop n times, and hence the algorithm is $O(n \times n^2) = O(n^3)$

We often look for “bounding” cases, i.e., worst cases where a loop is actuated the maximum possible number of times

- Break the program into blocks
 - ▶ we can just add up the cost of each block
- Look for the “biggest bits”
- Look for loops
 - ▶ the cost of the “stuff” inside the block is *multiplied* by the number of times the loop runs
- Where there are indeterminate parts, look for the worst case
 - ▶ branches (conditionals) – choose worst branch
 - ▶ loops – look for bounds
- We can often cheat
 - ▶ skip calculations for “small” bits
 - ▶ there are known complexities for many common algorithms – you can use these as blocks

Other measures of complexity

- Communications complexity
 - ▶ e.g., how much network traffic on a parallel cluster
- Memory complexity
 - ▶ how much memory is needed for the algorithm

We still use techniques like Big-0 and worst case analysis

Others ways to assess complexity

- Average case
 - ▶ Simplex typical case is polynomial, instead of exp
- Best case

Conclusion

- Complexity analysis is just counting, but most of the time the number of operations will depend on the data or the problem
- When the code complexity is indeterminate a common strategy is to calculate the worst case



Complexity of the Simplex Algorithm

Matthew Roughan

<matthew.roughan@adelaide.edu.au>

<http://roughan.info/>

School of Mathematical Sciences,
University of Adelaide

January 7, 2022

Example: Simplex

What is the computational complexity of Simplex?

Let's just look at Phase II, with n variables and n constraints

- The first thing to do is identify the important blocks
 - ▶ There are many steps that aren't costing a lot – we'll ignore them
 - ▶ It seems like the pivots might be costly, let's focus on them first
- The main source of indeterminacy is how many steps are needed



Example: pivot

```
1 function [Mout] = pivot(M, i, j, epsilon);
2 %
3 % pivot.m, (c) Matthew Roughan, 2015
4 %
5 % created:      Wed Jul 1 2015
6 % author:       Matthew Roughan
7 % email:        matthew.roughan@adelaide.edu.au
8 %
9 % Perform a pivot at position (i,j) of matrix M
10 %
11 % INPUTS:
12 %      M      = Tableau on which we operate
13 %      (i,j) = pivot location
14 %      optional inputs
15 %          epsilon = small number so that we don't test "exactly" zero, b
16 %
17 % OUTPUTS:
18 %      M_out
19 %
```



Example: pivot

```
20 if nargin < 4
21     epsilon = 1.0e-12;
22 end
23
24 % check inputs
25 assert(i>=1 && i<=size(M,1) && i == round(i), 'invalid value of i');
26 assert(j>=1 && j<=size(M,2) && j == round(j), 'invalid value of j');
27 assert(abs(M(i,j)) > epsilon, 'M(i,j) close to zero');
28
29 if abs(M(i,j)) < epsilon
30     error('M(i,j) close to zero');
31 end
```



Example: pivot

```
32
33 % create the output array
34 Mout = zeros(size(M));
35
36 % divide row i by M(i,j)
37 Mout(i,:) = M(i,:)/M(i,j);
38
39 % subtract enough of the new row from each other row to make other colu
40 for k=1:size(M,1)
41     if k ~= i
42         Mout(k,:) = M(k,:)-Mout(i,:)*M(k,j);
43     end
44 end
```



- Each pivot requires $O(n^2)$ operations
- If we do k pivots, then the algorithm would be $O(kn^2)$

But what is k ? It probably depends on n but how?

- ? $k = O(n^2)$
- ? $k = O(\exp(n))$

Simplex steps



THE UNIVERSITY
of ADELAIDE

- The Klee-Minty example (see your exercises) is an example of a Simplex problem where k is exponential, so the worst case behaviour of Simplex is (at least) exponential

Conclusion

- In worst case analysis Simplex is an exponential algorithm
 - ▶ but it works well in practice
- There are (guaranteed) polynomial time algorithms for solving LP (see interior point algorithms), though these aren't necessarily much faster on many problems, its just their worst case is guaranteed to be better.



Turing Machines

Max Ward-Graham

<max.ward-graham@adelaide.edu.au>

<http://roughan.info/>

School of Mathematical Sciences,
University of Adelaide

February 21, 2022

The Problem



THE UNIVERSITY
of ADELAIDE

- Computation time depends on the specifics of the computer
 - ▶ e.g., can it parallelise some operations?
 - ▶ What is an operation?
- Naive complexity analysis can be bogged down in details
- So we need a “universal” model computer to create formal arguments about what is computable and how fast
- Turing created one before we had computers (as we know them)

- An abstract model of a computer
- Turns out that all sufficiently powerful models of computation end up being “equivalent”:
 - ▶ Turing completeness
 - ▶ computable functions intuitively have a finite program, that completes in a finite number of steps to the result
 - ▶ almost all functions we deal with in math are computable (though maybe not efficiently)
 - ▶ there are a few that aren’t
- Turing machines have a few variants, but simplest has
 - ▶ a tape
 - ▶ a finite state machine that can write/read from the tape



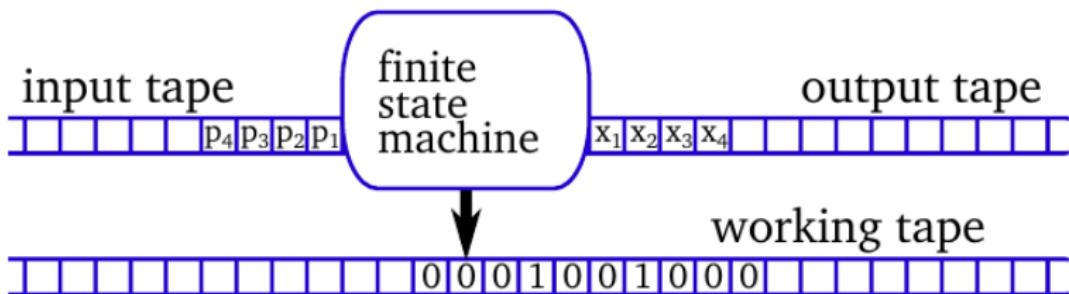
Simple Turing Machine

- a tape
 - ▶ a **tape** is an idealisation of computer memory
 - ▶ imagine a strip of paper on which we can write or erase some symbols (often binary 1s and 0s)
 - ▶ the tape can be moved back and forth so that the machine can write and read any point on the tape
- a finite state machine that can write/read from each tape
 - ▶ n states, plus “halt”
 - ▶ transition function has inputs of current state and current tape value
 - ▶ transition causes three outputs:
 - ★ can write over the current bit of the tape
 - ★ it can move the tape
 - ★ the state machine’s state can change
- running the machine means setting a set of tape values, and a starting state, and then allowing transitions until “halt” is reached

Our Turing Machine



- Ours will be just a little different (but equivalent)



- It's helpful to separate inputs and outputs from working memory
 - ▶ input tape (with the input **p** – the *program* – on it)
 - ▶ output tape (which we will write the output **x** on)
 - ▶ a working tape
 - ▶ a finite state machine that can write/read from each tape
- We'll call this a *universal computer*
 - ▶ measure complexity by the number of operations (transitions)

- A *deterministic* Turing machine is just what we described earlier
 - ▶ given a particular input, its output is “deterministic”
 - ▶ people have built them (almost)
 - ▶ standard computers are analogous
- a *non-deterministic* Turing machine: it can have a set of rules that give more than one action for a given situation.
 - ▶ in a given state, input a given symbol, perform both *A* and *B*
 - ▶ can think of it as
 - ★ getting to try both possibilities
 - ★ being able to guess the correct branch

Non-deterministic Turing machines don't exist, but are useful for describing algorithm complexity.



Complexity of Problems

Max Ward-Graham

<max.ward-graham@adelaide.edu.au>

<http://roughan.info/>

School of Mathematical Sciences,
University of Adelaide

February 21, 2022



Algorithm v Problem complexity

Remember that

- Algorithms have complexity but so do problems
- We usually think about algorithms as solving a *problem*
- Problems comprise a family of *instances*
- Algorithms have a complexity
- We often attribute a *problem* with the complexity of the **best algorithm**, on the **worst case instances**
- describe complexity with big-O notation, e.g., $O(n^2)$

General problem descriptions

Common types of (general) problem

decision : does a solution exist?

search : find a solution.

counting : how many solutions exist?

optimisation : find the best solution.

The distinction is arbitrary: e.g., we can solve a decision problem by searching for a solution, but its helpful in thinking about complexity.

General problem descriptions: example 1

Example

decision : is n prime?

search : find the prime factorization of n

counting : how many factors does the prime factorization of n have?

optimisation : find the factorisation which has the largest sum of factors.

General problem descriptions: example 2

Example

Given a set of numbers, e.g., $S = \{-7, -3, -2, 5, 8\}$

decision : does some subset of S add to give zero? Yes.

search : find a subset that adds to give zero: $\{-3, -2, 5\}$

counting : how many subsets of S add to give zero? 1?

optimisation : find the subset that adds to zero with the least members.

$\{-3, -2, 5\}$

Example

TSP (Travelling Sale-person's Problem) variants:

decision : is there a path visiting each city with distance less than k ?

search : find a path visiting each city with distance less than k .

counting : how many paths have distance less than k ?

optimisation : find the shortest path visiting all cities.

Definition (Decision problem)

A *decision problem* is a problem whose answer is YES or NO.

- Can be viewed as dividing problem instances in member and non-member instances
- Avoids issues related to needing to construct a valid solution
- Often solving a decision problem is not too different from solving a search (or optimisation, counting) problem e.g.,

$$a \times b$$

can be recast as a binary search on “is $a \times b \leq c$? ”

- NB: often, we solve decision problems by searching for a solution!
Think of the solution as a *certificate*.



Tractability

Problems that can be solved in theory (e.g., given large but finite time), but which in practice take too long for their solutions to be useful, are known as *intractable* problems

We can't trivially distinguish the intractable and tractable problems, so we often divide them by their asymptotic performance into

polynomial means there is an algorithm which takes time $\text{poly}(n)$ for some polynomial $p(n)$ on inputs of length n

- remember this is the worst case performance
- write it as $O(n^k)$ for some fixed k
- polynomial time algorithms are often treated as equivalent to tractable

exponential means it takes time *at least* $2^{\text{poly}(n)}$

- grows faster than any polynomial
- exponential algorithms are assumed to (generally) be intractable



P v NP

Max Ward-Graham
<max.ward-graham@adelaide.edu.au>
<http://roughan.info/>

School of Mathematical Sciences,
University of Adelaide

February 21, 2022



P and NP

Definition (P)

P is the set of *decision* problems that are *Polynomial time*, i.e., they can be solved by a deterministic Turing machine in polynomial time.

Definition (NP)

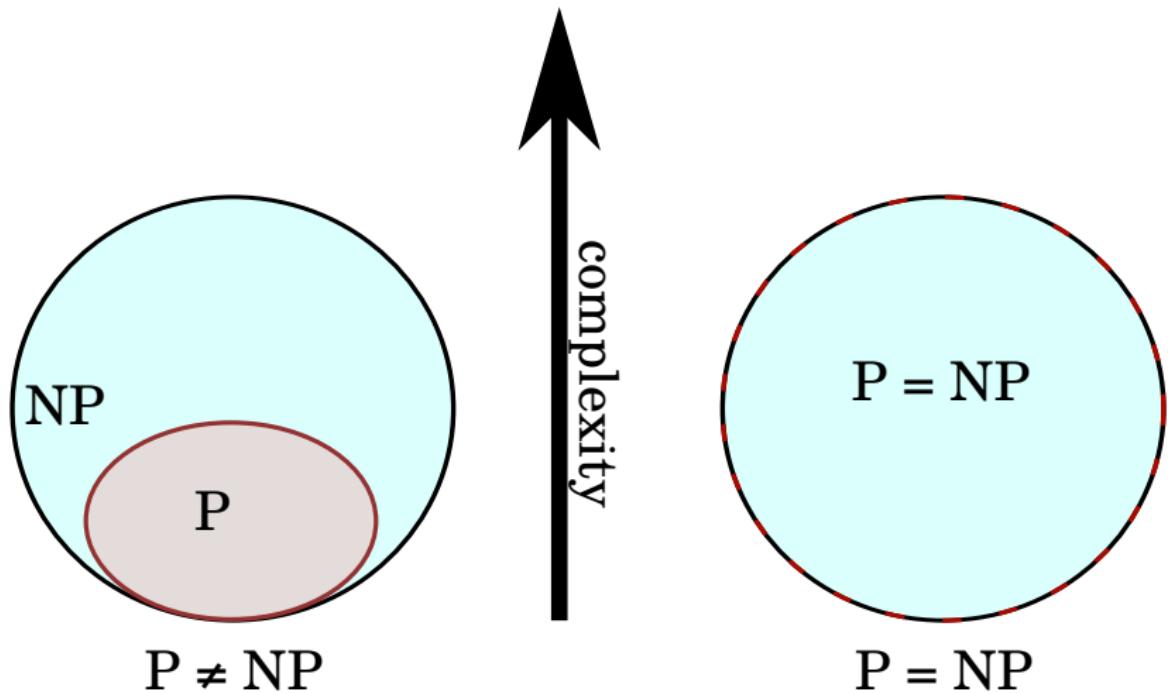
NP is the set of *decision* problems that are *Non-deterministic Polynomial time*, i.e., they can be solved by a non-deterministic Turing machine in polynomial time.

- NP does **NOT** mean Non-Polynomial
- It actually includes all polynomial-time decision problems, i.e.,
 $P \subseteq NP$
- We *don't know for sure* if it has anything else in it
 - ▶ Is $P = NP$?
 - ▶ Win \$1,000,000 if you can answer this

Euler Diagrams



THE UNIVERSITY
of ADELAIDE



Ways to think about NP

- NP problems have an efficient (polynomial time) *verifier*
 - ▶ computing the decision might be hard
 - ▶ but checking a YES decision is easy
 - ▶ e.g., is this Sudoku puzzle solvable?
 - ★ finding a solution can be very hard
 - ★ *verifying* a given solution is easier

assumes the YES result comes with a “proof certificate” (often a solution) which can be checked.

- They can be solved in polynomial time by a non-deterministic Turing machine using the following approach
 - ① Guess a solution
 - ② Check it

A non-deterministic Turing machine can make the right guess, so compute time is just the time to check the solution.

NP examples

- All P problems
- Traveling Salesperson problem
- Generalised Sudoku
- Graph isomorphism problem
- integer factorisation
- SAT

A very general class of decision problems is SAT

Definition (SAT)

A (Boolean) *satisfiability* (SAT) problem has n Boolean variables x_1, \dots, x_n and a Boolean formula ϕ involving the variables. The question is whether there is an assignment (of TRUE and FALSE) to the variables, such that $\phi(x_1, \dots, x_n) = \text{TRUE}$, i.e., we satisfy the formula.

Example

One variable x_1 and Boolean formula

$$\phi(x) = x_1 \wedge \neg x_1$$

where \wedge = AND and \neg = NOT, is *not satisfiable* because

$$\text{TRUE AND NOT TRUE} = \text{FALSE}$$

$$\text{FALSE AND NOT FALSE} = \text{FALSE}$$

so there is no value of x_1 that leads to $\phi(x_1) = \text{TRUE}$.

Example

Three variables x_1 , x_2 and x_3 and Boolean formula

$$\phi(\mathbf{x}) = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge \neg x_1$$

where

\vee = OR

\wedge = AND

\neg = NOT

is satisfied by $x_1 = FALSE$, $x_2 = FALSE$, and x_3 arbitrarily.

We *think* some problems in NP aren't in P

- We certainly know some problems in NP for which we have no polynomial-time algorithm *at present*
 - ▶ e.g., SAT
 - ▶ so we think these might be harder than P
 - ▶ a problem is called *NP-hard* if it is at least as hard as the hardest problem in NP
 - ▶ we'll define formally in a moment
- If $P \neq NP$ then NP-hard problems cannot be solved in polynomial time.

We *think* some problems in NP aren't in P

If $P = NP$, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in “creative leaps,” no fundamental gap between solving a problem and recognizing the solution once it’s found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss; everyone who could recognize a good investment strategy would be Warren Buffett. It’s possible to put the point in Darwinian terms: if this is the sort of universe we inhabited, why wouldn’t we already have evolved to take advantage of it?

Scott Aaronson

<http://www.scottaaronson.com/blog/?p=122>



NP-hard definitions

Definition

A problem A can be reduced to B if we could solve A using the algorithm that solves B as a subroutine.

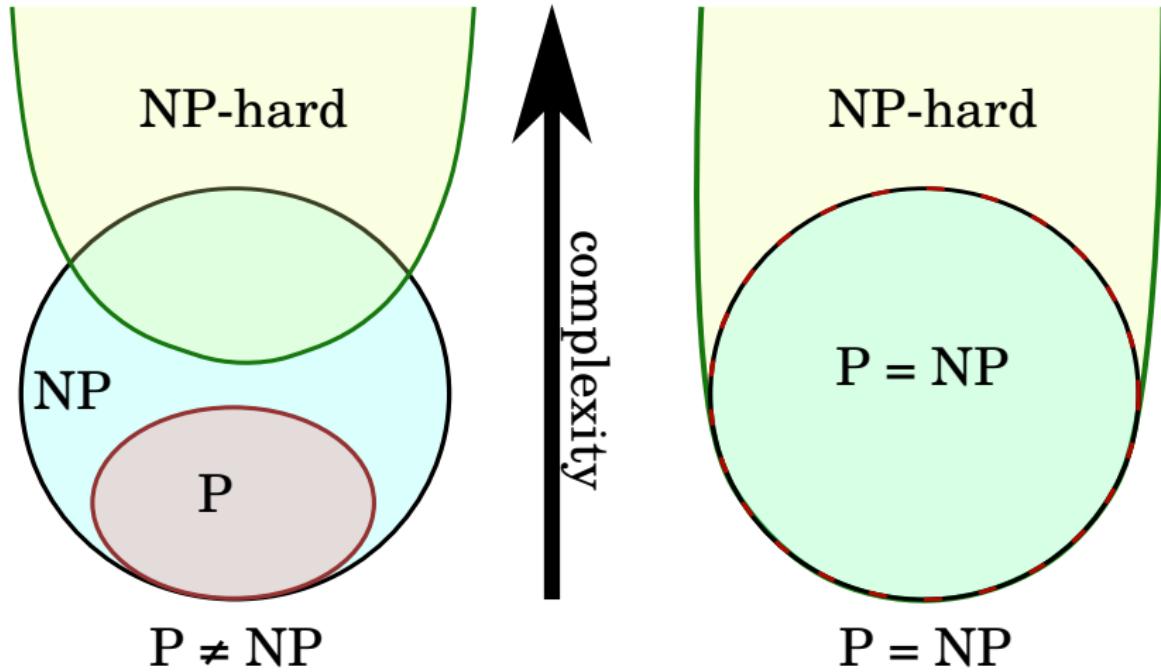
- If we have a polynomial time reduction (that is one that can be done in polynomial time, excluding the time in the subroutine for solving B) then we can efficiently convert one problem into the other.
- So A is (up to polynomial factors) no more difficult than B

Definition (NP-hard)

A problem H is NP-hard if every problem L in NP can be *reduced* in polynomial time to H .

- So H is at least as hard as any L in NP.
- Note that an NP-hard problem isn't necessarily in NP!

Euler Diagrams



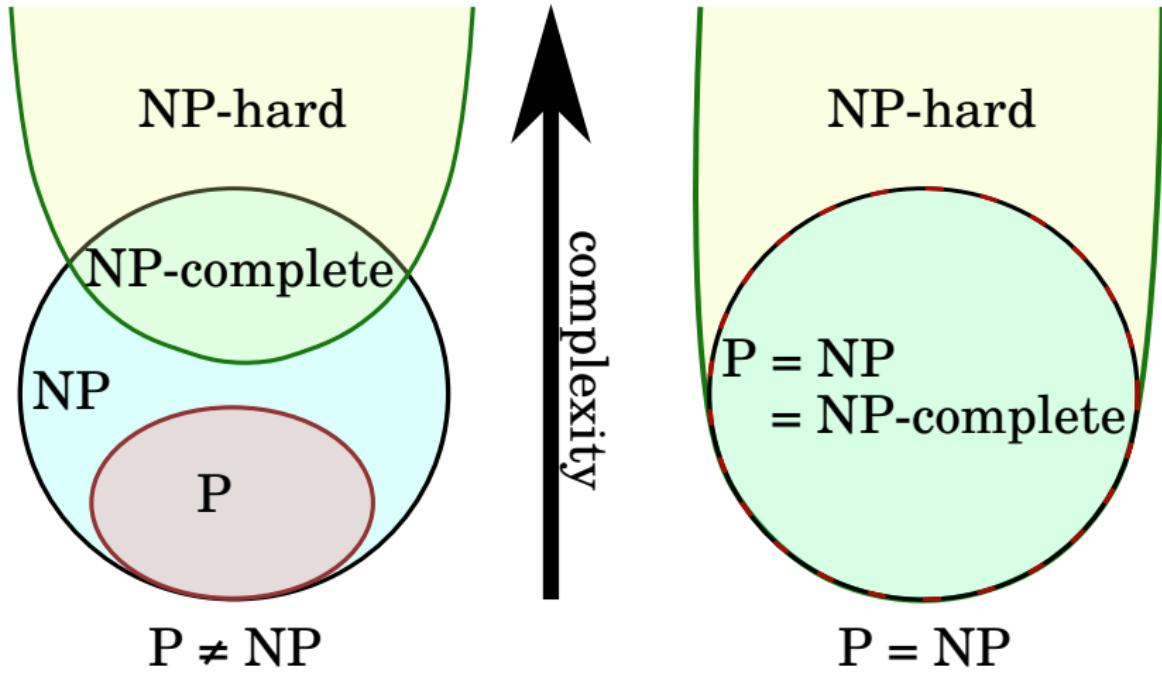


Definition (NP-complete)

A problem that is in NP, and in NP-hard is called NP-complete.

- A problem p in NP is NP-complete if every other problem in NP can be reduced to p in polynomial time.
- Cook's theorem: the Boolean satisfiability problem (SAT) is NP-complete
 - ▶ having just one NP-complete problem lets us show lots of problems are NP-complete!

Euler Diagrams



Example NP-complete problems

- SAT (and many variants)
- *Binary linear programming*
- Set covering
- Hamiltonian circuit
- Graph colouring
- Bin-packing and Knapsack
- TSP problem
- Many others

All the above have a decision variant.

- If a decision problem is NP-complete, then its optimisation version is NP-hard
- A few problems are not in P or NP-complete
 - ▶ The graph isomorphism problem
 - ★ is graph G_1 isomorphic to G_2
 - ★ its in NP
 - ★ its suspected to be neither in P or NP-complete
 - ★ very recently a *quasi-polynomial* time algorithm was found

call these NPI = NP-intermediate

- ▶ in NP, but not in P or NP-complete
- There are NP-hard problems that are not NP-complete
 - ▶ e.g., the halting problem
 - ★ given a program and its input, will it run forever?
 - ★ its undecidable (so not in NP)
 - ★ SAT can be reduced to the halting problem by writing Turing machine program that tries all values

Misconceptions

- NP-complete problems are not the “hardest”
 - ▶ they are in NP – some problems aren’t!
 - ★ some problems can’t even be verified in polynomial time
 - ▶ verify a decision version of chess: given a board state, can you win the game?
- Not all *instances* of NP-complete problems are hard
 - ▶ many (even most) instances of some NP-complete problems can be solved in polynomial time
 - ▶ complexity refers to worst case
- Problems with an exponential number of possibilities are not all NP-complete
 - ▶ counter-example: shortest paths is solvable in $O(n \log n)$ time



Takeaways

- We talked about complexity *classes*

- ▶ P
- ▶ NP
- ▶ NP-complete
- ▶ NP-hard

we don't know if $P = NP$

- *At the moment*, we can't solve an NP-complete problem in **guaranteed polynomial time**
 - ▶ integer programming is, in general, NP-complete
 - ★ some of these problems are currently intractable
 - ★ but some restricted subsets of integer programming problems might have polynomial time algorithms
 - ★ others might have good approximations
 - ▶ in general, though, we are going to have to be a bit more clever when tackling integer programming problems
 - ★ there is no "one-size-fits-all" like the Simplex for LPs

4 Integer Programming Problems

4.1 Native Integer Variables

Ashley Dennis-Henderson

School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



Integer variables

A long time ago we looked at a manufacturing scheduling problem

$$\begin{aligned} \max z &= 13x_1 + 12x_2 + 17x_3 \\ \text{s.t.} \quad & 2x_1 + x_2 + 2x_3 \leq 225 \\ & x_1 + x_2 + x_3 \leq 117 \\ & 3x_1 + 3x_2 + 4x_3 \leq 420 \\ & x_1 \geq 0, x_2 \geq 0, x_3 \geq 0 \end{aligned}$$

x_1 = the number of desks;

x_2 = the number of chairs; and,

x_3 = the number of bed frames, made per time period.

Shouldn't we ensure that x_1 , x_2 and x_3 are integers?!

Allocation problem – 1

Example (Knapsack problem)

A hiker can choose from the following items when packing a knapsack:

Item	1 chocolate	2 raisins	3 camera	4 jumper	5 drink
w_i (kg)	0.5	0.4	0.8	1.6	0.6
v_i (value)	2.75	2.5	1	5	3.0
v_i / w_i	5.5	6.25	1.25	3.125	5

However, the hiker cannot carry more than 2.5 kg all together.

Objective: choose the number of each item to pack in order to maximise the total value of the goods packed, without violating the mass constraint.

Example (Knapsack problem – Formulation)

Let x_i bet the number of copies of item i to be packed, such that $x_i \geq 0$ and integer (cannot pack 1/2 a jumper!).

$$\max v = 2.75x_1 + 2.5x_2 + x_3 + 5x_4 + 3x_5$$

$$\text{s.t.} \quad 0.5x_1 + 0.4x_2 + 0.8x_3 + 1.6x_4 + 0.6x_5 \leq 2.5$$

$$x_i \geq 0 \quad \text{and integer.}$$

Allocation problems



THE UNIVERSITY
of ADELAIDE

MY HOBBY:

EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS

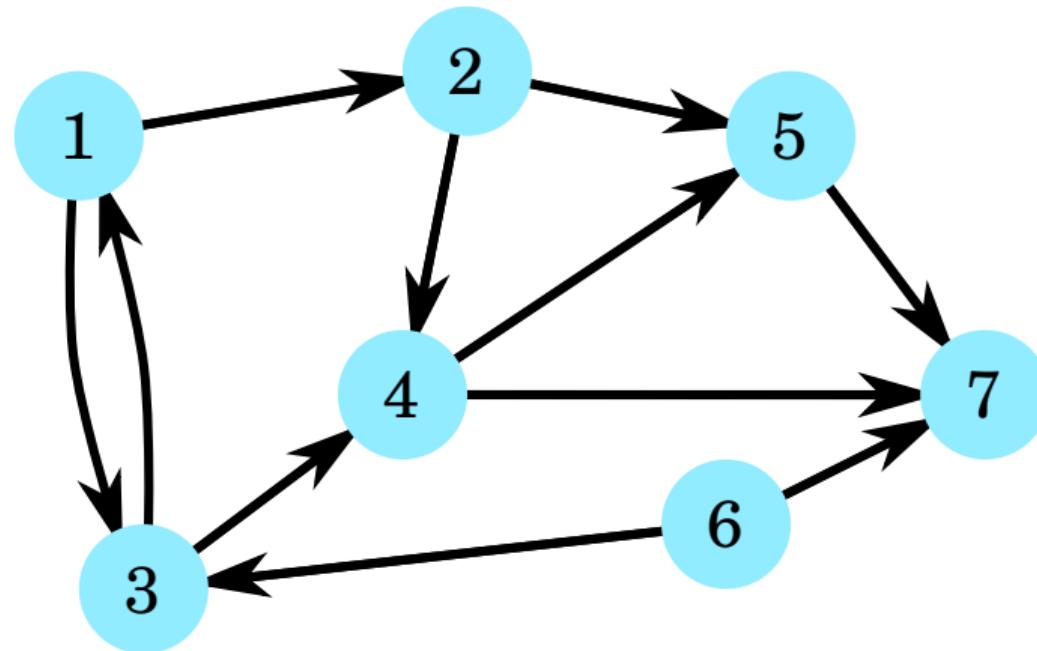
CHOTCHKIES RESTAURANT	
~~ APPETIZERS ~~	
MIXED FRUIT	2.15
FRENCH FRIES	2.75
SIDE SALAD	3.35
HOT WINGS	3.55
MOZZARELLA STICKS	4.20
SAMPLER PLATE	5.80
~~ SANDWICHES ~~	
BARBECUE	6.55



Network Problems

Many optimisation problems are related to a *Network* or *Graph*

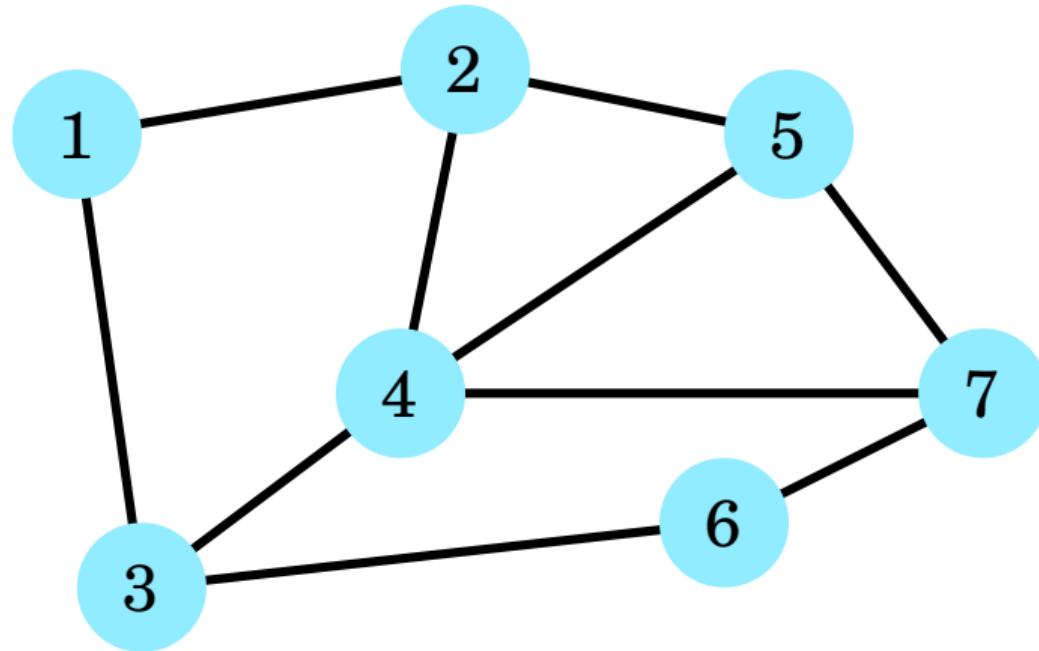
Consider a set of nodes (or vertices) N and a set of directed links (or edges) L between those nodes. These directed links then give us a *directed network* or *directed graph* $G(N, L)$ like that below



Network Problems

Many optimisation problems are related to a *Network*

Consider a set of nodes (or vertices) N and a set of undirected links (or edges) L between those nodes. These directed links then give us an *undirected network* $G(N, L)$ like that below



Definition (undirected graph)

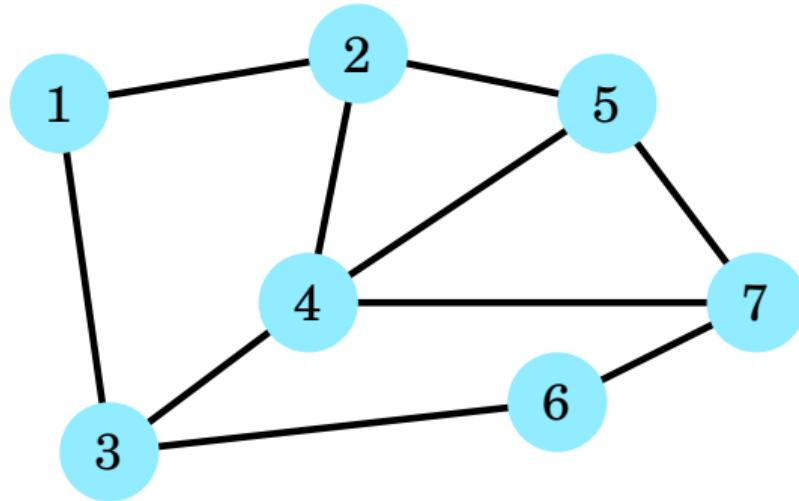
An *undirected graph* has edges (or links) that are unordered pairs of nodes $\{i, j\} \in L$, $i, j \in N$, meaning node i is *adjacent* to node j and visa versa.

Definition (directed graph)

A *directed graph* has edges (or arcs) that are ordered pairs of nodes (i, j) , meaning node i is *adjacent* to node j .

Graph terminology

Example (Undirected graph)



$G(N, E)$ where N is the set of nodes, and E is the set of edges

$$N = \{1, 2, 3, 4, 5, 6, 7\}$$

$$E = \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 4), (3, 6), (4, 5), (4, 7), (5, 7), (6, 7)\}$$

- ▶ A *walk* is an ordered list of nodes i_1, i_2, \dots, i_t such that, in an undirected graph, $\{i_k, i_{k+1}\} \in L$, or, in a directed graph, $(i_k, i_{k+1}) \in L$ for $k = 1, 2, \dots, t - 1$.
- ▶ A *path* is a walk where the nodes i_1, i_2, \dots, i_k are all distinct.
 - ▶ A graph is *connected* if there is a path connecting every pair of nodes.
- ▶ A *cycle* is a walk where the nodes i_1, i_2, \dots, i_{k-1} are all distinct, but $i_1 = i_k$.
 - ▶ A directed graph is *acyclic* if it contains no cycles.
 - ▶ we call it a *DAG* = Directed Acyclic Graph



Example (The Travelling Salesperson Problem (TSP))

Given a set of towns, $i = 1, \dots, n$, and links (i, j) between the towns. The links each have a specified length, given by a distance matrix

$$D = \begin{bmatrix} & 1 & 2 & \cdots & j & \cdots & n \\ 1 & & & & & \vdots & \\ 2 & & & & & \vdots & \\ \vdots & & & & & \vdots & \\ i & & \cdots & \cdots & \cdots & d_{ij} & \\ \vdots & & & & & & \\ n & & & & & & \end{bmatrix}$$

Objective: construct a directed cycle of minimum total distance going through each town exactly once.



TSP

The decision is, basically, which links do we choose to use in the tour.

$$x_{ij} = \begin{cases} 1 & \text{if link } (i,j) \text{ is chosen} \\ 0 & \text{if link } (i,j) \text{ is not chosen} \end{cases}$$

then the ILP (Integer Linear Program) formulation is

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

$$\text{s.t.} \quad \sum_{j=1}^n x_{ij} = 1, \quad \forall i = 1, \dots, n \quad (\text{only one link from } i)$$

$$\sum_{i=1}^n x_{ij} = 1, \quad \forall j = 1, \dots, n \quad (\text{only one link to } j)$$

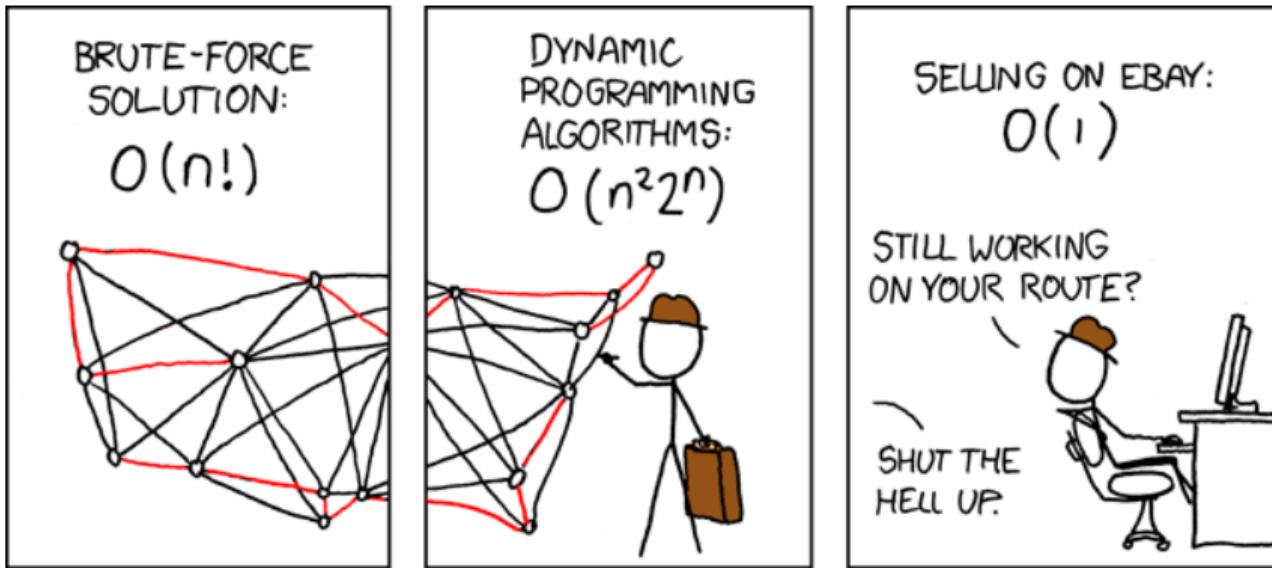
$$\sum_{i \in S} \left(\sum_{j \in S^c} x_{ij} \right) \geq 1, \quad \forall S \subset N \quad (\text{connectedness})$$

$$x_{ii} = 0 \text{ or } 1 \quad \text{for all } i, i$$

Network Problems



THE UNIVERSITY
of ADELAIDE



<http://xkcd.com/399/>



Graph and network problems come up a lot!

Example (Shortest path problem)

(Differs from a TSP in that it does not look for a *tour* through all nodes, but rather a path from one node to another.)

Find a minimum length path through a network $G(N, L)$, from a specified source node, say 1, to a specified destination node n , where each link $(i, j) \in L$ has an associated length, d_{ij} .

Example (Maximum flow problem)

Given a network with a single source node (generator of traffic) and a sink (attractor of traffic); network has directed links with links (i, j) having an upper capacity of u_{ij} .

We have no cost for unit flows, just bounds on the capacities of the links and we wish to send the maximum flow from one specified node to another.

4 Integer Programming Problems

4.2 Supplementary Integer Variables

Ashley Dennis-Henderson

School of Mathematical Sciences

The University of Adelaide



THE UNIVERSITY
of ADELAIDE

Sometimes the original variables in the problem aren't variables, but we have to add in extra, artificial variables for some other reason

- ▶ disjoint objective functions
 - ▶ fixed-cost problems
- ▶ disjunctive constraints

Example (Production planning)

Here we have N products, where the production cost for product j ($j = 1, \dots, N$) consists of a fixed setup cost $K_j \geq 0$ and a variable cost c_j that depends on the number of copies of item j produced. That is, the cost associated with producing x_j copies of item j is

$$C_j(x_j) = \begin{cases} K_j + c_j x_j & x_j > 0, \\ 0 & x_j \leq 0. \end{cases}$$

Objective: minimise total cost, $\min z = \sum_{j=1}^N C_j(x_j)$, subject to $x_j \geq 0$, and any other constraints on supplies, orders, etc.

Note that variables x_j are not necessarily integer!



Fixed costs problems

The problem is *non-linear*, because of the discontinuity at $x_j = 0$ in $C_j(x_j)$, which can be removed by introducing j new variables y_j , where

$$y_j = \begin{cases} 1 & x_j > 0 \\ 0 & x_j = 0. \end{cases}$$

Then $C_j(x_j) = K_j y_j + c_j x_j$, and the problem can be formulated as

$$\min z = \sum_{j=1}^N (K_j y_j + c_j x_j)$$

$$\text{s.t. } x_j \geq 0$$

$$y_j = 0 \text{ or } 1$$

$$x_j \leq M y_j \quad (+\text{other constraints, e.g., order sizes})$$

where $M \geq$ upper bound on all x_j .

- (a) The original problem didn't have integer variables – these arose as artificial variables to keep the problem linear.
- (b) This is a *mixed* Integer, Linear Program (*ILP*), because some variables are continuous and some are integers.
- (c) If $x_j = 0$ then minimising z requires $y_j = 0$.
- (d) For all production of item j given by $x_j > 0$ means that we have to set $y_j > 0$ to satisfy the new constraints $x_j \leq My_j$ for each item j .

Definition

*A constraint in the form of A and B is called **conjunctive**.*

Most of our earlier constraints are already conjunctive – we require *all* of them to be true in the feasible region.

Definition

*A constraint in the form of A or B is called **disjunctive**.*

These might be used to make it possible to create multiple (separate) feasible regions.

Note, mathematicians sometimes denote AND by \wedge and OR by \vee

Disjunctive constraints



THE UNIVERSITY
of ADELAIDE

Example (Sorta Auto)

Sorta Auto is considering manufacturing 3 types of cars: compact, medium and large. Resources required and profits yielded by each type of car are

	Compact	Medium	Large
Steel required (tonnes)	0.5	1	3
Labour required (hrs)	30	25	40
Profit (\$1,000's)	3	5	8

The amount of steel available is at most 2,000 tonnes and at most 60,000 hrs of labour are available. **Production of any car type is feasible only if at least 1,000 cars of that type are made.**

Objective: maximise profit, by determining how many of each type to produce, while satisfying the constraints.



Disjunctive constraints

Variables

$$\begin{cases} x_1 = \text{number of compact cars produced} \\ x_2 = \text{number of medium size cars produced} \\ x_3 = \text{number of large size cars produced.} \end{cases}$$

Then the objective function is

$$\max \quad 3x_1 + 5x_2 + 8x_3$$

with constraints:

$$x_1, x_2, x_3 \geq 0, \text{ and integer}$$

$$0.5x_1 + x_2 + 3x_3 \leq 2000$$

$$30x_1 + 25x_2 + 40x_3 \leq 60,000$$

$$\text{either } x_1 = 0 \text{ or } x_1 \geq 1000$$

$$\text{either } x_2 = 0 \text{ or } x_2 \geq 1000$$

$$\text{either } x_3 = 0 \text{ or } x_3 \geq 1000.$$



Disjunctive constraints

Instead of either $x_i = 0$ or $x_i \geq 1000$ introduce binary variable y_i such that

$$\begin{cases} x_i \leq M_i y_i & \text{(a)} \\ 1000 - x_i \leq M_i(1 - y_i) & \text{(b)} \\ y_i \in \{0, 1\} & \text{(c)} \end{cases}$$

Notice that

- ▶ if $x_i > 0$, then $y_i = 1$ by both constraints (a) and (c)
 $\Rightarrow x_i \geq 1000$ because of constraint (b)
- ▶ $y_i = 0$ then $x_i = 0$ by constraint (a)

and so we get the desired result!

Disjunctive constraints in general

In general,

“either $f(\mathbf{x}) \leq 0$ or $g(\mathbf{x}) \leq 0$ ”

is equivalent to

“if $f(\mathbf{x}) > 0$ then $g(\mathbf{x}) \leq 0$ ”.

Introduce 0–1 variable y and the constraints

$$g(\mathbf{x}) \leq M(1 - y), \quad f(\mathbf{x}) \leq My$$

where M is a large, positive number.

- ▶ if $f(\mathbf{x}) > 0$, then $y > 0$ and so $y = 1$, giving $g(\mathbf{x}) \leq 0$.
- ▶ if $y = 0$ then $f(\mathbf{x}) \leq 0$.

4 Integer Programming Problems

4.3 Integer Programming: Naïve Solutions

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

Naïve solutions

1. Exhaustive search
2. Approximate with a LP

- ▶ When first considering (*LP*)s we suggested solving them by enumerating all basic solutions, determining which were feasible ones, and then choosing the one which gave the optimal evaluation of the objective function.
 - ▶ we saw this was a bad idea (there are too many possibilities)
 - ▶ but we have restricted the space now, maybe exhaustive works?
- ▶ Approach
 - ▶ enumerate every potential solution,
 - ▶ check its feasibility, and
 - ▶ from amongst the feasible ones, choose the one that gives the optimal value of the objective function.

- ▶ Consider a simple binary linear program with n binary variables
 - ▶ there are 2^n possible solutions
 - ▶ that is $O(\exp(n))$so an exhaustive search is hopeless for even moderate n .
- ▶ Consider the TSP
 - ▶ N towns to visit
 - ▶
$$\frac{(N - 1)!}{2} \quad \text{different tours.}$$

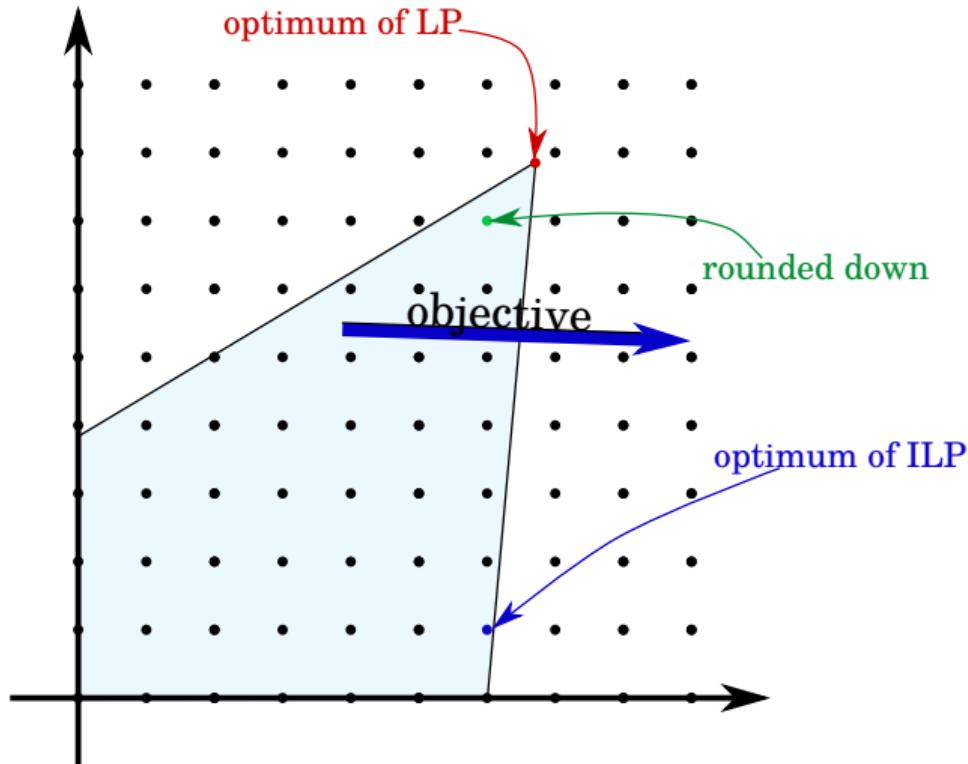
For example, if there were $N = 100$ towns to visit, this is then 4.6663×10^{157} different tours.

Exhaustive searches are a really bad idea except for toy problems.

1. Idea: drop the integrality constraint and solve the resulting LP
 - ▶ we know how to solve LPs efficiently
 - ▶ sounds a bit like rounding off, so there might be some errors, but how big can they be?
2. We call this *relaxation*
 - ▶ in general relaxation means loosening up some constraint
 - ▶ here we relax the integer constraint

Relaxation (of integrality)

We can guess that rounding might not be optimal, but it can be **far** away from optimum.



It gets worse in higher dimensions.

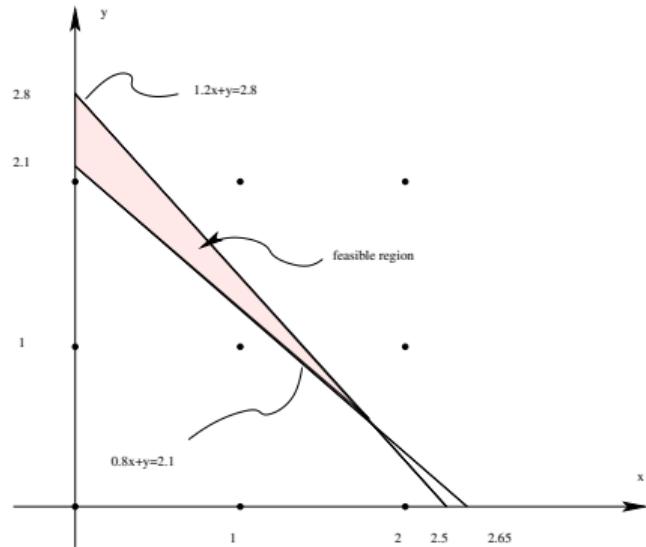
Relaxation (of integrality)

Solving a relaxed (*LP*) may lead to different solutions

- ▶ they *might not even be feasible*

Example

$$\begin{aligned} \max z &= 3x + y \\ \text{s.t. } 0.8x + y &\geq 2.1 \\ 1.2x + y &\leq 2.8 \\ x, y &\geq 0 \text{ (and integral.)} \end{aligned}$$



So relaxation is a crude tool, but it can be useful if used carefully (we will see how later on).



Methods we will look at . . .

- ▶ Greedy algorithms
 - (heuristics)
- ▶ Branch and bound
 - (enumeration with pruning)
- ▶ General purpose heuristics
 - (genetic algorithms)

And we'll look at some toolkits that help.

- ▶ Lot's of real problems have *integer* variables
- ▶ Integer programming is much more than just linear programming with integer variables.
- ▶ Even if variables are continuous, other parts of the problem need extra integer variables
 - ▶ disjunctive constraints (either or)
 - ▶ discontinuous objectives
- ▶ Naïve solutions to ILPs aren't a great idea

4 Integer Programming Problems

4.4 Integer Programming: Matlab

Ashley Dennis-Henderson

School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

Similar to the `linprog` command in Matlab for linear programs that have variables which can take on real solutions, there exists a command `intlinprog` for those linear programs which have (some or all) variables constrained to be *integer*.¹

That is, `intlinprog` solves linear programming problems of the form

$$\min_{\mathbf{x}} \mathbf{f}^T \mathbf{x}, \quad \text{such that} \quad \begin{cases} \mathbf{A} \mathbf{x} \leq \mathbf{b} \\ \mathbf{A}_{eq} \mathbf{x} = \mathbf{b}_{eq} \\ \text{some } x_i \text{ integer} \end{cases}$$

where, \mathbf{f} , \mathbf{b} , and \mathbf{b}_{eq} are vectors, \mathbf{A} and \mathbf{A}_{eq} are matrices, and some of the the variables are required to be integers.



MATLAB intlinprog example

Commands: for Binary program below

```
> f = [-9; -5; -6; -4];  
> A = [6,3,5,2; 0,0,1,1; -1,0,1,0; 0,-1,0,1];  
> b = [9; 1; 0; 0];  
> Aeq = [];  
> beq = [];  
> intcon = [1,2,3,4];  
> ub = ones(4,1);  
> lb = zeros(4,1);  
> x = intlinprog(f,intcon,A,b,Aeq,beq,lb,ub)
```

Output:

```
x = 1  
    1  
    0  
    0
```



When to use intlinprog

- ▶ Don't cheat
 - ▶ use this to check solutions
 - ▶ but solve them the way required in your assignments
- ▶ Yes, you can use it in your project
 - ▶ but display understanding
 - ▶ show alternatives
- ▶ In general
 - ▶ you still need to construct matrices and vectors which is awkward
 - ▶ you need to write every constraint, even if they fit a pattern
 - ▶ you still need explicit (closed form) constraints and objective functions
 - ▶ Matlab doesn't tell you much about how it does it, and what its limitations are
 - ▶ we know the problem might be NP-complete, so this could be an issue



OPTIMISATION & OPERATIONS RESEARCH

4 Integer Programming Problems

4.5 Heuristics & the Greedy Heuristic

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



So far in this course, we have used *algorithms*:

- ▶ e.g., Simplex

An algorithm precisely specifies a recipe for a computation.

A *heuristic* is a rule of thumb or an educated guess

- ▶ in our context they are rules that might lead to *good* solutions
- ▶ often based on simple intuition
- ▶ sometimes easier to code up
- ▶ often used when there isn't a fast-enough algorithm known



A heuristic usually leads to an “algorithm”

In optimisation we often make the distinction that

- ▶ an *algorithm* is guaranteed to find the optimal solution
- ▶ a *heuristic* makes no guarantees
 - ▶ though we hope it will find a good solution
 - ▶ and it may find the optimal solution

We might even talk about a *meta-heuristic*, which is a general idea that can be converted into a heuristic for a particular problem, which leads to an “algorithm”, sometimes an exact one, and other times not.

- ▶ *greedy* meta-heuristic
 - ⇒ Dijkstra's algorithm on shortest paths problem

Iterate

- ▶ Create a set of feasible candidates/choices
 - ▶ local “move” from current solution
 - ▶ partial solutions (don’t need to know all of the variables at once)
- ▶ Rate candidates by value (in terms of the objective)
- ▶ Choose the best

Stop when you run out of choices

- ▶ Intuition
 - ▶ often an optimal solution has a few important pieces, and the rest are “noise”
 - ▶ greedy gets the important bits first
 - ▶ sometimes this is even guaranteed to find the optimal solution
- ▶ Bad bits
 - ▶ locally good decisions can be globally bad
 - ▶ method is short-sighted
 - ▶ go down a dead end and there isn’t any way to go back

Examples

- ▶ Knapsack problem
- ▶ Coin Changing
- ▶ TSP
- ▶ Huffman coding
- ▶ Shortest paths

4 Integer Programming Problems

**4.6 Greedy heuristic example 1:
knapsack problem**

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

Knapsack problem [KV00]

Example (Knapsack problem)

A hiker can choose from the following items when packing a knapsack:

Item	1 chocolate	2 raisins	3 camera	4 jumper	5 drink
w_i (kg)	0.5	0.4	0.8	1.6	0.6
v_i (value)	2.75	2.5	1	5	3.0
v_i / w_i	5.5	6.25	1.25	3.125	5

However, the hiker cannot carry more than 2.5 kg all together.

Objective: choose the number of each item to pack in order to maximise the total value of the goods packed, without violating the mass constraint.



Knapsack problem in general

Integral knapsack problem

- ▶ we have a knapsack (backpack) which can take weight W
- ▶ we want to fit as much useful stuff into it as possible
 - ▶ maximize the value of the items contained in the knapsack
- ▶ each item i
 - ▶ has a weight w_i
 - ▶ has a value v_i (we want to maximise total value)
- ▶ we have one *indicator* variable, z_i , for each item
 - ▶ if we include the item, we say $z_i = 1$
 - ▶ otherwise $z_i = 0$
- ▶ summarizing

$$\max \left\{ \sum_i v_i z_i \mid \sum_i w_i z_i \leq W, z_i = 0 \text{ or } 1 \right\}$$

- ▶ The knapsack decision problem is NP-complete
 - ▶ the decision problem is:
"Can we find an allocation with value at least V and weight less than W?"
- ▶ The knapsack optimisation problem (described above) is NP-hard
 - ▶ it is at least as hard as the decision problem
 - ▶ there are no known polynomial-time checks for optimality

Greedy knapsack heuristic (due to Dantzig)

1. Calculate the value to weight ratio v_i/w_i
2. Sort the items in decreasing order
3. For $i = 1..n$
 - 3.1 if there is room for item i , add it

Sorting is $O(n \log n)$, so this component dominates performance.

Very common (in different forms)

- ▶ fractional (allows fractions of items)
- ▶ unbounded (multi-items, i.e., $z_i \in \mathbb{Z}^+$)
- ▶ multiple constraints: e.g., volume and weight
- ▶ multiple knapsacks \Rightarrow Bin-packing problem

4 Integer Programming Problems

**4.7 Greedy heuristic example 2:
coin chaning problem**

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

Coin Changing Problem

Problem: given possible coins and banknotes pay an amount $\$z$ using the smallest number of coins and banknotes.

Example

Australian currency:

banknotes \$100, \$50, \$20, \$10, \$5;

coins \$2, \$1, 50c, 20c, 10c, 5c.

So \$105.50 can be paid (minimally) using \$100 + \$5 + 50c

General problem: given coins and banknotes of value c_i for $i = 1, \dots, n$, then solve

$$\min \left\{ \sum_{i=1}^n x_i \mid \sum_{i=1}^n x_i c_i = z, x_i \in \mathbb{Z}^+ \right\}$$

Where, x_i is the number of value c_i coins/banknotes.

Coin Changing Greedy Solution

Input: z , and (decreasing) coin values
 $c = (100, 50, 20, 10, 5, 2, 1, 0.5, 0.2, 0.1, 0.05)$

Output: $x^* \in \mathbb{Z}^n$

```
1  $i \leftarrow 1$ 
2  $x_i \leftarrow 0$ 
3 while  $z > 0$  do
4   if  $c_i \leq z$  then
5      $x_i \leftarrow x_i + 1$ 
6      $z \leftarrow z - c_i$ 
7   else
8      $i \leftarrow i + 1$ 
9      $x_i \leftarrow 0$ 
10  end
11 end
```

Algorithm 3: Greedy Coin Change

Coin Changing Greedy Solution

Example

Given currency $\mathbf{c} = (4, 3, 1)$ and $z = 6$

1. $i = 1, c_i = 4$
 - 1.1 $x_1 = 1, z = 2$
2. $i = 2, c_i = 3$
 - 2.1 $x_2 = 0, z = 2$
3. $i = 3, c_i = 1$
 - 3.1 $x_3 = 1, z = 1$
 - 3.2 $x_3 = 2, z = 0$

So greedy gives $\mathbf{x} = (1, 0, 2)$

Actual optimal solution is $\mathbf{x} = (0, 2, 0)$

- ▶ There are smarter ways to do this
 - ▶ add all of a particular coin you can in one go
 - ▶ complexity is $O(n)$, where n is number of coins
 - ▶ but I like the recursive nature of the above
- ▶ For *canonical* coins systems, greedy is optimal

Definition (Canonical Coin System)

A *coin system* is canonical if the greedy solution is always optimal.

- ▶ US coins are canonical
- ▶ Conditions to check if a system is canonical are involved
- ▶ We could treat design of coin system as an optimisation in itself
- ▶ Frobenius coin problem is find the largest amount that *cannot* be obtained using only specified coins.
 - ▶ see also postage stamp problem and McNugget problem

4 Integer Programming Problems

**4.8 Greedy heuristic example 3:
travelling salesperson**

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



Travelling salesperson problem (TSP)

Given a set of towns, $i = 1, \dots, n$, and distances between the towns

$$D = [d_{ij}] = \begin{bmatrix} & 1 & 2 & \cdots & j & \cdots & n \\ 1 & & & & & \vdots & \\ 2 & & & & & \vdots & \\ \vdots & & & & & \vdots & \\ i & & \cdots & \cdots & \cdots & & d_{ij} \\ \vdots & & & & & & \\ n & & & & & & \end{bmatrix}$$

Objective: construct a directed cycle of minimum total distance going through each town exactly once.



TSP Formulation

The decision is, basically, which links do we choose to use in the tour.

Letting $x_{ij} = \begin{cases} 1 & \text{if link } (i,j) \text{ is chosen} \\ 0 & \text{if link } (i,j) \text{ is not chosen} \end{cases}$, then we have

$$(ILP) \quad \begin{aligned} \min d &= \sum_{i=1}^n \sum_{j=1}^n d_{ij} x_{ij} \\ \text{s.t.} \quad \sum_{j=1}^n x_{ij} &= 1, \quad \forall i = 1, \dots, n \quad (\text{only one link from } i) \\ \sum_{i=1}^n x_{ij} &= 1, \quad \forall j = 1, \dots, n \quad (\text{only one link to } j) \\ \sum_{i \in S} \left(\sum_{j \in S^c} x_{ij} \right) &\geq 1, \quad \forall S \subset N \quad (\text{connectedness}) \\ x_{ij} &= 0 \text{ or } 1 \quad \text{for all } i, j \end{aligned}$$

This is an example of a classic 0–1 (Binary) Integer Linear Program.
The equations are linear, but the variables are integer (here binary).

- ▶ The ILP describes links by n^2 binary variables x_{ij}
- ▶ We can write the same information much more concisely by describing the *tour* as a *permutation* of the integers $1, \dots, n$.
 - ▶ a permutation is just lists the cities in some order
 - ▶ it's hard to write this formulation in ILP, but it can be easier to work with when programming
 - ▶ we often see this

- ▶ Start at an arbitrary node (usually city 1)
- ▶ Choose the nearest town i_1 to city 1 as the second town
- ▶ Choose the nearest town i_2 to city i_1 as the third town
- ▶ And so on ...

Greedy doesn't work very well for the TSP, but it can provide an initial solution, which we can then improve.

4 Integer Programming Problems

**4.9 Greedy heuristic example 4:
coding**

Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE



Coding

- ▶ We have a “text” made up of a series of messages, or symbols

a, b, c, d

- ▶ We know the PMF (prob. mass function) of the messages

$P(a), P(b), P(c), P(d)$

- ▶ We want to have a binary code for each symbol, e.g.,

$a \leftrightarrow 00$

$b \leftrightarrow 01$

$c \leftrightarrow 10$

$d \leftrightarrow 11$

- ▶ We want to minimise the average number of bits

- ▶ in the example, the average is 2
- ▶ can we do better?



Coding

- ▶ Imagine

$$P(a) = 1/2$$

$$P(b) = 1/4$$

$$P(c) = 1/8$$

$$P(d) = 1/8$$

- ▶ And we use the code

$$a \leftrightarrow 0$$

$$b \leftrightarrow 10$$

$$c \leftrightarrow 110$$

$$d \leftrightarrow 111$$

Average message length

$$\text{bits per word} = 1\frac{1}{2} + 2\frac{1}{4} + 3\frac{1}{8} + 3\frac{1}{8} = \frac{7}{4} < 2$$

How should we minimise code length in general?

Formalised coding problem



THE UNIVERSITY
of ADELAIDE

Objective: minimise the average code length

$$L = E[\ell] = \sum_{k=1}^m \ell_k p_k$$

where

ℓ_k = length of k th code word

p_k = probability of k th code word

Subject to the Kraft inequality (won't go into this here, but it's needed to make it possible to decode)

1. We are building a tree
2. Start with each symbol in Ω as a leaf of the tree.
3. Repeat the following rule
 - 3.1 merge the two current nodes with the lowest probabilities to get a new node of the tree
4. The root is when we get a probability 1.

Huffman coding example 1

X	Probability
a	0.25
b	0.25
c	0.2
d	0.15
e	0.15

Huffman coding example 1



X	Probability
a	0.25 → 0.25
b	0.25 → 0.25
c	0.2 → 0.2
d	0.15 → 0.3
e	0.15 → 0.3

Huffman coding example 1



X	Probability
a	0.25 → 0.25 → 0.25
b	0.25 → 0.25 → 0.45
c	0.2 → 0.2
d	0.15 → 0.3 → 0.3
e	0.15

Huffman coding example 1



X	Probability
a	0.25 → 0.25 → 0.25 → 0.55
b	0.25 → 0.25 → 0.45 → 0.45
c	0.2 → 0.2
d	0.15 → 0.3 → 0.3
e	0.15

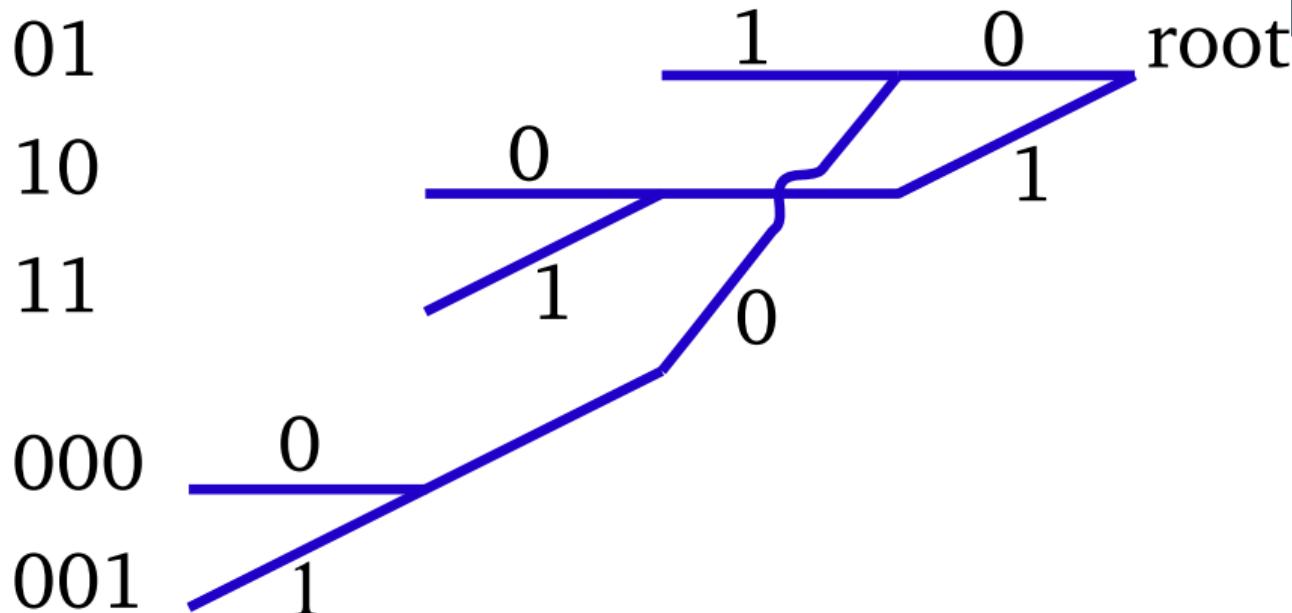
Huffman coding example 1



THE UNIVERSITY
of ADELAIDE

X	Probability
a	0.25 → 0.25 → 0.25 → 0.55 → 1.0
b	0.25 → 0.25 → 0.45 → 0.45
c	0.2 → 0.2
d	0.15 → 0.3 → 0.3
e	0.15

Huffman coding example 1



- ▶ Read the codes from the root to the end point.
- ▶ Assign 0 to the branch with higher probability at each node.
 - ▶ this choice is arbitrary, but will mean we get consistent results

Huffman coding example 1

X	Probability	Codeword
a	0.25	01
b	0.25	10
c	0.2	11
d	0.15	000
e	0.15	001

Theorem

Huffman coding is optimal (in the sense that the expected length of its codewords is at least as good as any other code).

- ▶ Huffman coding is not so obviously greedy
 - ▶ we group the two smallest probabilities
 - ▶ roughly it is trying to grab as much *entropy* as it can each step
- ▶ There's a lot more to this topic
 - ▶ information theory
 - ▶ unique decodability
- ▶ But it's another example of a *good* greedy algorithm
 - ▶ and it's a real example (Huffman like codes are really used in many, many places)

- ▶ Heuristics are used to construct algorithms to attack difficult problems
 - ▶ not guaranteed to find optimal solution
 - ▶ but can often find good solutions to hard problems
- ▶ Greedy heuristic is one of the most common
 - ▶ very simple and easy to implement
 - ▶ works well for some problems
 - ▶ when optimal solutions are sparse
 - ▶ when optimal solutions are built up from optimal solutions to subproblems
 - ▶ works badly for others, but still might be used to construct an initial solution that we can build on



OPTIMISATION & OPERATIONS RESEARCH

4 Integer Programming Problems

4.10 Graphs & networks

Ashley Dennis-Henderson

School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

- ▶ A directed graph is a *tree* if it is connected and acyclic.
- ▶ A directed graph (N', L') is a *subgraph* of (N, L) if $N' \subseteq N$ and $L' \subset L$.
- ▶ A subgraph (N', L') is a *spanning tree* if it is a tree and $N' = N$.

Definition (A Path)

A **path** in a directed network $G(N, L)$ (of node set N and link set L) is a list of nodes, $i_1, i_2, i_3 \dots i_{r-1}, i_r$, where

- (i) $i_j \in N$ for all $j = 1 \dots r$
- (ii) for each successive pair of nodes, $(i_k, i_{k+1}) \in L$, and
- (iii) with no repetition of nodes i.e $i_k \neq i_j$ when $k \neq j$.

Definition (A cycle)

A **cycle** in a directed network $G(N, L)$ is a list of nodes, $i_1, i_2, i_3 \dots i_{r-1}, i_r, i_1$, where $i_1, i_2, i_3 \dots i_{r-1}, i_r$ is a path and $(i_r, i_1) \in L$ (i.e., the link from the last node of a path to the first is included).

Graph Terminology

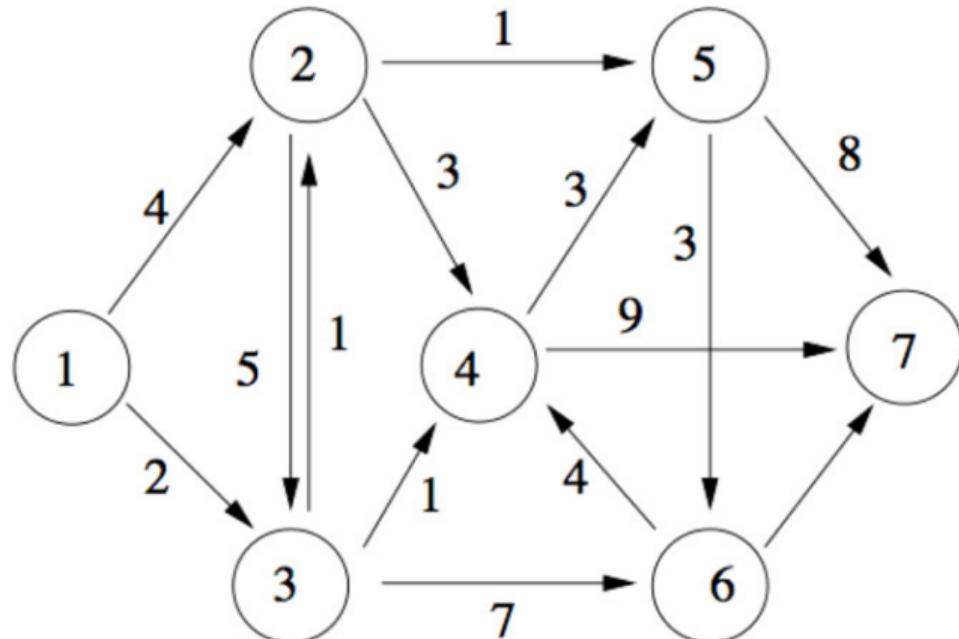
Example

Consider a path from node 1 to node 7

1, 2, 4, 7,
is one path,
while another is
1, 3, 2, 4, 5, 6, 7.

Note that 1, 2, 4, 6, 7
is not a path,
because $(4, 6) \notin L$.

A (directed) cycle is
4, 5, 6, 4.



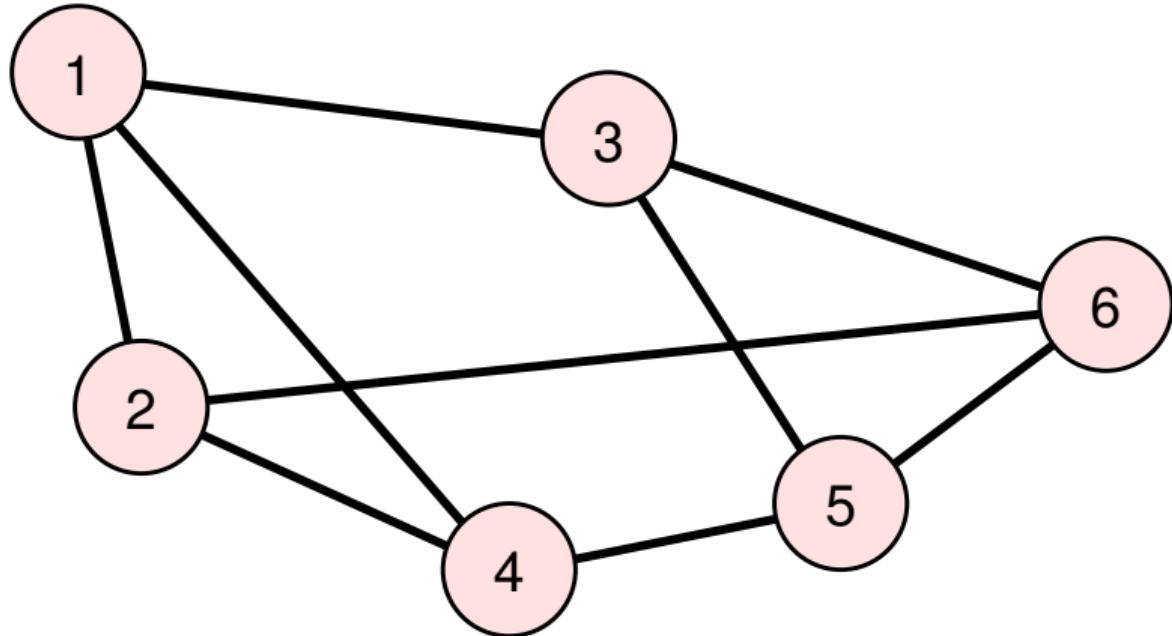


- ▶ Origin-Destination (O-D) pair $(p, q) \in N \times N$
- ▶ Let K be the set of all O-D pairs, with $K = \{[p, q] : p, q \in N\}$.
- ▶ The set of **paths** joining an O-D pair (p, q) is denoted P_{pq} .
- ▶ The set of all paths in $G(N, L)$ is denoted P .

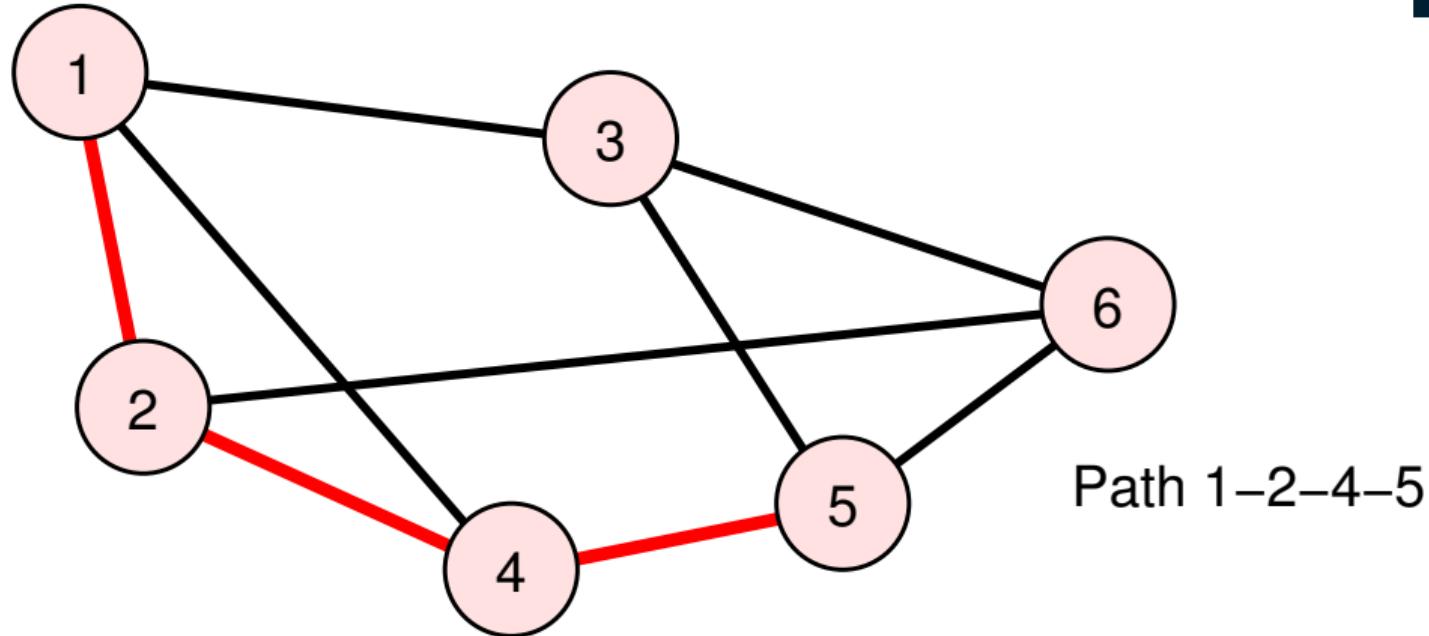
$$P = \bigcup_{[p, q] \in K} P_{pq}$$

- ▶ There can be exponentially many possible paths in a network

Network Paths

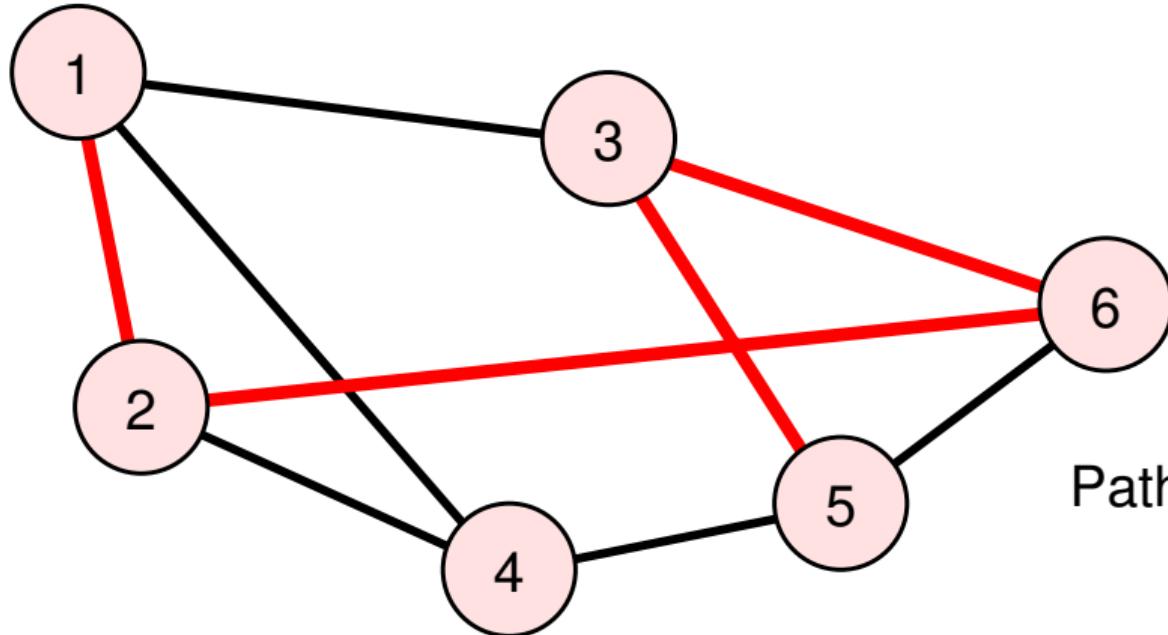


Network Paths



Paths P_{15} : 1-2-4-5

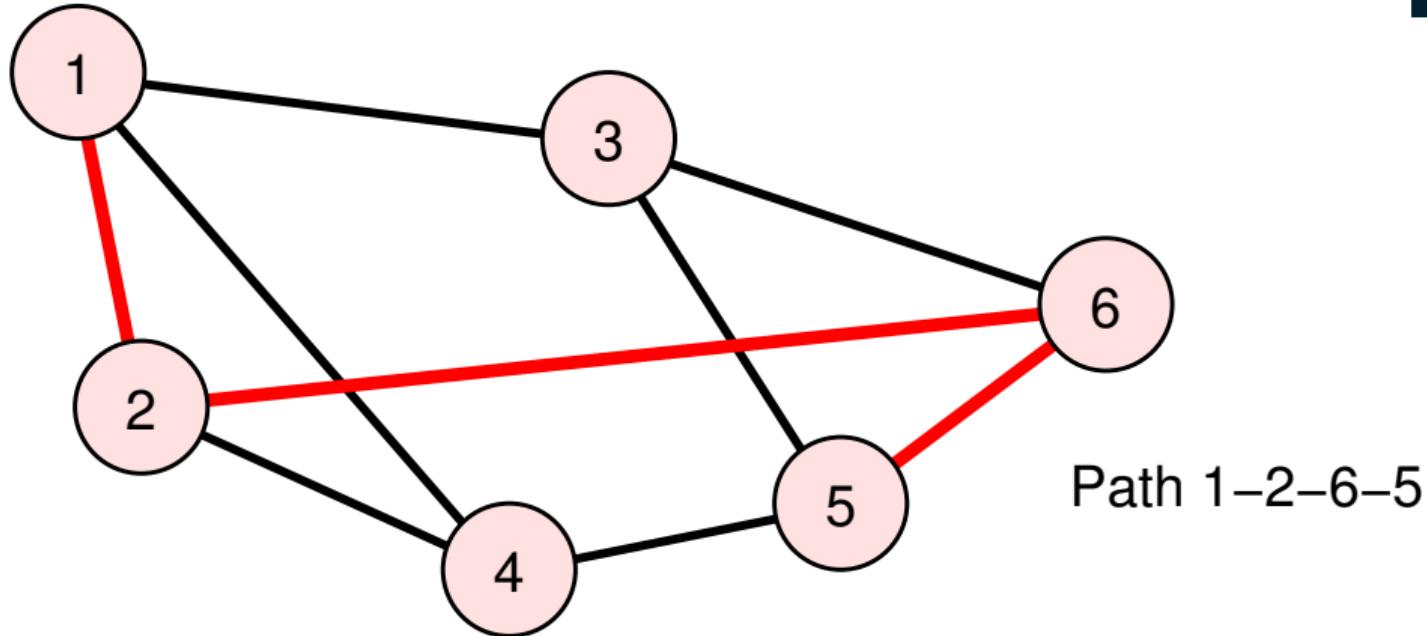
Network Paths



Path 1-2-6-3-5

Paths P_{15} : 1-2-4-5, 1-2-6-3-5

Network Paths

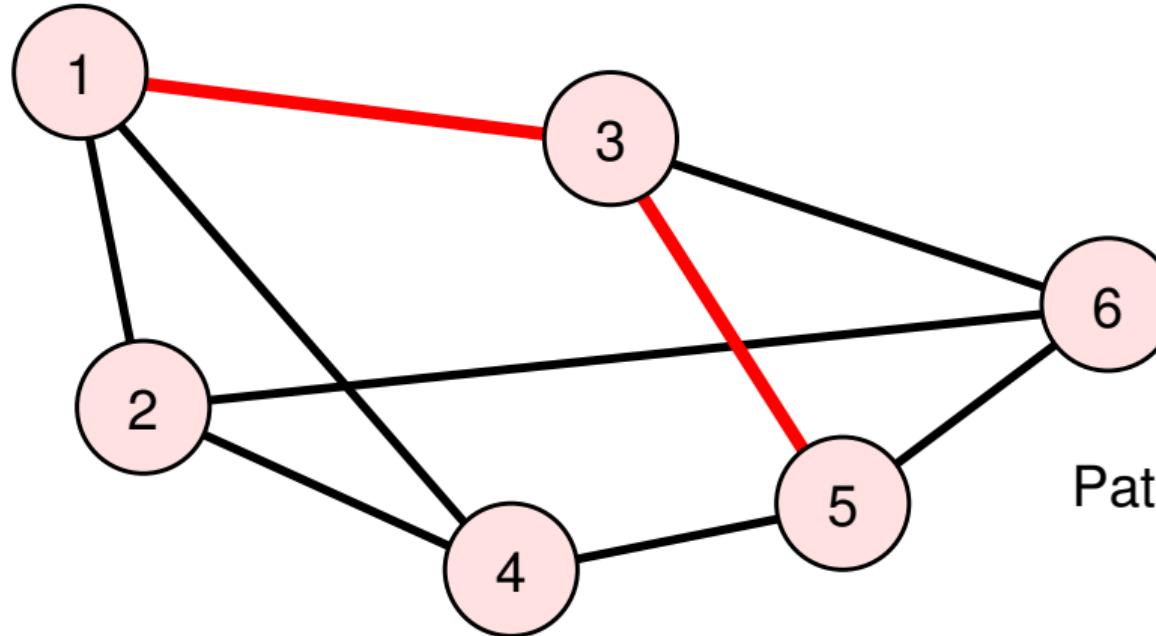


Paths P_{15} : 1-2-4-5, 1-2-6-3-5, 1-2-6-5

Network Paths



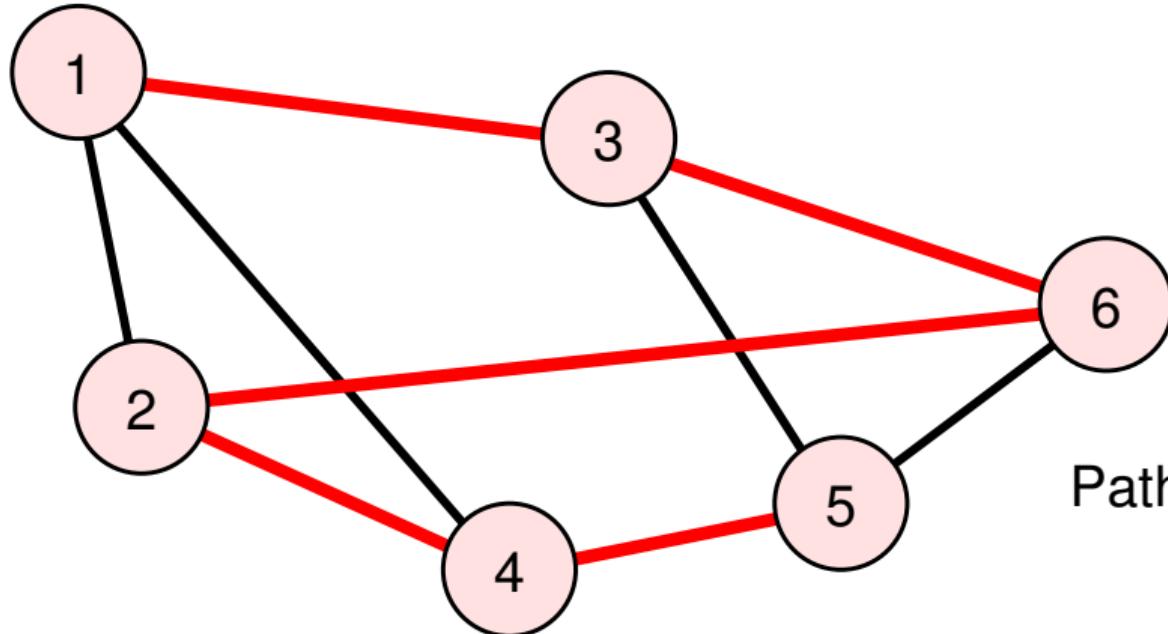
THE UNIVERSITY
of ADELAIDE



Path 1–3–5

Paths P_{15} : 1-2-4-5, 1-2-6-3-5, 1-2-6-5, 1-3-5

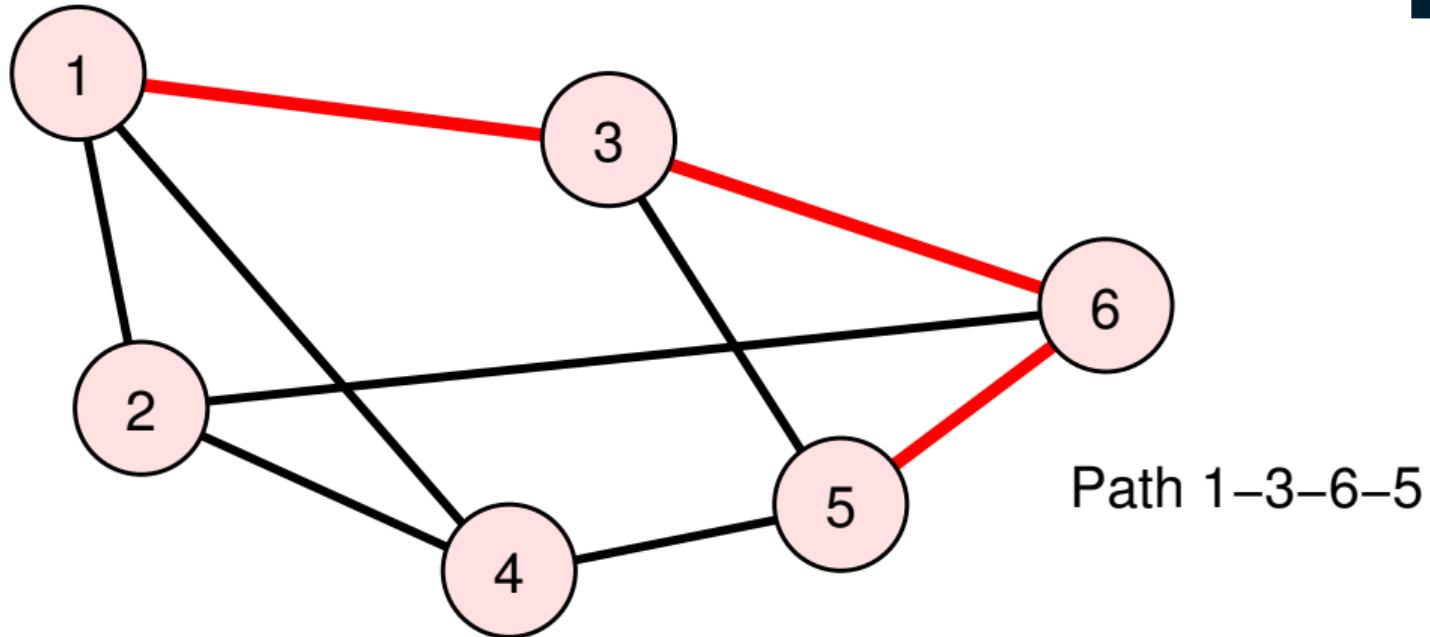
Network Paths



Path 1-3-6-2-4-5

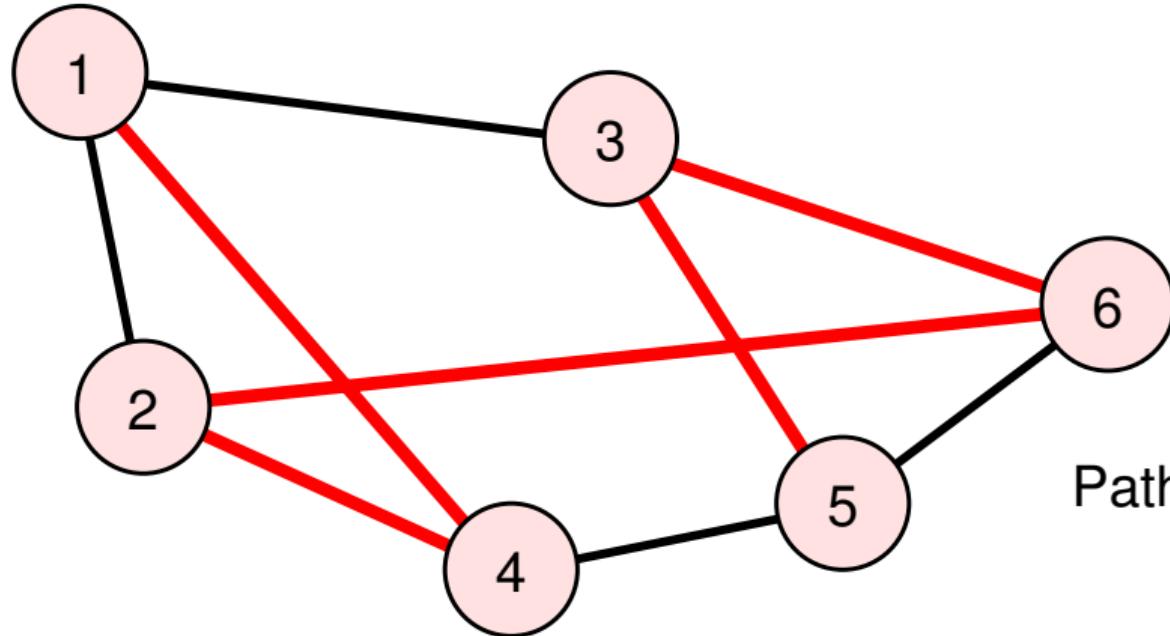
Paths P_{15} : 1-2-4-5, 1-2-6-3-5, 1-2-6-5, 1-3-5, 1-3-6-2-4-5

Network Paths



Paths P_{15} : 1-2-4-5, 1-2-6-3-5, 1-2-6-5, 1-3-5, 1-3-6-2-4-5, 1-3-6-5

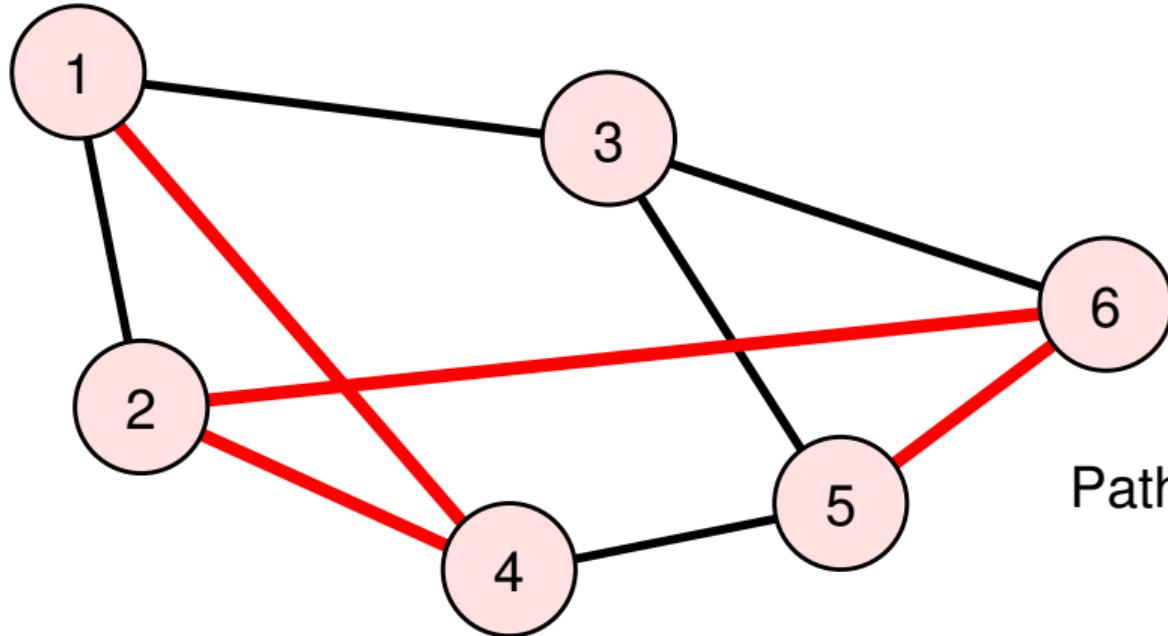
Network Paths



Path 1-4-2-6-3-5

Paths P_{15} : 1-2-4-5, 1-2-6-3-5, 1-2-6-5, 1-3-5, 1-3-6-2-4-5, 1-3-6-5, 1-4-2-6-3-5

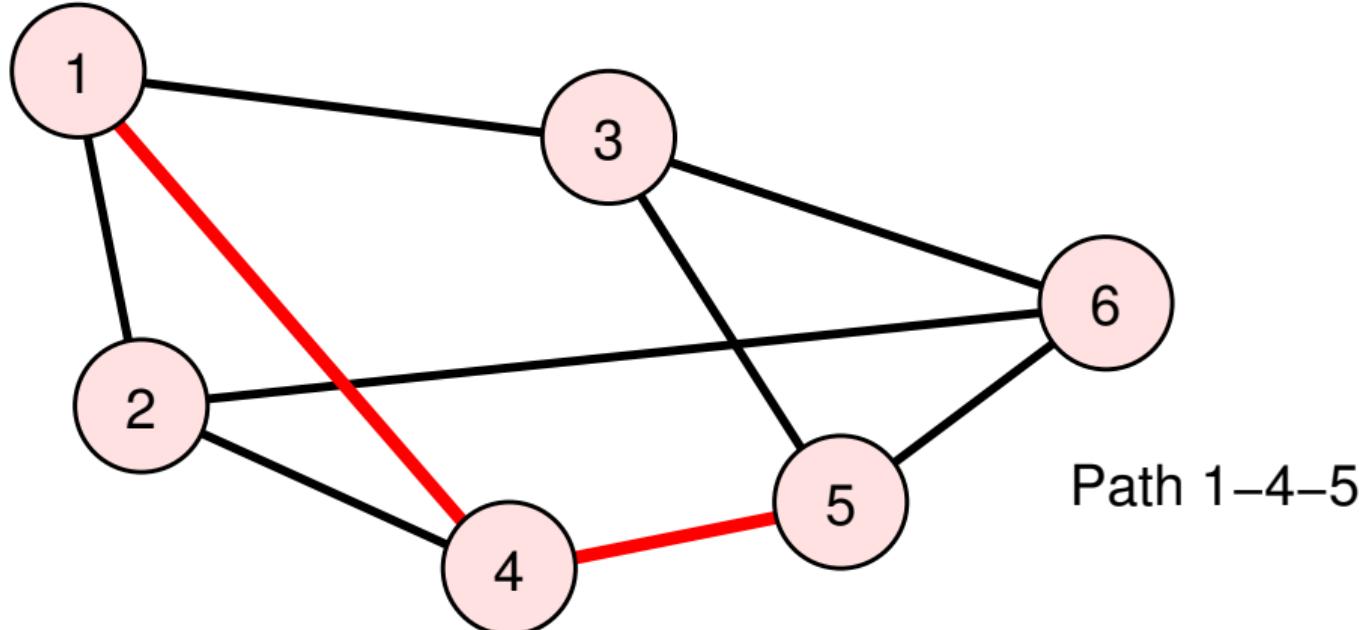
Network Paths



Path 1-4-2-6-5

Paths P_{15} : 1-2-4-5, 1-2-6-3-5, 1-2-6-5, 1-3-5, 1-3-6-2-4-5, 1-3-6-5, 1-4-2-6-3-5,
1-4-2-6-5

Network Paths



Paths P_{15} : 1-2-4-5, 1-2-6-3-5, 1-2-6-5, 1-3-5, 1-3-6-2-4-5, 1-3-6-5, 1-4-2-6-3-5, 1-4-2-6-5, **1-4-5**

4 Integer Programming Problems

4.11 Shortest Paths

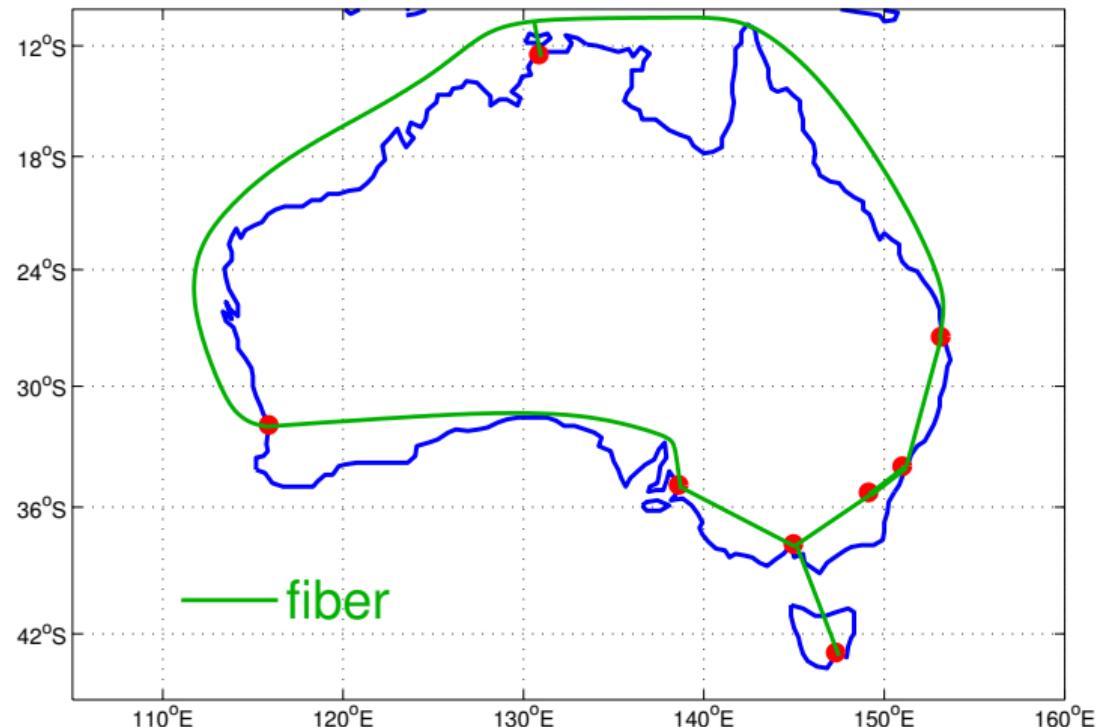
Ashley Dennis-Henderson
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

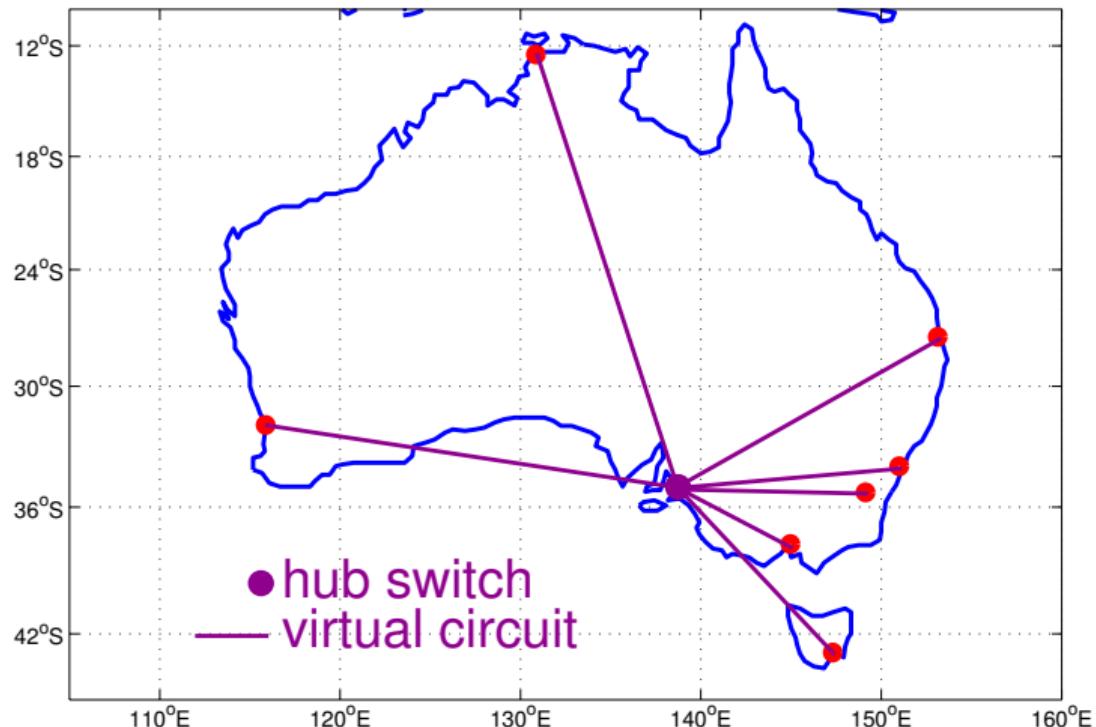
Logical vs. Physical Network

Imagine a physical network



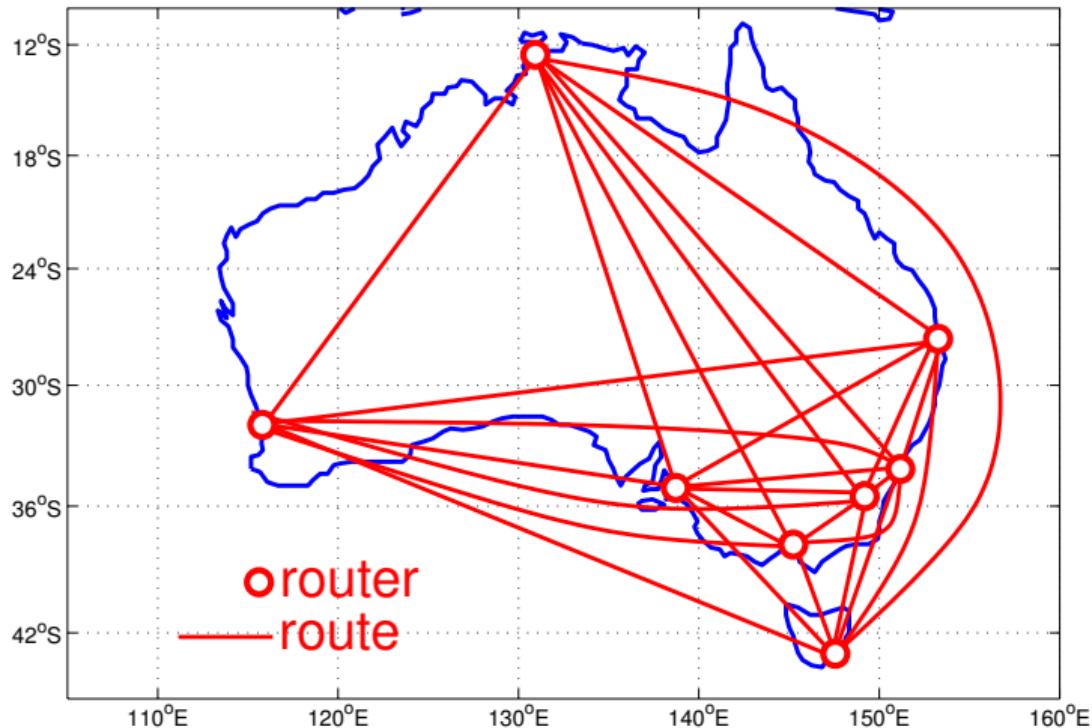
Logical vs. Physical Network

But a logical network that looks like



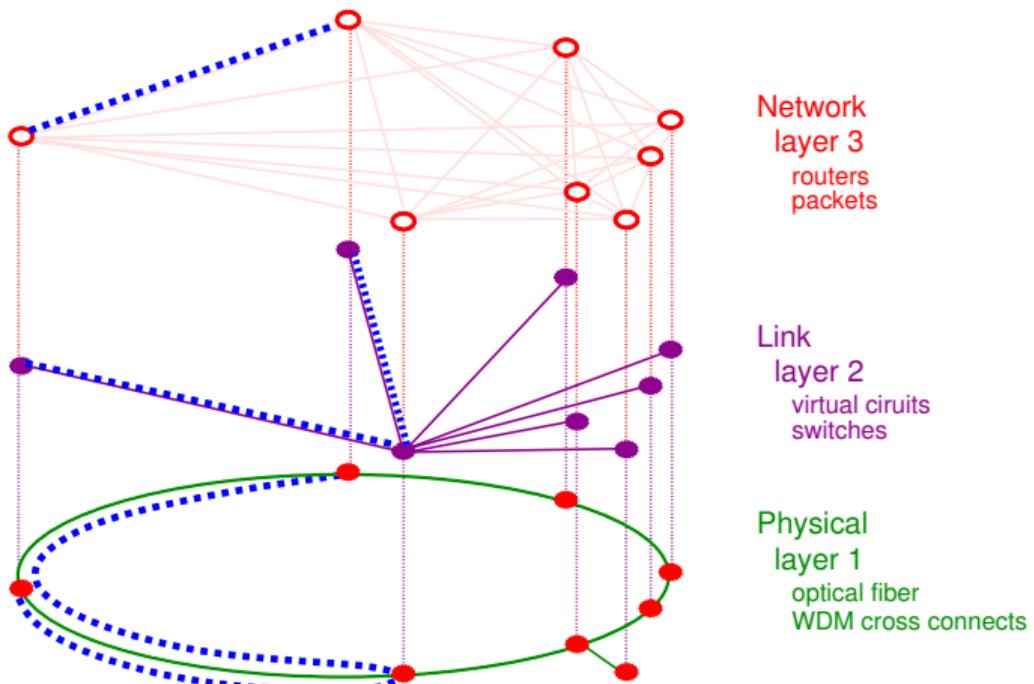
Logical vs. Physical Network

And potential pairs of network locations that want to communicate



Mapping the logical to the physical

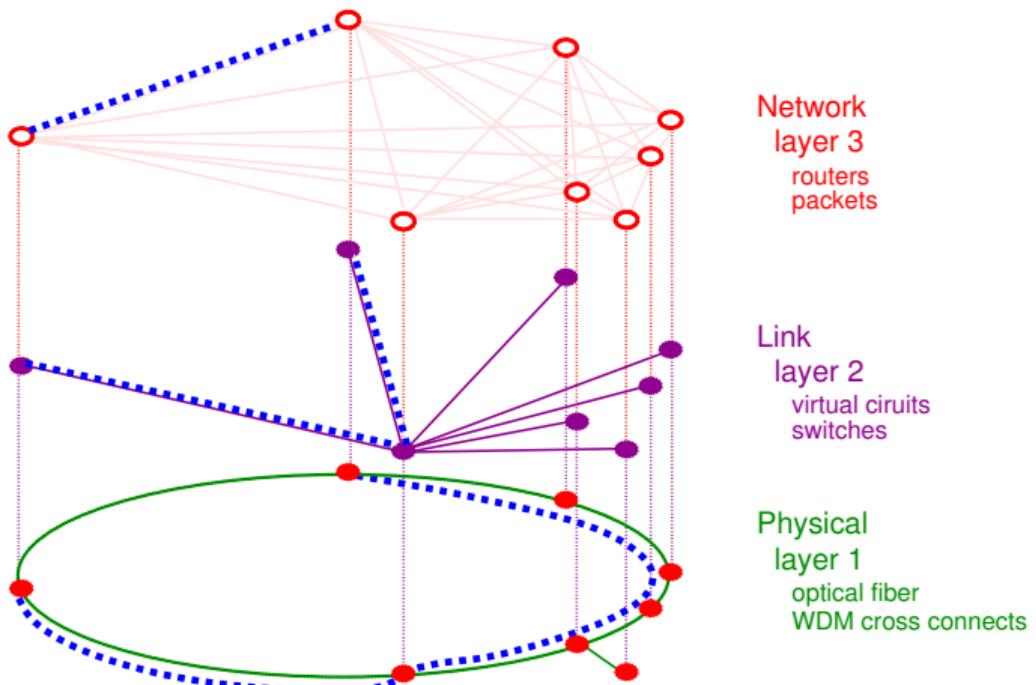
We need to map from one layer to another



That is, the logical links must be *routed* across physical links.

Mapping the logical to the physical

We need to map from one layer to another



That is, the logical links must be *routed* across physical links.

We need a method to map packet *routes* to *links*

- ▶ called a routing protocol
- ▶ several types exist
- ▶ we consider here shortest path protocols

- ▶ We have a problem of working out what path packets should take from origin to destination
- ▶ Often, links in networks have lengths associated with them
 - ▶ shortest paths would get packets to their destinations fastest
 - ▶ shortest paths also use the least resources
 - ▶ shortest paths come up in lots of other applications
- ▶ We'll look at an algorithm (Dijkstra's) for computing shortest paths
 - ▶ it's a greedy algorithm
 - ▶ but it guarantees to find the optimal path

Shortest path problem assumptions

For a network $G(N, L)$,

1. The length of link (i, j) is $c_{ij} \geq 0$
 - ▶ some shortest path algorithms can work with negative distances, but we then need to assume there are no negative cycles
2. There exists a path from s to all other $i \in N$.
 - ▶ if not, add a dummy link from s to i with very large cost M
3. What problem?
 - ▶ APSP = All Pairs Shortest Paths
 - ▶ SSSP = Single Source Shortest Paths ⇐ we'll start with this

A circular graphic on the left side of the slide features a complex network of nodes (small circles) connected by lines (edges). The colors of the nodes range from white to red, blue, and yellow. The background is a dark blue gradient with blurred red and blue circular bokeh effects.

OPTIMISATION & OPERATIONS RESEARCH

4 Integer Programming Problems

4.12 Dijkstra's algorithm

Ashley Dennis-Henderson

School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
of ADELAIDE

- ▶ in essence, routing maps
 - ▶ end-to-end traffic from p to q , i.e., t_{pq}
 - ▶ to end-to-end paths in P_{pq}
 - ▶ to links in E
- ▶ there are very many paths
 - ▶ can't search them all
 - ▶ have to be clever about choice of paths
- ▶ can use multiple paths
 - ▶ load-balancing — spreads load over paths

- ▶ fast method to find shortest paths is *Dijkstra's algorithm* [Dij59]
 - ▶ Edsger Dijkstra (1930-2002)
 - ▶ Dutch computer scientist
 - ▶ Turing prize winner 1972.
 - ▶ "Goto Statement Considered Harmful" paper
 - ▶ find distance of all nodes from one start point
 - ▶ works by finding paths in order of shortest first
 - ▶ longer paths are built up of shorter paths

Input

- ▶ graph (N, E)
- ▶ link *weights* α_e , define link distances

$$d_{ij} = \begin{cases} 0 & \text{if } i = j \\ \alpha_e & \text{where } (i, j) = e \in E \\ \infty & \text{where } (i, j) = e \notin E \end{cases}$$

- ▶ a start node, WLOG assume it is node 1

Output

- ▶ distances D_j of each node $j \in N$ from start node 1.
- ▶ a predecessor node for each node (gives path)



Dijkstra's algorithm

Let S be the set of *labelled* nodes.

Initialise: $S = \{1\}$,

$$D_1 = 0,$$

$$D_j = d_{1j}, \quad \forall j \notin S, \text{ i.e., } j \neq 1.$$

Step 1: *Find the next closest node*

Find $i \notin S$ such that $D_i = \min\{D_j : j \notin S\}$

Set $S = S \cup \{i\}$.

If $S = N$, **stop**

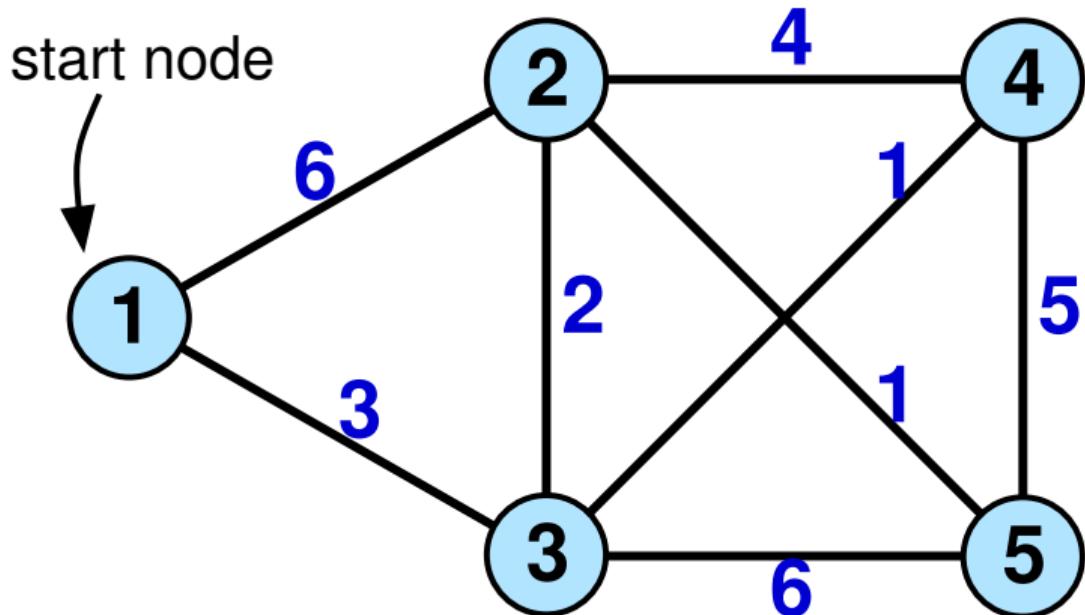
Step 2: *Find new distances*

For all $j \notin S$, set

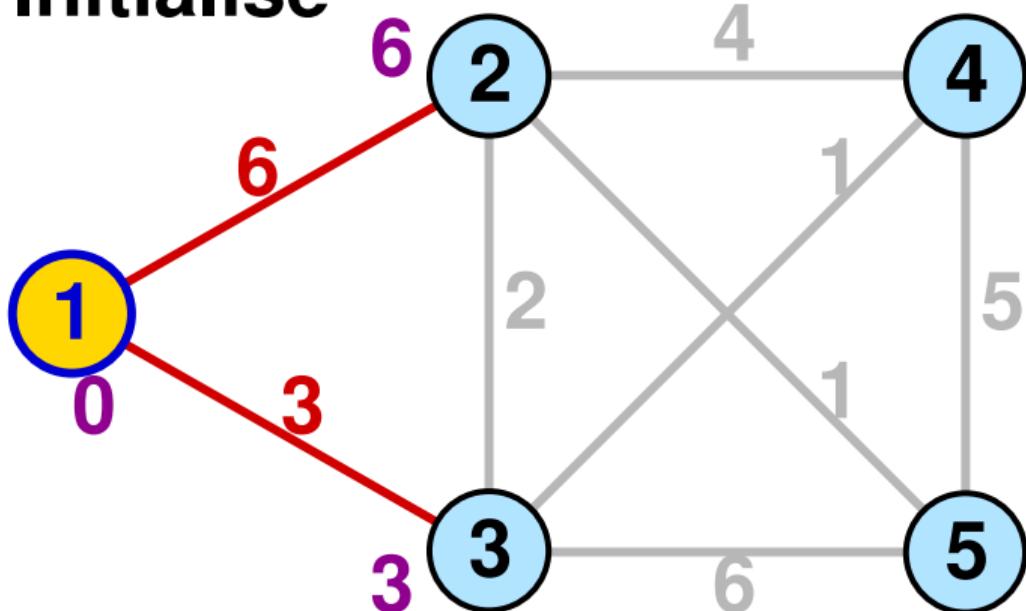
$$D_j = \min\{D_j, D_i + d_{ij}\}$$

Goto Step 1.

Dijkstra Example



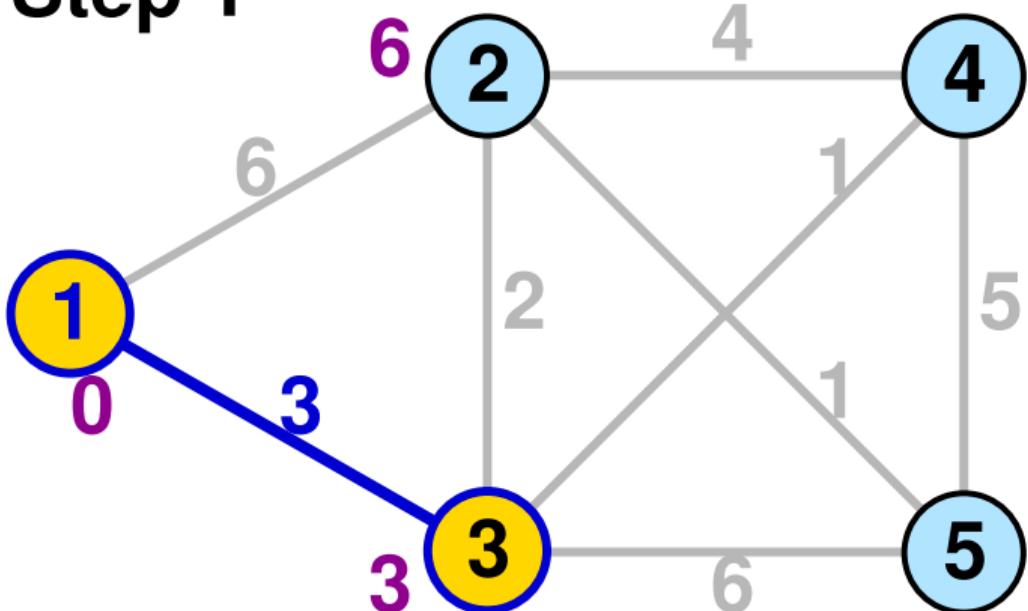
Initialise



$$S=\{1\}$$

$$D=(0, 6, 3, \ , \)$$

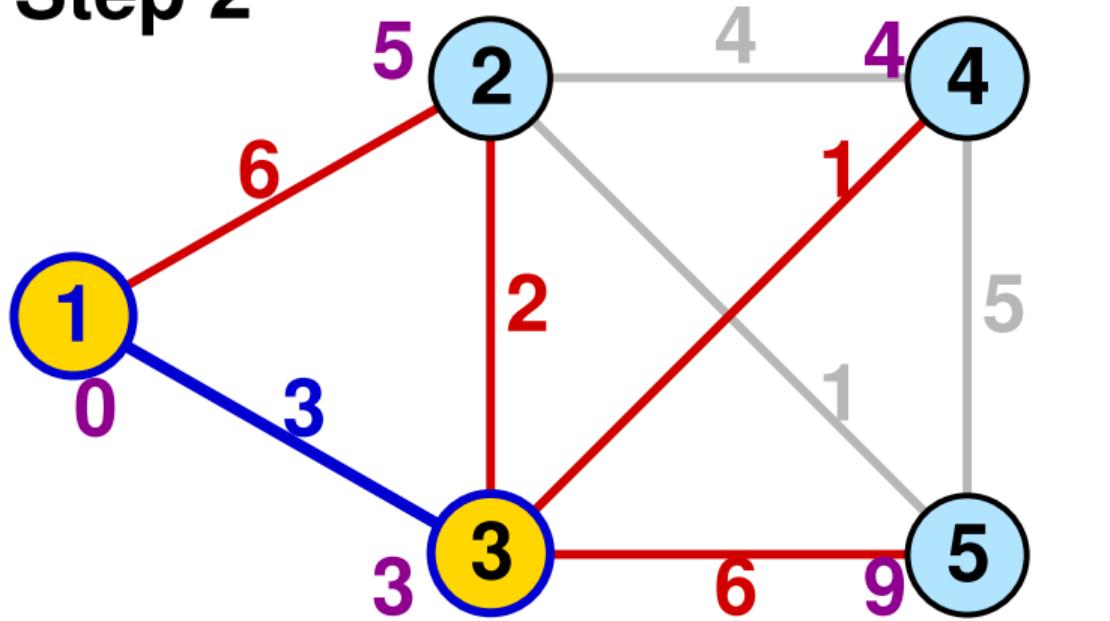
Step 1



$$S = \{1, 3\}$$

$$D = (0, 6, 3, ,)$$

Step 2

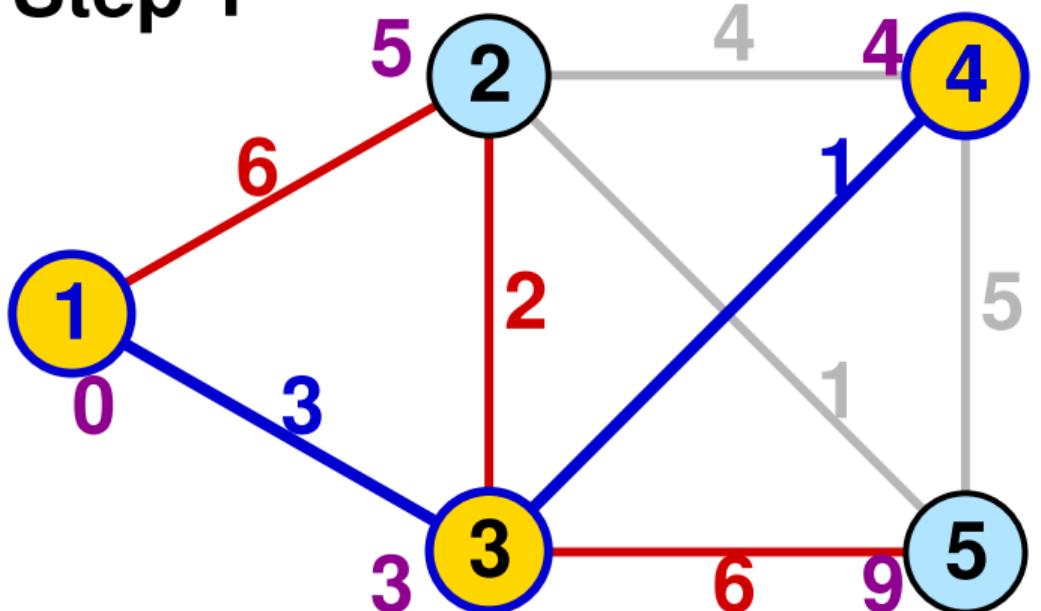


$$S = \{1, 3\}$$

$$D = (0, 5, 3, 4, 9)$$

changed

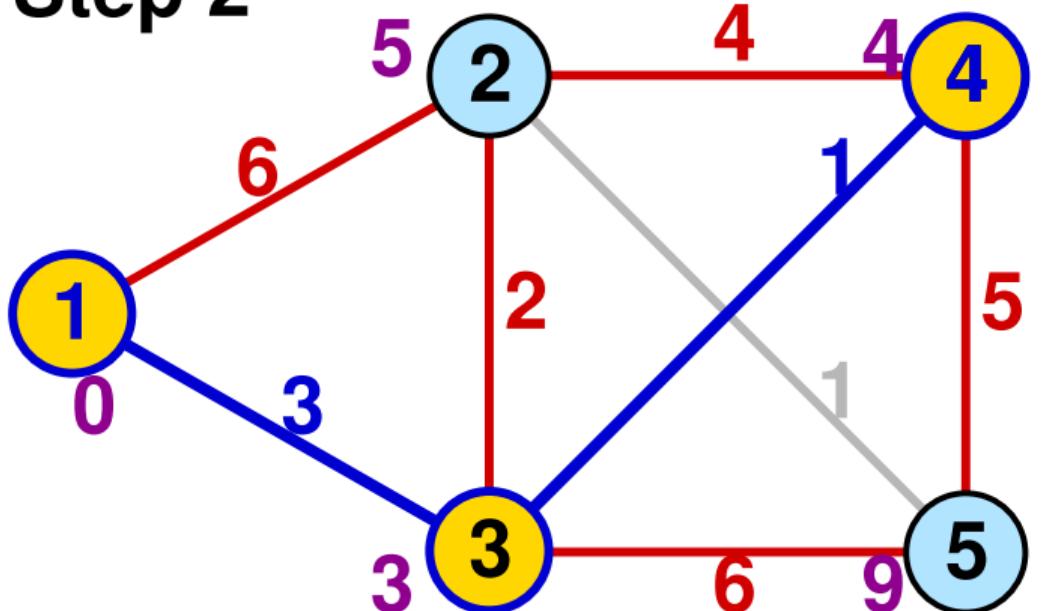
Step 1



$$S = \{1, 3, 4\}$$

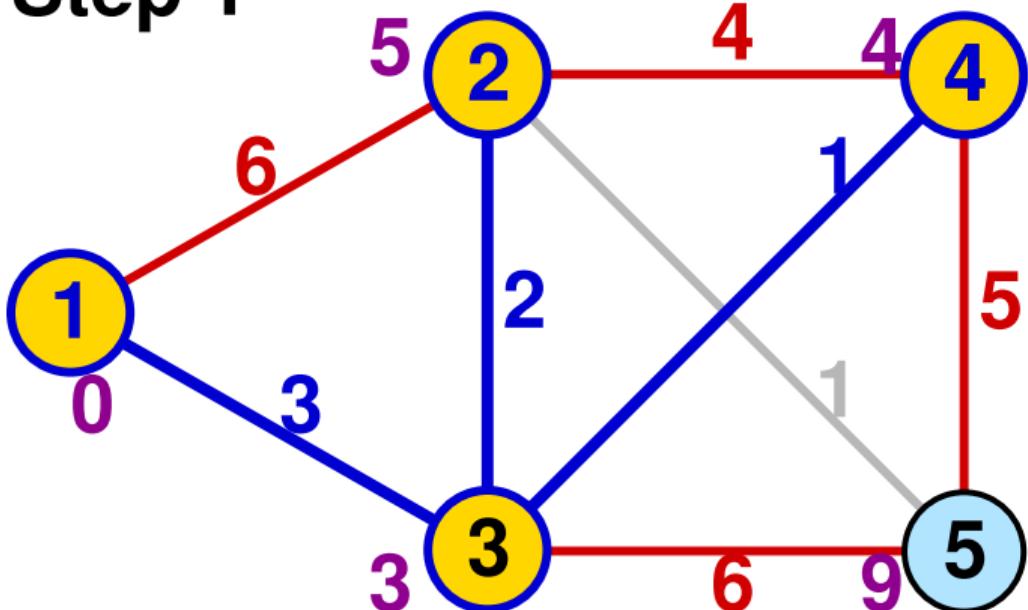
$$D = (0, 5, 3, 4, 9)$$

Step 2



$$S = \{1, 3, 4\}$$

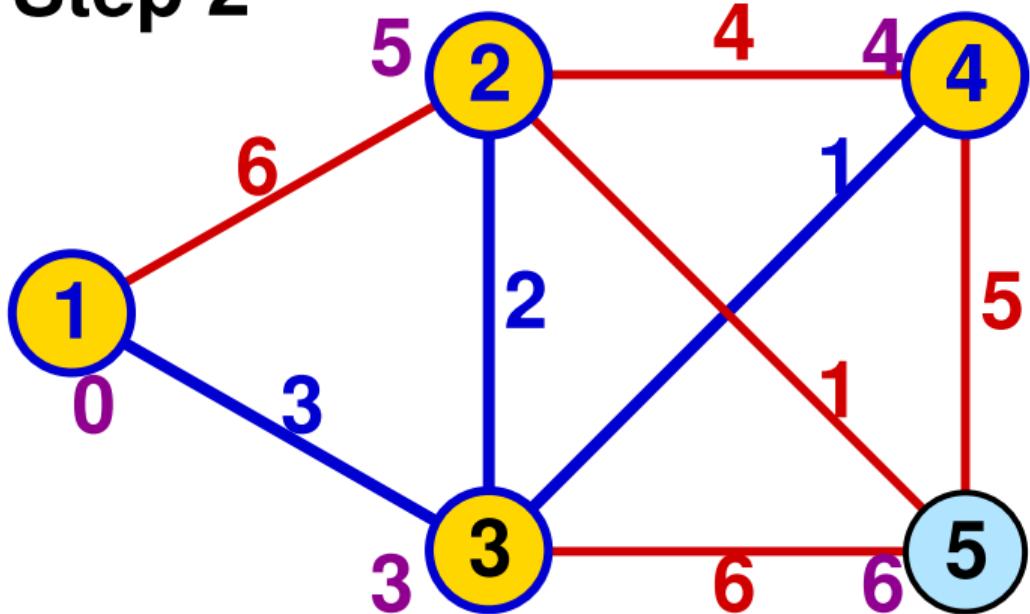
$$D = (0, 5, 3, 4, 9)$$

Step 1

$$S = \{1, 3, 4, 2\}$$

$$D = (0, 5, 3, 4, 9)$$

Step 2

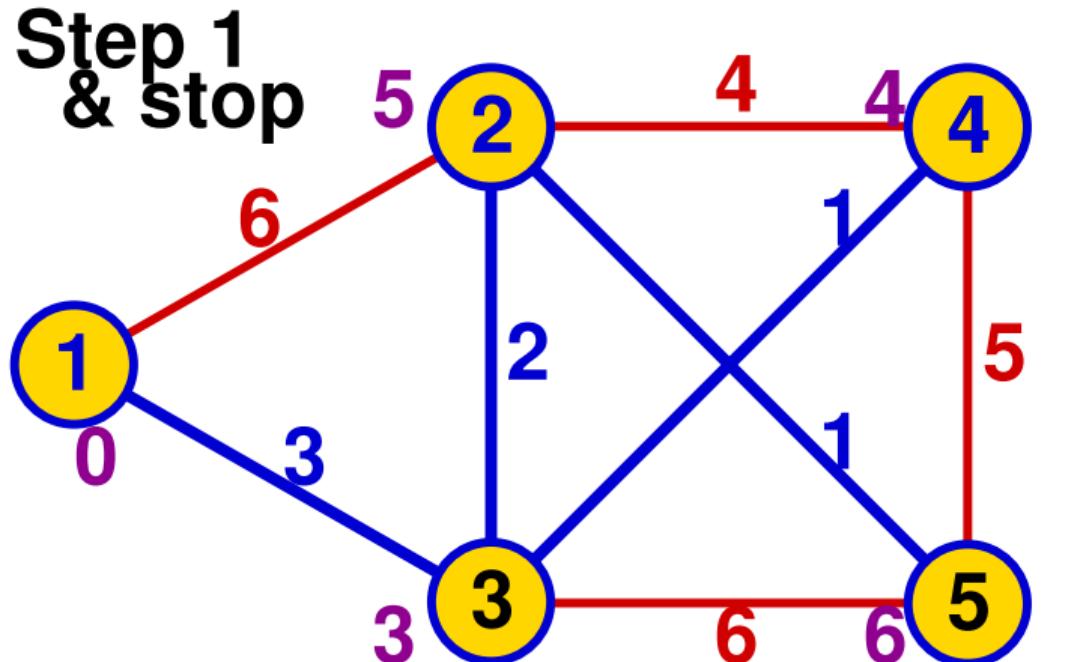


$$S = \{1, 3, 4, 2\}$$

$$D = (0, 5, 3, 4, 6)$$

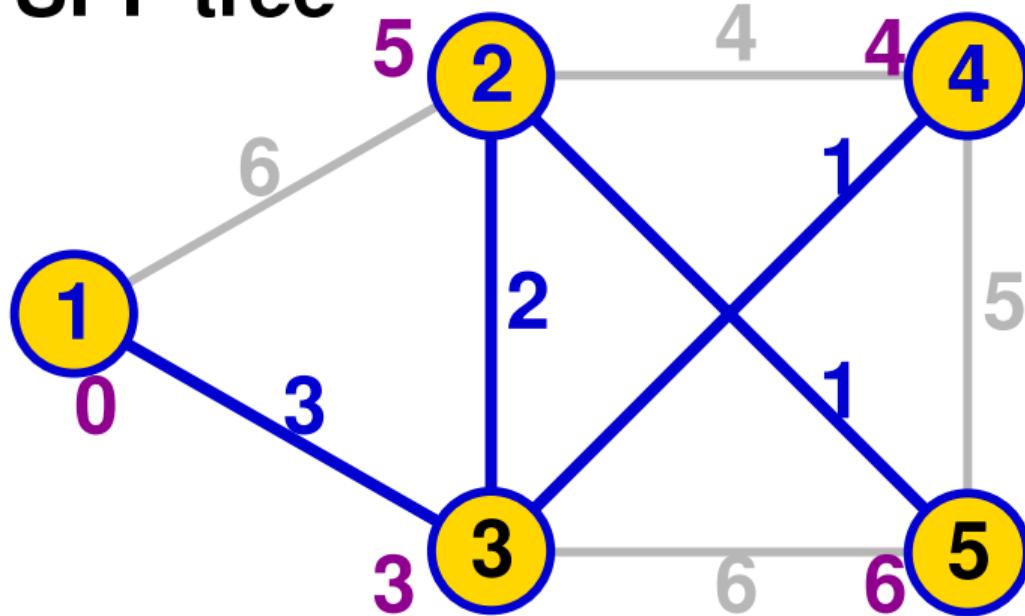
changed

Dijkstra Example



$$S = \{1, 3, 4, 2, 5\} \quad D = (0, 5, 3, 4, 6)$$

SPF tree



$$S=\{1,3,4,2,5\} \quad D=(0,5,3,4,6)$$

- ▶ build a (Shortest-Path First) *SPF tree*
- ▶ let it grow
- ▶ grow by adding shortest paths onto it
- ▶ solution must look like a tree
 - ▶ to get paths, we only need to keep track of *predecessors*, e.g., previous example

node	predecessor
1	-
2	3
3	1
4	3
6	2

- ▶ Dijkstra's algorithm solves *single-source all-destinations problem*
- ▶ easily extended to a directed graph
 - ▶ can only join up in the direction of a link
- ▶ link-distances (weights) must be non-negative
 - ▶ there are other algorithms to deal with negative weights

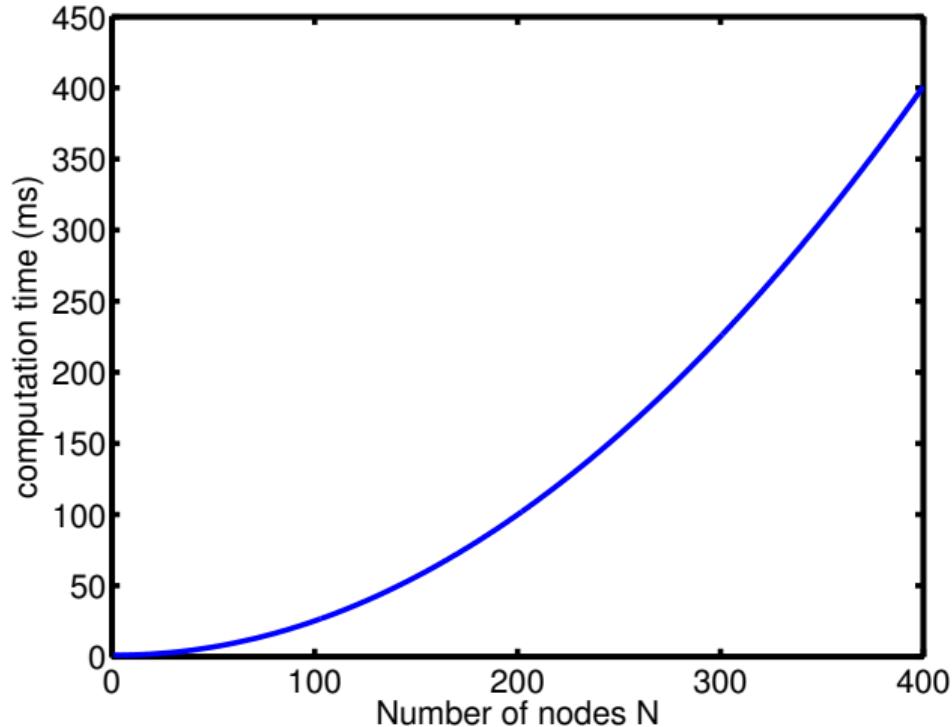
- ▶ Instance size given by number of nodes $|N|$ and edges $|E|$ in the graph
- ▶ Simple implementation complexity $O(|N|^2)$
- ▶ Cisco's implementation of Dijkstra tested in [SG01]

$$\text{comp.time} = 2.53N^2 - 12.5N + 1200 \text{ microseconds}$$

- ▶ Complexity (assuming smart data structures, i.e., Fibonacci heap) is $O(|E| + |N| \log |N|)$,
 - ▶ $|E|$ = number of edges
 - ▶ $|N|$ = number of nodes
- ▶ To compute paths for all pairs, we can perform Dijkstra for each starting point, with complexity $O(|N||E| + |N|^2 \log |N|)$,

Dijkstra complexity

Empirical Cisco 7500 and 12000 (GSR) computation times for Dijkstra [SG01]



$$2.53N^2 - 12.5N + 1200\mu s$$

Theorem

Dijkstra's algorithm solves the single-source shortest-paths problem in networks that have nonnegative weights.

Proof: Call the source node s the root, then we need to show that the paths from s to each node x corresponds to a shortest path in the graph from s to x . Note that this set of paths forms a tree out of a subset of edges of the graph.

The proof uses induction. We assume that the subtree formed at some point along the algorithm has the property (of shortest paths). Clearly the starting point satisfies this assumption, so we need only prove that adding a new node x adds a shortest path to that node. All other paths to x must begin with a path from the current subtree (because these are shortest paths) followed by an edge to a node not on the tree. By construction, all such paths are longer than the one from s to x that is produced by Dijkstra.



- ▶ Shortest paths
 - ▶ common optimisation problem
 - ▶ Dijkstra is a good solution
 - ▶ but not the only one: there are better approaches
 - ▶ some deal with more general cases
 - ▶ some are distributed
 - ▶ some are slightly faster
- ▶ Greed is good
 - ▶ greedy algorithms can be optimal
 - ▶ there are lots of similar algorithms
 - ▶ e.g., Prim's algorithm for finding minimum spanning trees

5 More sophisticated algorithms

5.1 Randomised Algorithms

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



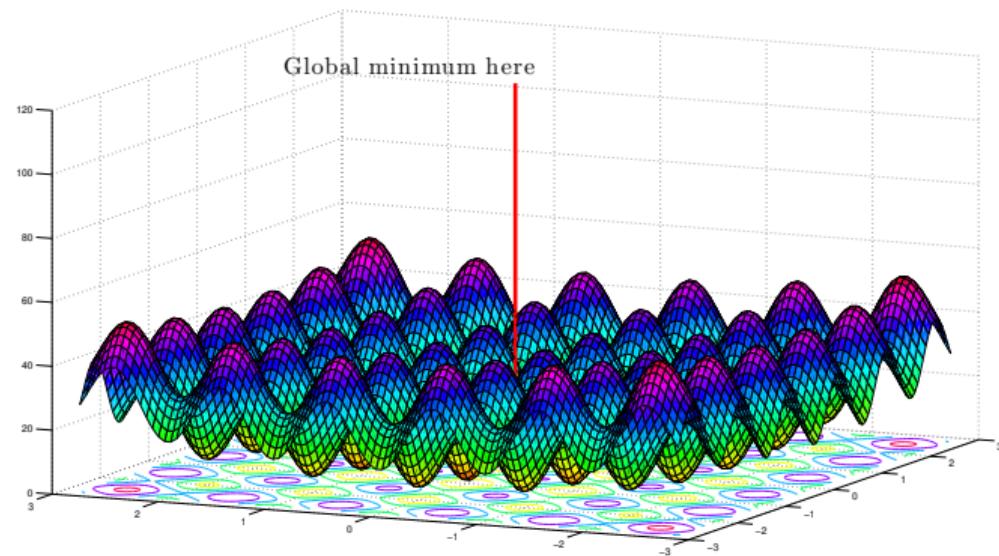
THE UNIVERSITY
*of*ADELAIDE

- ▶ Integer Linear Programs
 - ▶ *NP-hard*: no known deterministic polynomial time algorithm
 - ▶ *non-convex*: local minimum isn't necessarily a global minimum
- ▶ Leads to heuristic approaches, e.g., Greedy
 - ▶ think of as searching “downhill”
 - ▶ it can easily be caught in a local minimum

A graphical motivation

Consider $f(x, y) = 20 + x^2 - 10 \cos(2\pi x) + y^2 - 10 \cos(2\pi y)$ (Rastrigin's function)

Global minimum at $x = y = 0$, many local minima.





- ▶ Introduce a stochastic element
 - ▶ don't always take the "best" next step
 - ▶ sometimes even allow a "bad" step
 - ▶ allows you to backtrack out of local minima
- ▶ The algorithm can't be completely random
 - ▶ it's still a guided search
- ▶ There are many approaches, often motivated by nature
 - ▶ Ants
 - ▶ Simulated Annealing
 - ▶ *Genetic Algorithms*

5 More sophisticated algorithms

5.2 Genetic Algorithms and Evolutionary Computing

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

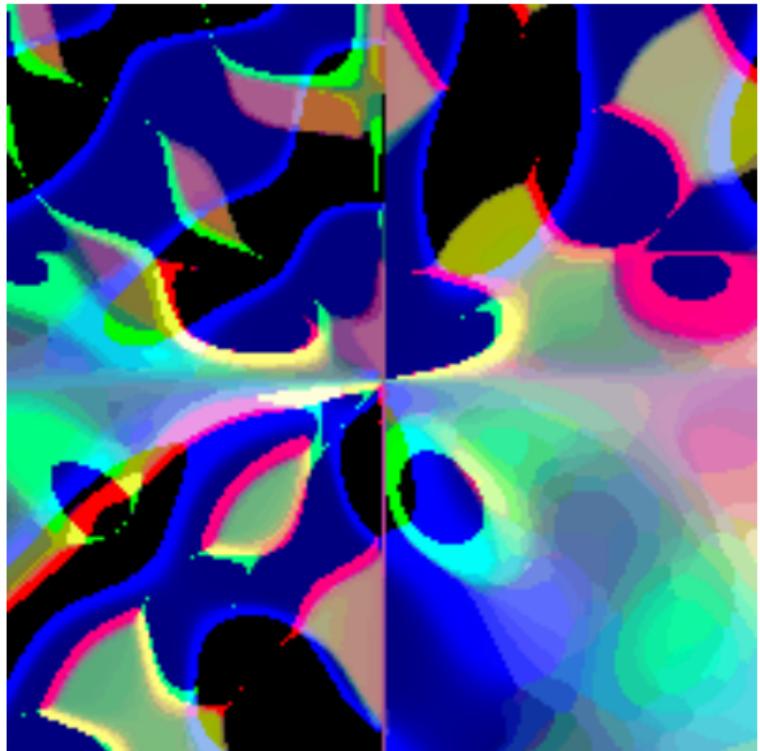
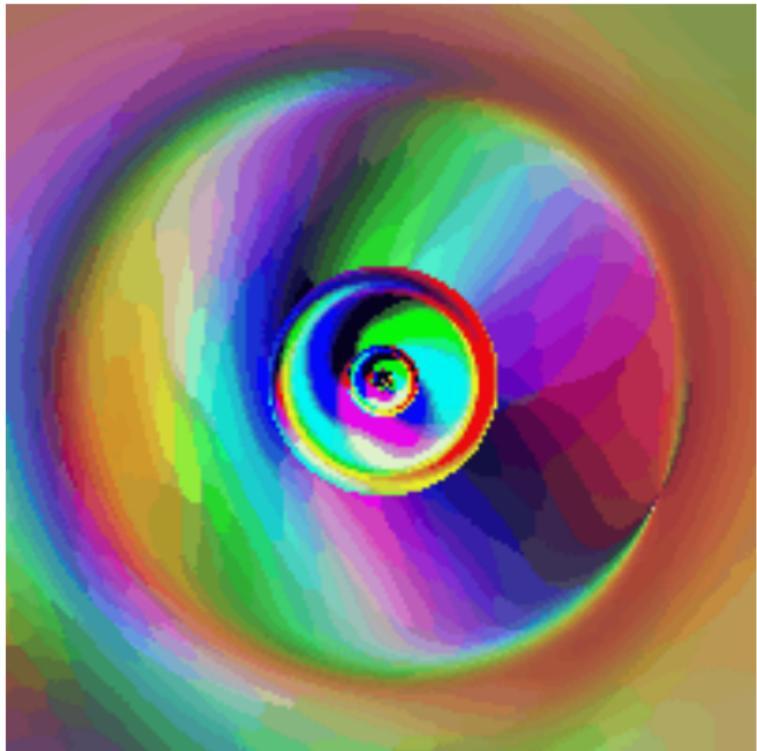
A set of randomized algorithms which derive their behavior from a metaphor of the processes of *evolution* in nature.

- ▶ inspired by Darwin's theory of evolution
 - ▶ survival of the fittest
- ▶ pioneered by John Holland in the 60s (see [Hol75]).
- ▶ lots of applications, e.g., [Gre85, Sch89]

- ▶ ease with which it can handle arbitrary kinds of constraints and objectives
 - ▶ only have to be able to compute them
 - ▶ don't even need to be able to express (as math)
- ▶ makes them highly applicable where
 - ▶ search space is complex or poorly understood
 - ▶ expert knowledge is difficult to encode to narrow the search space
 - ▶ mathematical analysis is not available
 - ▶ the objective is evaluable, but not expressable
 - ▶ e.g., optimising game-playing AI

Used in many areas, even in art

- ▶ “I don't know much about art, but I know what I like”



- ▶ living things are built using a plan described in our *chromosomes*
- ▶ chromosomes are strings of *DNA* and serve as a model for the whole organism
- ▶ a chromosome's DNA is grouped into blocks called *genes*, which have a location called a *locus*
- ▶ notionally, a gene codes for a particular trait
 - ▶ e.g., blue, or brown eyes
 - ▶ possible settings for a trait are called *alleles*
- ▶ a complete set of genetic material (all chromosomes) is called a *genotype*
- ▶ expression produces a *phenotype* (the organism) from the genotype

- ▶ during reproduction, recombination occurs
 - ▶ in this context we call it *crossover*
 - ▶ genes from parents combine to give genes for offspring
- ▶ *mutation* also happens
 - ▶ it means that the elements of DNA are a little changed (randomly)
- ▶ fitness of an organism is measured by success of the organism in its reproduction
 - ▶ fitter organisms reproduce more, and so propagate their genes further
 - ▶ ⇒ the theory of evolution is sometimes referred to as “survival of the fittest”
 - ▶ all sorts of interesting variations here

1. *initialization*: create (randomly) an initial set of N solutions called the *population*, \mathcal{P}
2. *while* not finished
 - 2.1 *evaluate fitness*: $f(x)$ of each $x \in \mathcal{P}$
 - 2.2 *generate a new population*: the offspring
 - 2.2.1 *selection*: select two parents from population according to their fitness (better fitness makes them more likely to be selected)
 - 2.2.2 *crossover*: With a crossover probability p cross over the parents to form new offspring, otherwise direct copy of the parents.
 - 2.2.3 *mutation*: With a mutation probability q mutate new offspring at each locus
 - 2.3 *replace old population*:
 - 2.4 *decide whether to finish*



- ▶ very general
 - ▶ once again it's really a *meta-heuristic*
- ▶ every bit can be implemented in different ways
- ▶ key components
 - ▶ chromosome encoding
 - ▶ crossover method
 - ▶ mutation method
 - ▶ selection method
 - ▶ fitness criteria
- ▶ don't need explicit fitness function
 - ▶ could be the result of winners of a game
 - ▶ e.g., competition between population



OPTIMISATION & OPERATIONS RESEARCH

5 More sophisticated algorithms

5.3 Genetic Algorithms: key components

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

► *Binary Encoding:*

Chromosome A 1101100100110110

- each bit represents some characteristic
 - e.g., a link is used in a particular path
- string can represent a number: using Gray codes

► *Permutation Encoding:*

Chromosome A 1 5 3 2 6 4 7 9 8

Chromosome B 8 5 6 7 2 3 1 4 9

- each chromosome is a permutation

► *Value Encoding:* encode values directly

► *Tree Encoding:* used for encoding complex meaning such as a computer program or a game strategy

- ▶ represents each number in the sequence of integers $\{0 \dots 2^{N-1}\}$ as a binary string of length N
- ▶ in an order such that adjacent integers have Gray code representations that differ in only one bit position
- ▶ marching through the integer sequence therefore requires flipping just one bit at a time
- ▶ Example $N = 3$ (of a binary-reflected Gray code)

The binary coding of $\{0 \dots 7\}$

numbers	0	1	2	3	4	5	6	7
binary coding	000	001	010	011	100	101	110	111
Gray coding	000	001	011	010	110	111	101	100



- ▶ operate on selected genes from parents' chromosomes to create offspring's genes
- ▶ simplest way is single crossover point
 - ▶ randomly choose a *crossover point*
 - ▶ copy first chromosome up to the crossover point
 - ▶ copy second chromosome after the crossover point

Single crossover point example

Parent 1: 1101100100110110

Parent 2: 1111111000011110

Offspring: 1101111000011110



crossover

Crossover (continued)

There are other ways how to make crossover

- ▶ *multiple crossover points*: more than one random crossover point is chosen
- ▶ *random crossover*: randomly select genes from each parent
- ▶ *arithmetic crossover*: some arithmetic operation is performed to make a new offspring

Different types of crossovers work better for different problems.

Crossover for permutation encoding

- ▶ crossover for permutation coding is a little different
- ▶ Single point crossover
 - ▶ one crossover point is selected
 - ▶ copy from the first parent to the crossover
 - ▶ then the other parent is scanned and if the number is not yet in the offspring, it is added

Example:

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

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

Offspring: 1 2 3 4 5 6 8 9 7



crossover

Mutation for permutation encoding

- ▶ as with crossover, lots of possibilities
 - ▶ if a group of bits encode a gene, we could mutate whole genes at each step
 - ▶ random mutation, but only allow solutions with increased fitness
 - ▶ for permutation encoding, need different approach
 - ▶ e.g., swap a randomly chosen pair

- ▶ *Roulette Wheel Selection:* select randomly based on fitness function. Probability of selection of x_i is

$$p_i = \frac{f(x_i)}{\sum_{i \in P} f(x_i)}$$

- ▶ *Rank Selection:* rank the population in order, so that $f(x_{(1)}) \leq f(x_{(2)}) \leq \dots \leq f(x_{(N)})$. The probability of selection of $x_{(i)}$ is

$$p_i = \frac{i}{\sum_{i \in P} i} = \frac{2i}{N(N+1)}$$

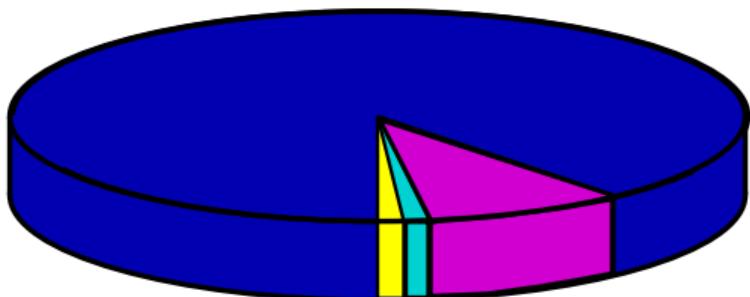
- ▶ *Elitism:* we automatically keep the best one from each generation.

Roulette Wheel Selection



THE UNIVERSITY
ofADELAIDE

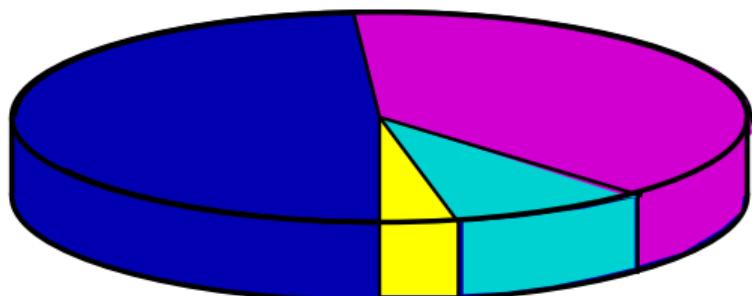
- ▶ Parents are selected according to their fitness
- ▶ The better the genotype is, the more chances it has to be selected
- ▶ Imagine a roulette wheel where all the genotypes in the population are placed.
- ▶ The size of the section in the roulette wheel for an individual is proportional to its fitness function
 - ▶ the bigger the value is, the larger the section is



Rank Selection



- ▶ Roulette Wheel Selection has problems when there are big differences in fitness values
 - ▶ one individual may completely dominate
 - ▶ other parents have little chance to be selected
- ▶ Rank selection ranks the population from $1, \dots, N$
 - ▶ selection with probability determined by ranking
 - ▶ worst will have probability $2/[N(N + 1)]$
 - ▶ best will have probability $2N/[N(N + 1)]$





- ▶ create new population by crossover and mutation
 - ▶ parents don't appear in new population
 - ▶ likely that we will lose the best parent
- ▶ *elitism*
 - ▶ keep a few of the best current generation
 - ▶ rest of new population constructed as above
- ▶ elitism can rapidly increase the performance of GA
 - ▶ prevents a loss of the current best solution
 - ▶ algorithm never goes completely backwards

5 More sophisticated algorithms

5.4 Genetic algorithms example: travelling salesperson

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



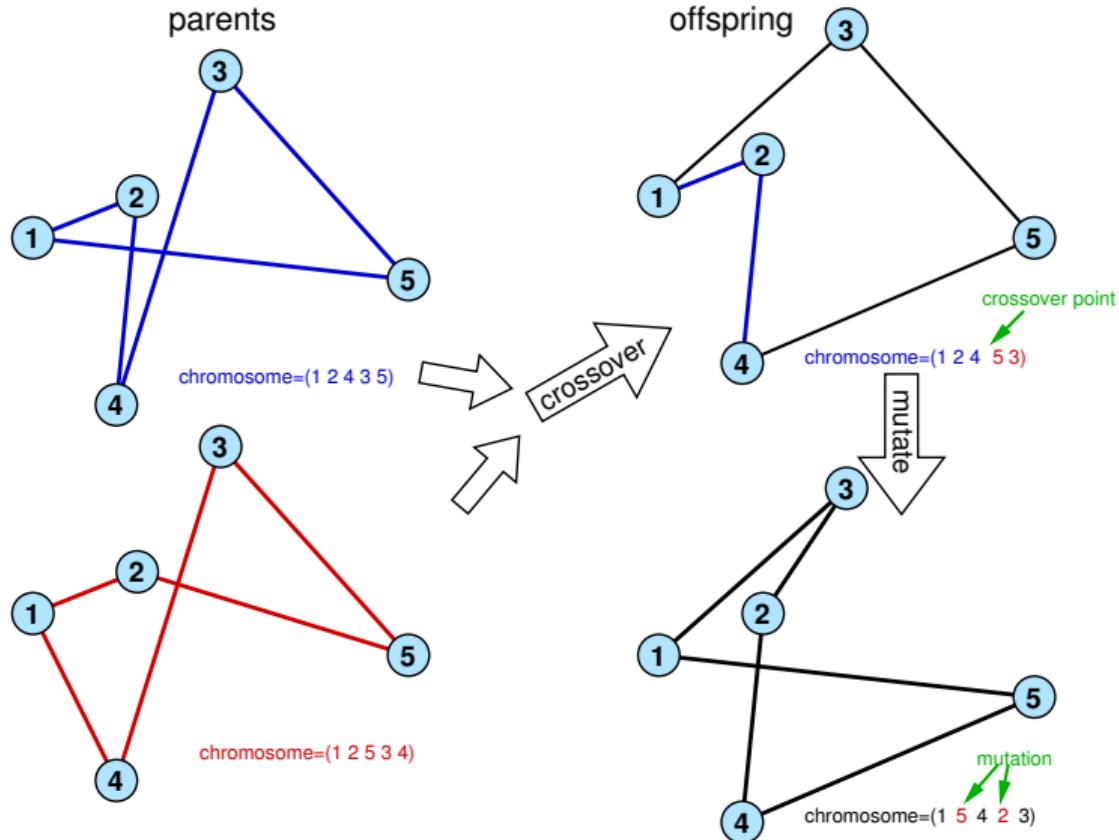
THE UNIVERSITY
*of*ADELAIDE

- ▶ We could encode by putting $z_{(i,j)}$ into the chromosome
 - ▶ $z_{(i,j)} \in \{0, 1\}$ indicates whether link (i, j) is used
 - ▶ this doesn't include the constraint that we visit each city once, in a circuit
 - ▶ we would have to include this constraint in the fitness function
 - ▶ much larger search space
- ▶ Easier encoding is the permutation encoding
 - ▶ gives the order of the cities we visit
 - ▶ automatically includes the constraint
- ▶ If we have N cities, the chromosome has length N

Many possible schemes

- ▶ Crossover
 - ▶ One point
 - ▶ Two point
 - ▶ None
- ▶ Mutation
 - ▶ Normal random - a few cities are chosen and exchanged
 - ▶ Random, only improving - a few cities are randomly chosen and exchanged only if they improve solution (increase fitness)
 - ▶ None - no mutation

TSP example



5 More sophisticated algorithms

5.5 Genetic algorithms: further details

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

Parameters of GAs

- ▶ crossover probability
- ▶ mutation probability
- ▶ population size

Parameters: crossover probability

- ▶ If there is no crossover, offspring are exact copies of parents
 - ▶ but this doesn't mean the population is the same
- ▶ If there is crossover, offspring are made from parts of both parent's genotype (often just one chromosome)
- ▶ Crossover is made in hope that new chromosomes will contain good parts of old chromosomes and therefore the new chromosomes will be better. However, it is good to leave some part of old population survive to next generation.
- ▶ Crossover rate should be high generally, about 80%-95% (though it can vary)

Parameters: mutation probability

- ▶ Mutation prevents the GA from falling into local extrema
- ▶ Mutation rate
 - ▶ if there 0% mutation, can get stuck
 - ▶ if mutation probability is 100%, whole chromosome is changed, and search is purely random
 - ▶ rate expressed in terms of individual genes, so mutation rate should be very low
 - ▶ best rates seems to be about 0.5%-1%

- ▶ Too small a population there are
 - ▶ few possibilities to perform crossovers
 - ▶ too small part of search space covered
- ▶ Too large a population
 - ▶ GA slows down
 - ▶ at some point hit diminishing returns
- ▶ Good population size is about 20-30, however sometimes sizes 50-100 are reported as the best
 - ▶ Some research also shows, that the best population size depends on the size of chromosomes, e.g., for chromosomes with 32 bits, the population should be higher than for chromosomes with 16 bits.



- ▶ even though simulated annealing and genetic algorithms are called random algorithms they are not completely random
- ▶ it's not just randomly testing solutions
- ▶ we use a stochastic process
- ▶ however the result is highly non-random

There are other randomized algorithms

- ▶ **Simulated Annealing:** metaphor is cooling/crystal formation
- ▶ **Ants:** metaphor is a colony of ants (simple agents) running simple rules, to achieve highly organized collective behaviour (also called Swarm Intelligence)
- ▶ **Tabu search:** iteratively try to find solutions to the problem, but to keep a short list of previously found solutions and to avoid 're-finding' those solutions in subsequent iterations. Basically, if you try a solution, it becomes tabu in future tries [Glo90].

- ▶ Randomisation can be helpful
 - ▶ has to be controlled somehow though
- ▶ Genetic algorithms
 - ▶ exploit the idea of Darwinian evolution
 - ▶ very general, powerful technique
 - ▶ often needs some experimentation though

5 More sophisticated algorithms

5.6 Branch and Bound

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

Are “heuristics” the only approach?

- ▶ We are solving ILPs (Integer Linear Programs)
- ▶ So far have considered heuristics
 - ▶ assumption is there is no tractable method to guarantee a solution
 - ▶ but complexity analysis is about “worst case”
 - ▶ also, we might have

$$O(\exp(n)) = 0.0000000001 \times e^n$$

- ▶ typical cases might be quite tractable
- ▶ So can we find an algorithm that works well when the problem is notionally NP-hard, but the particular instance isn't too bad?

Example ILP



THE UNIVERSITY
of ADELAIDE

Example

Consider the Knapsack Problem we considered earlier (which is a Binary Linear Program). A hiker can choose from the following items:

Item	1 chocolate	2 raisins	3 camera	4 jumper	5 drink
w_i (kg)	0.5	0.4	0.8	1.6	0.6
v_i (value)	2.75	2.5	1	5	3.0
v_i/w_i	5.5	6.25	1.25	3.125	5

The hiker wants to maximise the value of the carried items subject to a total weight constraint of 2.5 kg, i.e., in general solve

$$\max \left\{ \sum_i v_i z_i \mid \sum_i w_i z_i \leq W, z_i = 0 \text{ or } 1 \right\}$$

where the z_i are binary indicator variables for each item.



Let's see what intlinprog does

... if you ask it not to use heuristics.

OUTPUT:

```
Solving problem using intlinprog.  
LP: Optimal objective value is -11.375000.
```

Branch and Bound:

nodes explored	total time (s)	num int solution	integer fval	relative gap (%)
1	0.00	1	-9.250000e+00	1.707317e+01
1	0.00	2	-1.025000e+01	6.666667e+00
3	0.00	2	-1.025000e+01	6.666667e+00

Optimal solution found.

SOLUTION: $\mathbf{z} = (1, 1, 0, 1, 0)^T$ and the value is 10.25



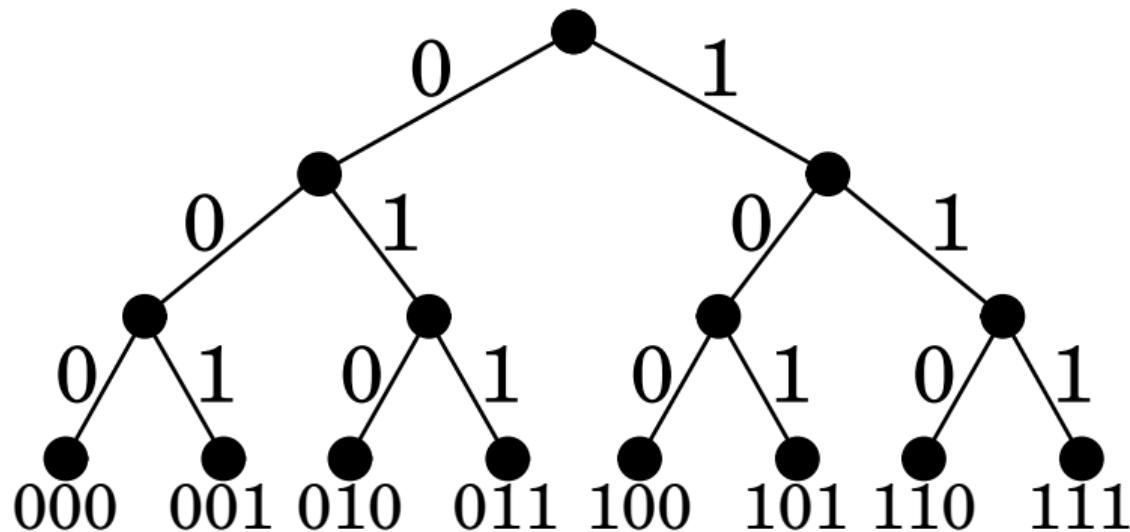
- ▶ intlinprog is using a method called “Branch & Bound”
 - ▶ it found the optimum solution
 - ▶ it “knows” it is the correct solution
 - ▶ somehow it used Simplex on the way?
- ▶ The goal of this lecture is to explain B&B

Branching

Imagine we are solving a Binary Linear Program, e.g.,

$$(BLP) \quad z^* = \max \{ \mathbf{c}^T \mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0, \mathbf{x} \in \{0, 1\}^n \}$$

Then we can *enumerate* all of the possible solutions on a tree



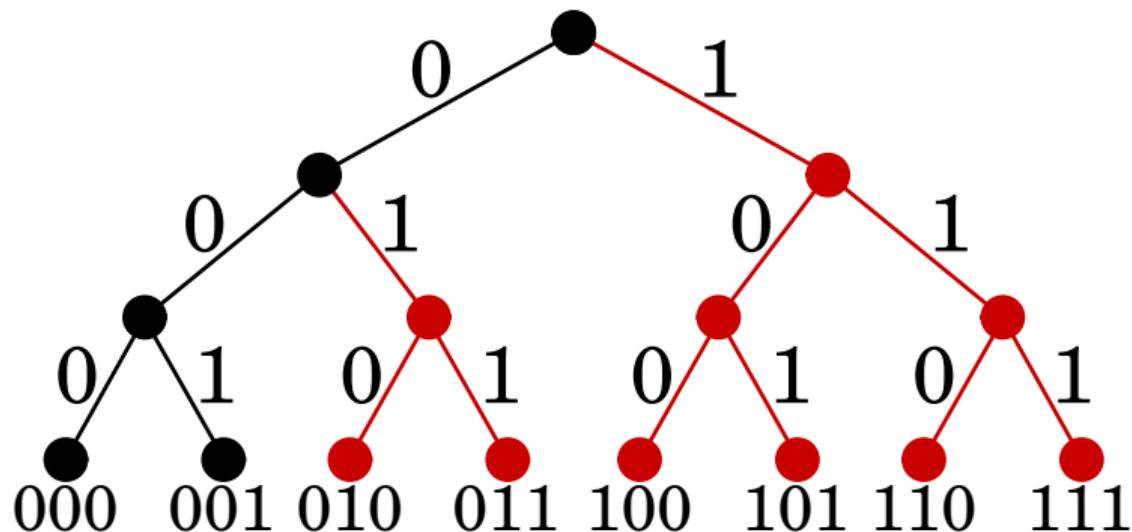
But there are 2^n solutions – we can't evaluate them all

Branching and Pruning

Imagine we are solving a Binary Linear Program, e.g.,

$$(BLP) \quad z^* = \max \{ \mathbf{c}^T \mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0, \mathbf{x} \in \{0, 1\}^n \}$$

What if we could eliminate some sub-branches



We don't have to search the whole tree

- ▶ Pruning reduces the search space
 - ▶ hopefully to the point where we can search the entire space
- ▶ Requires
 - ▶ a method to branch for general ILPs
 - ▶ binary branching, even when the problem isn't binary
 - ▶ a method to find “solutions” part way down a branch
 - ▶ a method to determine when a branch can be pruned
 - ▶ we will use *bounds* created by *relaxations*



Branching of ILPs

- ▶ Branching of Binary IPs
 - ▶ pick a variable z_i
 - ▶ left branch has $z_i = 0$, right branch has $z_i = 1$
 - ▶ in either case z_i is no longer a “variable”
 - ▶ we have *partitioned* the feasible solutions into two sets
 - ▶ divide and conquer
- ▶ Generalise the idea for Integer LPs
 - ▶ partition the set into two parts
 - ▶ pick a variable x_i and a divider c (which is NOT an integer)
 - ▶ left branch is $x_i \leq \lfloor c \rfloor$ and right branch is $x_i \geq \lceil c \rceil$

$\lfloor c \rfloor =$ the floor of c

$\lceil c \rceil =$ the ceiling of c

- ▶ x_i is still a variable, but on a restricted space

Example

Consider the Integer Linear Program

$$\begin{aligned} \max z &= x + y \\ \text{s.t.} \quad -x + 2y &\leq 8 \\ 23x + 10y &\leq 138 \end{aligned}$$

for non-negative integers x and y .

Branch on x at $c = 3.5$, and we get two new LPs $\max z = x + y$ such that

$$\begin{array}{rcl} -x + 2y &\leq& 8 \\ 23x + 10y &\leq& 138 \\ x &\leq& 3 \end{array} \quad \text{and} \quad \begin{array}{rcl} -x + 2y &\leq& 8 \\ 23x + 10y &\leq& 138 \\ x &\geq& 4 \end{array}$$



OPTIMISATION & OPERATIONS RESEARCH

5 More sophisticated algorithms

5.7 Relaxation gives bounds

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE



Relaxation: a reminder

- *Relaxation* means defining a new problem with some of the original constraints dropped
 - in this context, we drop some of the integrality constraints

Example (continued)

$$\begin{aligned} \max z &= x + y \\ \text{s.t.} \quad -x + 2y &\leq 8 \\ 23x + 10y &\leq 138 \\ x, y &\in \mathbb{Z}^+ \end{aligned}$$

Relax the integer constraints, *i.e.*, form a new problem (LP_0) with $x, y \in \mathbb{R}^+$. Solving (LP_0) gives the optimal solution as

$$z_0^* = 9\frac{1}{4} \quad \text{at} \quad (x_0^*, y_0^*)^T = \left(3\frac{1}{2}, 5\frac{3}{4}\right)^T$$

- ▶ *Relaxation* means defining a new problem with some of the original constraints dropped
 - ▶ in this context, we drop some of the integrality constraints
- ▶ Remember that in relaxing an ILP to a LP
 - ▶ the solution to the LP might not be close to that of the ILP
 - ▶ a feasible LP might not indicate a feasible ILP
- ▶ So relaxation by itself isn't a good approach to solve an ILP
 - ▶ but we can use these to generate “partial” solutions to help search for a fully feasible solution



What can we tell from a relaxation?

For each Integer Linear Program:

$$(ILP) \quad z^* = \max\{\mathbf{c}^T \mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0, \mathbf{x} \in \mathbb{Z}^n\}$$

there is an associated relaxed Linear Program:

$$(LP_0) \quad z_0^* = \max\{\mathbf{c}^T \mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0, \mathbf{x} \in \mathbb{R}^n\}$$

Now, (LP_0) is *less constrained* than the (ILP) so

- ▶ If (LP_0) is infeasible, then so is (ILP)
- ▶ If (LP_0) is optimised by integer variables, then that solution is feasible and optimal for the (ILP)
- ▶ The optimal objective value for (LP_0) is greater than or equal to the optimal objective for the (ILP)

$$z_0^* \geq z^*$$

Relaxation Gives Bounds



THE UNIVERSITY
ofADELAIDE

- ▶ The relaxed problem is a LP
 - ▶ we know how to solve this, e.g., Simplex
- ▶ The relaxed LP tells us something about the ILP
 - ▶ it doesn't give the solution
 - ▶ it does provide an *upper bound* on the solution

Example (continued)

Solving (LP_0) gives the optimal solution as

$$z_0^* = 9\frac{1}{4} \quad \text{at} \quad (x_0^*, y_0^*)^T = \left(3\frac{1}{2}, 5\frac{3}{4}\right)^T$$

The ILP has solution

$$z^* = 8 \leq z_0^*$$

- ▶ We can use the bounds to prune branches



- ▶ Keep a list of *subproblems* resulting from branching, and work on these one by one
 - ▶ solve relaxed versions to get upper bounds
 - ▶ sometimes we might also get an integer solution
- ▶ *key:* if upper bound of a subproblem is less than objective for a known integer feasible solution, then
 - ▶ the subproblem cannot have a solution greater than the already known solution
 - ▶ we can eliminate this solution
 - ▶ we can also prune all of the tree below the solution
- ▶ it lets us do a *non-exhaustive* search of the subproblems
 - ▶ if we get to the end, we have a proof of optimality without exhaustive search

5 More sophisticated algorithms

5.8 Branch and Bound: algorithm

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

1. *Initialization*: initialize variables, in particular, start a list of subproblems, initialized with our original integer program.
2. *Termination*: terminate the program when we reach the optimum (*i.e.*, the list of subproblems is empty).
3. *Problem selection and relaxation*: select the next problem from the list of possible subproblems, and solve a relaxation on it.
4. *Fathoming and pruning*: eliminate branches of the tree once we prove they cannot contain an optimal solution.
5. *Branching*: partition the current problem into subproblems, and add these to our list.

Branch and Bound: example

Consider the problem (from [LM01])

$$\text{IP}^0 \left\{ \begin{array}{ll} \text{maximize} & 13x_1 + 8x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 10 \\ & 5x_1 + 2x_2 \leq 20 \\ & x_1 \geq 0, x_2 \geq 0 \\ & x_1, x_2 \text{ integer} \end{array} \right.$$

Branch and Bound: algorithm

Initialization:

- ▶ initialize the **list** of problems \mathcal{L}
 - ▶ set initially $\mathcal{L} = \{\text{IP}^0\}$, where IP^0 is the initial problem
 - ▶ often store/picture \mathcal{L} as a tree
- ▶ incumbent objective value $z_{ip} = -\infty$
 - ▶ best (integer) solution we have found so far
 - ▶ initial value is the worst possible
- ▶ initial value of upper bound on problem is $\bar{z}_0 = \infty$
 - ▶ If the upper bound of a solution $\bar{z}_i < z_{ip}$ then this problem IP^i (and its dependent tree) obviously cannot achieve the same objective value that we have already achieved elsewhere in our solutions.
- ▶ constraint set of problem IP^0 is set to be

$$S^0 = \{\mathbf{x} \in \mathbb{Z}^n \mid A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0\}$$

Termination:

- ▶ If $\mathcal{L} = \emptyset$ then we stop
 - ▶ If $z_{ip} = -\infty$ then the integer program is infeasible.
 - ▶ our search didn't find an integer feasible solution
 - ▶ Otherwise, the subproblem IP^i which yielded the current value of z_{ip} is optimal gives the optimal solution \mathbf{x}^*

We stop branch and bound when we have run out of subproblems (which are listed in \mathcal{L}) to solve, i.e., when \mathcal{L} is empty.

Problem selection:

- ▶ select a problem from \mathcal{L}
 - ▶ there are multiple ways to decide which problem to choose from the list
 - ▶ the method used can have a big impact on speed
 - ▶ once selected, delete the problem from the list

Relaxation:

- ▶ solve a relaxation of the problem
 - ▶ denote the optimal solution by \mathbf{x}^{iR}
 - ▶ denote the optimal objective value by z_i^R
 - ▶ $z_i^R = -\infty$ if no feasible solutions exist



Branch and Bound: algorithm

For the example

$$\text{IP}^0 \left\{ \begin{array}{ll} \text{maximize} & 13x_1 + 8x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 10 \\ & 5x_1 + 2x_2 \leq 20 \\ & x_1 \geq 0, x_2 \geq 0 \\ & x_1, x_2 \text{ integer} \end{array} \right.$$

the relaxation is

$$\text{LP}^0 \left\{ \begin{array}{ll} \text{maximize} & z = 13x_1 + 8x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 10 \\ & 5x_1 + 2x_2 \leq 20 \\ & x_1 \geq 0, x_2 \geq 0 \end{array} \right.$$

which has solutions $x_1^{0R} = 2.5$ and $x_2^{0R} = 3.75$ with $z_0^R = 62.5$



Branch and Bound: algorithm

Fathoming :

- ▶ we say branch of the tree is **fathomed** if
 - ▶ infeasible
 - ▶ feasible solution, and $z_i^R \leq z_{ip}$
 - ▶ integral feasible solution
 - ▶ set $z_{ip} \leftarrow \max\{z_{ip}, z_i^R\}$

Pruning:

- ▶ in any of the cases above, we need not investigate any more subproblems of the current problem
 - ▶ subproblems have more constraints
 - ▶ their z must lie under the upper bound
- ▶ Prune any subtrees with $z_j^R \leq z_{ip}$
- ▶ If we pruned *Goto step 2*

We don't prune the example yet (see later for complete example).

Branching:

- ▶ also called partitioning
- ▶ want to partition the current problem into subproblems
 - ▶ there are several ways to perform partitioning
- ▶ If S^i is the current constraint set, then we need a disjoint partition $\{S^{ij}\}_{j=1}^k$ of this set
 - ▶ we add problems $\{\text{IP}^{ij}\}_{j=1}^k$ to \mathcal{L}
 - ▶ typically $k = 2$ for binary branching
 - ▶ IP^{ij} is just IP^i with its feasible region restricted to S^{ij}
- ▶ *Goto step 2*

5 More sophisticated algorithms

5.9 Branch and Bound: example 1

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE



Branch and Bound: example

Consider the problem (from [LM01])

$$\text{IP}^0 \left\{ \begin{array}{ll} \text{maximize} & 13x_1 + 8x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 10 \\ & 5x_1 + 2x_2 \leq 20 \\ & x_1 \geq 0, x_2 \geq 0 \\ & x_1, x_2 \text{ integer} \end{array} \right.$$

with relaxation

$$\text{LP}^0 \left\{ \begin{array}{ll} \text{maximize} & z = 13x_1 + 8x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 10 \\ & 5x_1 + 2x_2 \leq 20 \\ & x_1 \geq 0, x_2 \geq 0 \end{array} \right.$$

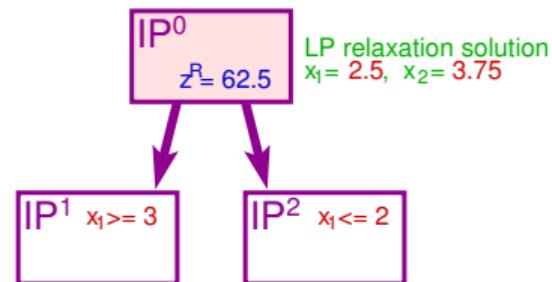
which has solutions $x_1^0 = 2.5$ and $x_2^0 = 3.75$ with $z_0^R = 62.5$

Branch and Bound: algorithm

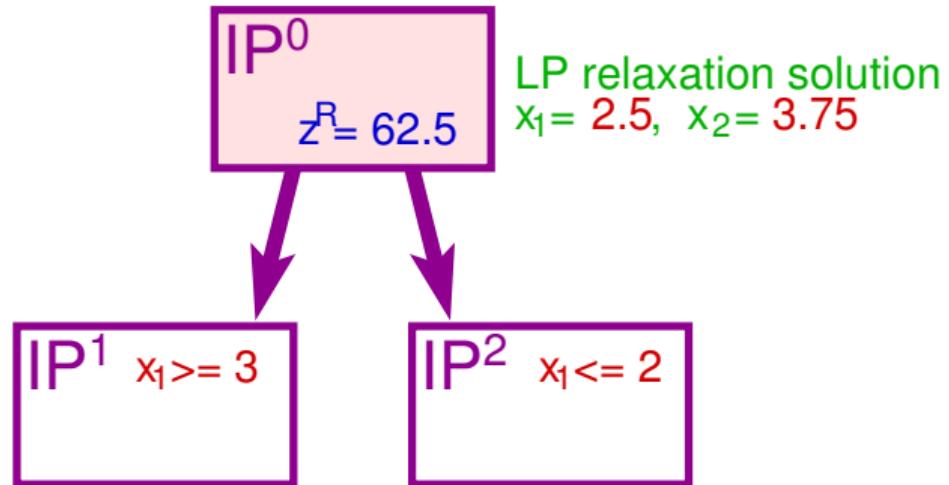
In the example we partition on x_1

- ▶ this is the “most infeasible”
 - ▶ furthest from an integral value (because $x_1^0 = 2.5$)
- ▶ partition into two subproblems around $c = 2.5$
 - ▶ IP¹ has $x_1 \geq 3$
 - ▶ IP² has $x_1 \leq 2$

So now $\mathcal{L} = \{\text{IP}^1, \text{IP}^2\}$



Branch and Bound: example





Branch and Bound: example

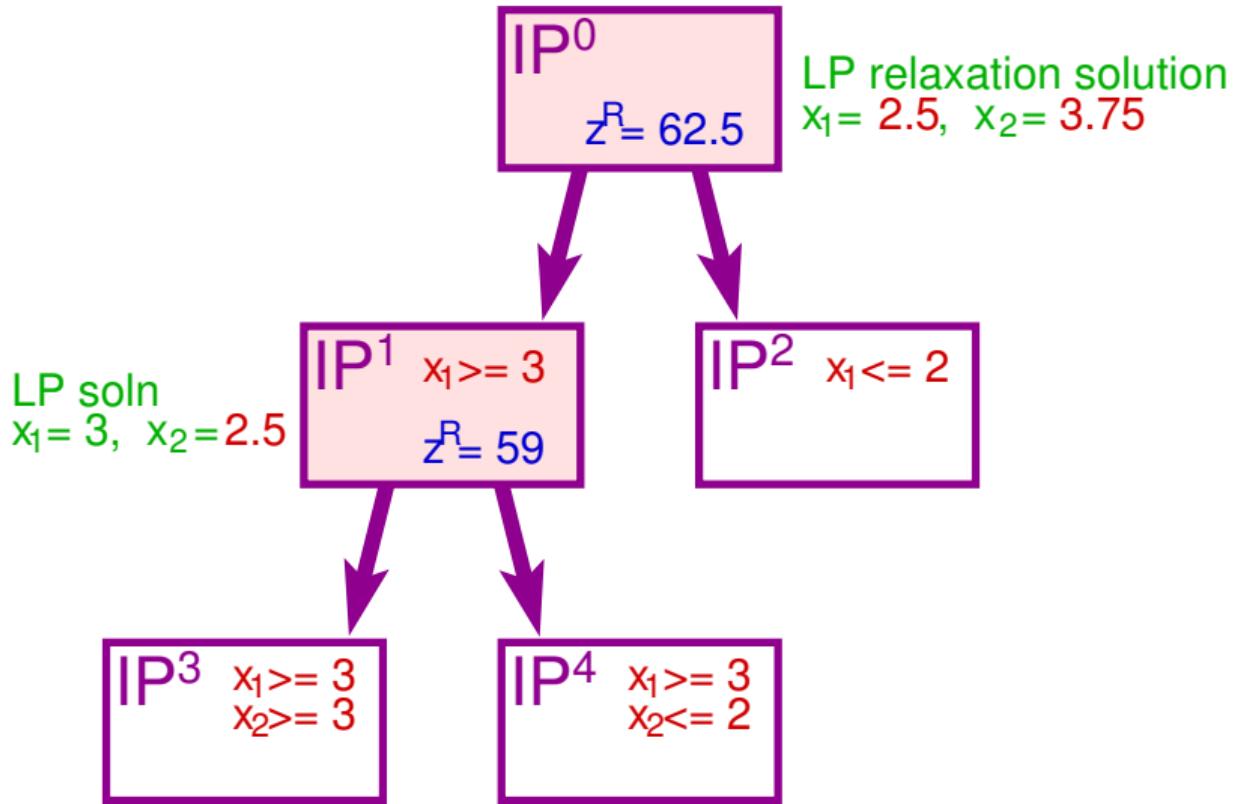
Problem selection (just chose in order) of IP¹

$$\left. \begin{array}{ll} \text{maximize} & 13x_1 + 8x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 10 \\ & 5x_1 + 2x_2 \leq 20 \\ & x_1 \geq 3 \\ & x_1 \geq 0, x_2 \geq 0 \\ & x_1, x_2 \text{ integer} \end{array} \right\} \text{IP}^1$$

The relaxation (to a LP) has solutions

- ▶ $x_1^1 = 3$ and $x_2^1 = 2.5$ with $z_1^R = 59$
- ▶ we will next partition on x_2
 - ▶ IP³ has $x_2 \leq 2$
 - ▶ IP⁴ has $x_2 \geq 3$

Branch and Bound: example





Branch and Bound: example

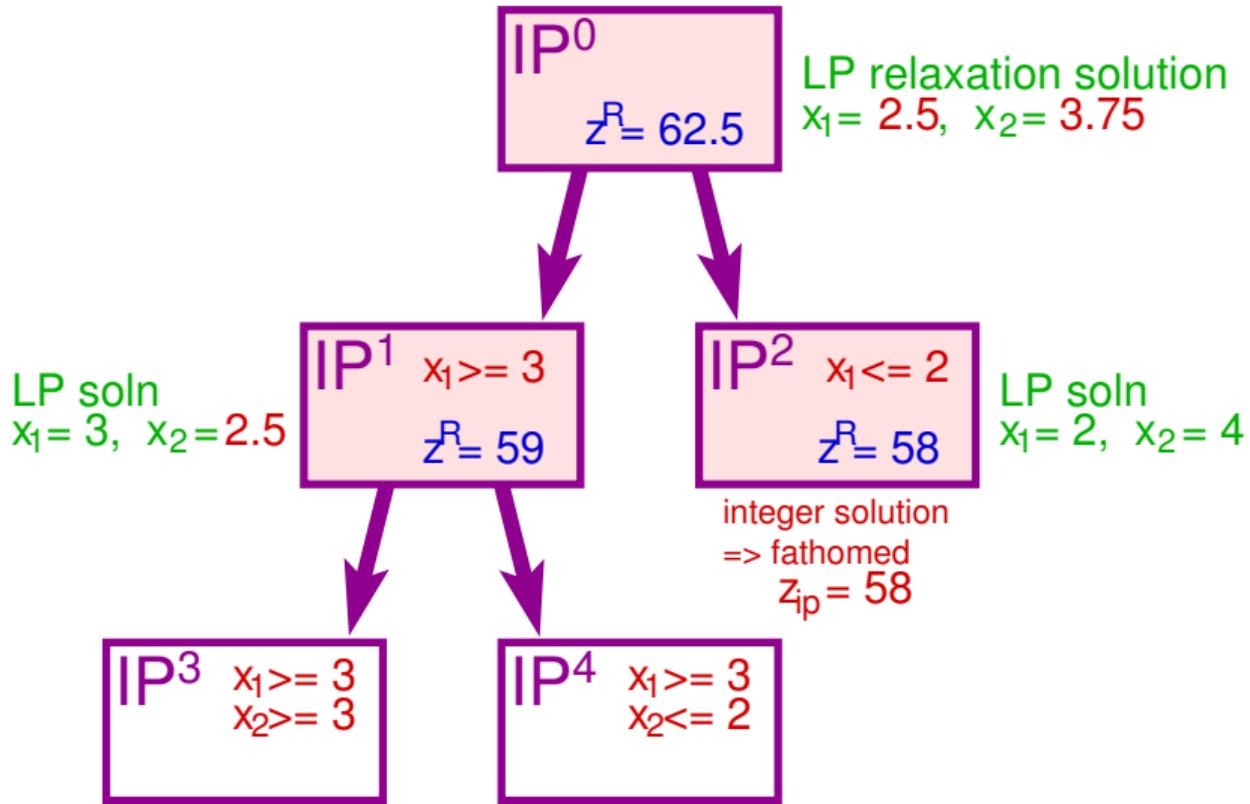
Problem selection (best bound) of IP²

$$\left. \begin{array}{ll} \text{maximize} & 13x_1 + 8x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 10 \\ & 5x_1 + 2x_2 \leq 20 \\ & x_1 \leq 2 \\ & x_1 \geq 0, x_2 \geq 0 \\ & x_1, x_2 \text{ integer} \end{array} \right\} \text{IP}^2$$

The relaxation (to a LP) has solutions

- ▶ $x_1^2 = 2$ and $x_2^2 = 4$ with $z_2^R = 58$
- ▶ *integral feasible*
- ▶ So set $z_{ip} = 58$
- ▶ And IP² is *fathomed*
 - ▶ no more subproblems

Branch and Bound: example





Branch and Bound: example

Problem selection (order) of IP³

$$\left. \begin{array}{ll} \text{maximize} & 13x_1 + 8x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 10 \\ & 5x_1 + 2x_2 \leq 20 \\ & x_1 \geq 3 \\ & x_2 \geq 3 \\ & x_1 \geq 0, x_2 \geq 0 \\ & x_1, x_2 \text{ integer} \end{array} \right\} \text{IP}^3$$

The relaxation (to a LP) is infeasible

- ▶ $z_3^R = -\infty$
- ▶ IP³ is fathomed
- ▶ $\mathcal{L} = \{\text{IP}^4\}$



Branch and Bound: example

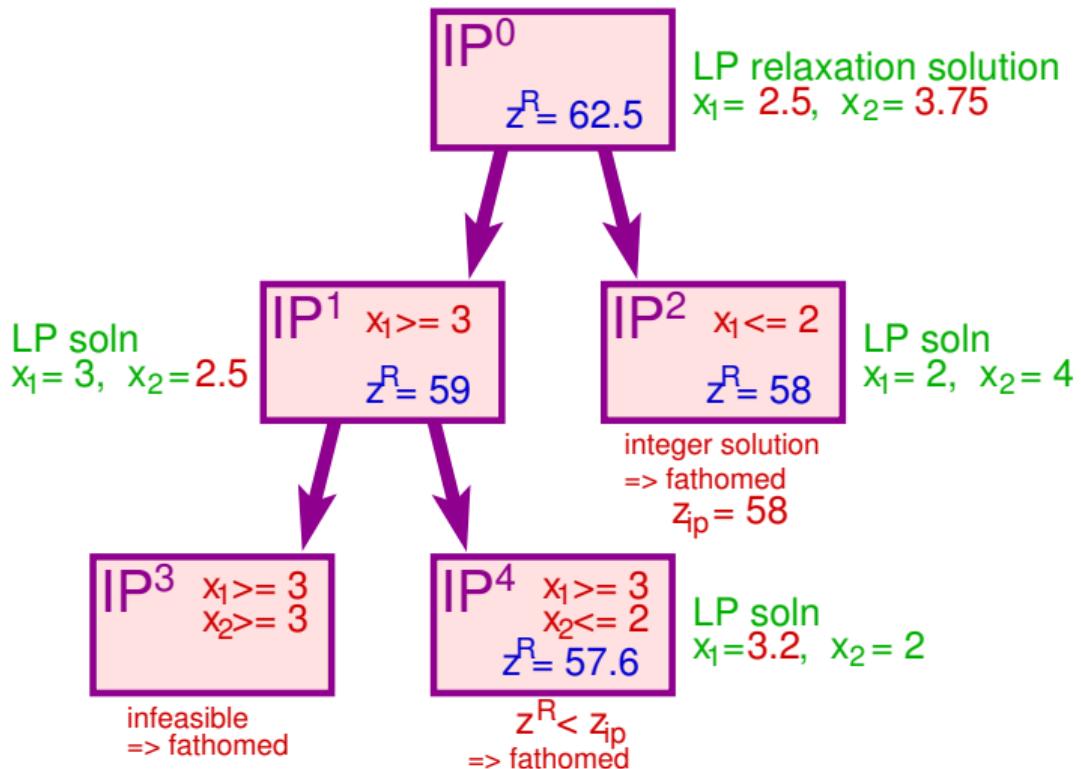
Problem selection (only possible one) of IP⁴

$$\left. \begin{array}{ll} \text{maximize} & 13x_1 + 8x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 10 \\ & 5x_1 + 2x_2 \leq 20 \\ & x_1 \geq 3 \\ & x_2 \leq 2 \\ & x_1 \geq 0, x_2 \geq 0 \\ & x_1, x_2 \text{ integer} \end{array} \right\} \text{IP}^4$$

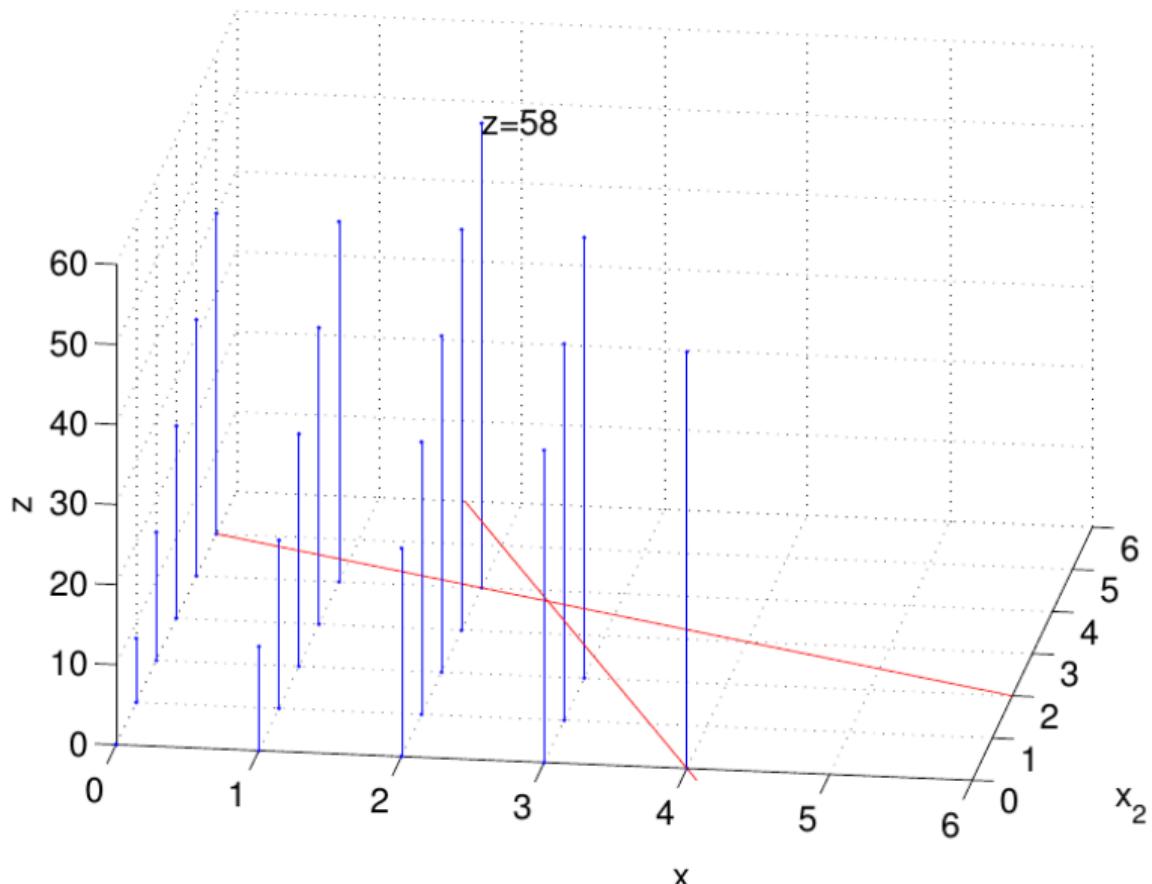
The relaxation (to a LP) has solution

- ▶ $x_1^2 = 3.2$ and $x_2^2 = 2$ with $\boxed{z_4^R = 57.6 < z_{ip}}$
- ▶ IP⁴ is fathomed

Branch and Bound: example



Branch and Bound: example



- ▶ B&B uses pruning to perform a non-exhaustive search
 - ▶ we can prune branches when they are
 - ▶ infeasible
 - ▶ integer feasible
 - ▶ their upper bound (on their relaxation) is less than an existing solution

5 More sophisticated algorithms

5.10 Branch and bound example 2: knapsack problem

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

Branch and Bound Example

Example (Knapsack problem)

Consider the binary linear program

$$\begin{aligned} \max z &= 100x_1 + 60x_2 + 70x_3 \\ \text{s.t.} \quad & 52x_1 + 23x_2 + 35x_3 \leq 60 \\ & x_1 \in \{0, 1\} \\ & x_2 \in \{0, 1\} \\ & x_3 \in \{0, 1\} \end{aligned}$$

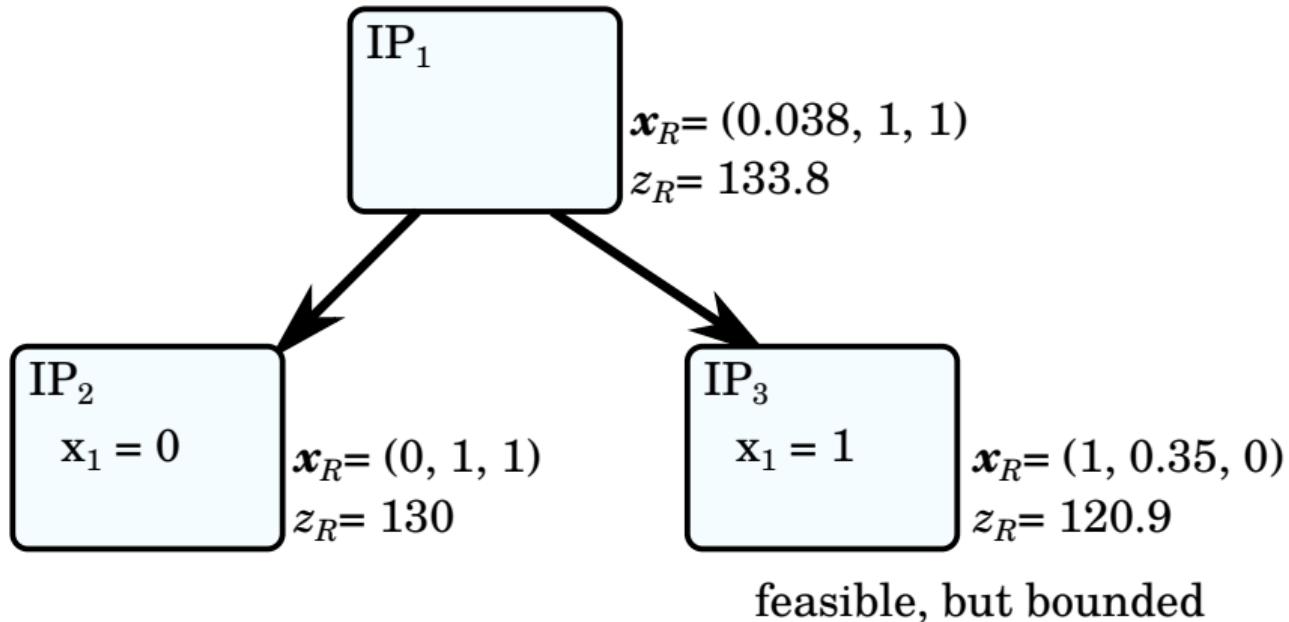
This is a knapsack problem where no more than 1 of each item is to be packed.

NB: the relative merit of the items, is $(\frac{100}{52}, \frac{60}{23}, \frac{70}{35})$ so a Greedy algorithm would get the solution $(1, 0, 0)$ with a value of $z = 100$.

Branch and Bound Example



THE UNIVERSITY
of ADELAIDE





Example (continued)

The relaxation of IP_1 has optimal solution

$$z^* = 133\frac{44}{52} \quad \text{at} \quad \mathbf{x}^* = \left(\frac{2}{52}, 1, 1 \right)^T.$$

Since the first entry is non-integral, this cannot be the optimal solution of the (ILP) and so we branch on x_1 as follows

- (a) (IP_2) : (IP_1) with $x_1 = 0$
- (b) (IP_3) : (IP_1) with $x_1 = 1$.

So the list of problems is $\mathcal{L} = \{IP_2, IP_3\}$

Example (continued)

The relaxation of IP_2 has optimal solution

$$z^* = 130 \quad \text{at} \quad \mathbf{x}^* = (0, 1, 1)^T.$$

This is integer, feasible, so it is a viable solution to the original ILP. Thus we store its objective value

$$z_{ip} = z^* = 130$$

and this branch is considered fathomed.

Example (continued)

The relaxation of IP_3 has optimal solution

$$z^* = 120 \frac{20}{23} \quad \text{at} \quad \mathbf{x}^* = \left(1, \frac{8}{23}, 0\right)^T.$$

This is non-integral, but z^* (the upper bound for IP_3 obtained from the relaxation) is already below $z_{ip} = 130$, so this solution is bounded, and hence fathomed.

But it's interesting to consider what we would have done if we tried to solve IP_3 *before* IP_2 . Then we would have branched on this case, and had a longer list of problems to solve!

A circular graphic on the left side of the slide features a complex network of nodes (small circles) connected by lines (edges). The colors of the nodes range from white to red, blue, and yellow, set against a dark blue background with blurred circular highlights.

OPTIMISATION & OPERATIONS RESEARCH

5 More sophisticated algorithms

5.11 Branch and bound: further example

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

Example

Consider the Integer Linear Program

$$\begin{aligned} \max z &= x_1 + x_2 \\ \text{s.t.} \quad -x_1 + 2x_2 &\leq 8 \\ 23x_1 + 10x_2 &\leq 138 \\ x_1, x_2 &\in \mathbb{Z}^+ \end{aligned}$$



Let's see what intlinprog does

OUTPUT:

```
Solving problem using intlinprog.  
LP: Optimal objective value is -9.250000.
```

```
Branch and Bound:
```

nodes explored	total time (s)	num int solution	integer fval	relative gap (%)
3	0.00	1	-8.000000e+00	1.111111e+01
3	0.00	1	-8.000000e+00	1.111111e+01

```
Optimal solution found.
```

SOLUTION: $x_1 = 3, x_2 = 5, z = 8$

Example (continued)

Relaxed solution to the original LP is $(x_1^*, x_2^*)^T = \left(3\frac{1}{2}, 5\frac{3}{4}\right)^T$

This is not integer feasible, so we select one of these (non-integer) variables, such as $x_1^* = 3\frac{1}{2}$ and branch as follows

- (a) $x_1 \leq 3$
- (b) $x_1 \geq 4$

Note that we now have 2 regions, which are mutually exclusive. This forced dichotomy is the “branching” part of “branch-and-bound”. Here we have chosen x_1 as the branching variable (and have “cut off” a strip of x_1 values $3 < x_1 < 4$). This helps force our solution of the (relaxed) (LP) to be integer and hopefully towards the solution of the original (ILP).



Example Branching

Example (continued)

After 1st branching, we have 2 (LP) problems to solve:

$$(LP_2) \quad \begin{aligned} \max z &= x_1 + x_2 \\ \text{s.t.} \quad &-x_1 + 2x_2 \leq 8 \\ &23x_1 + 10x_2 \leq 138 \\ &x_1 \leq 3 \\ &x_1, x_2 \geq 0 \end{aligned}$$

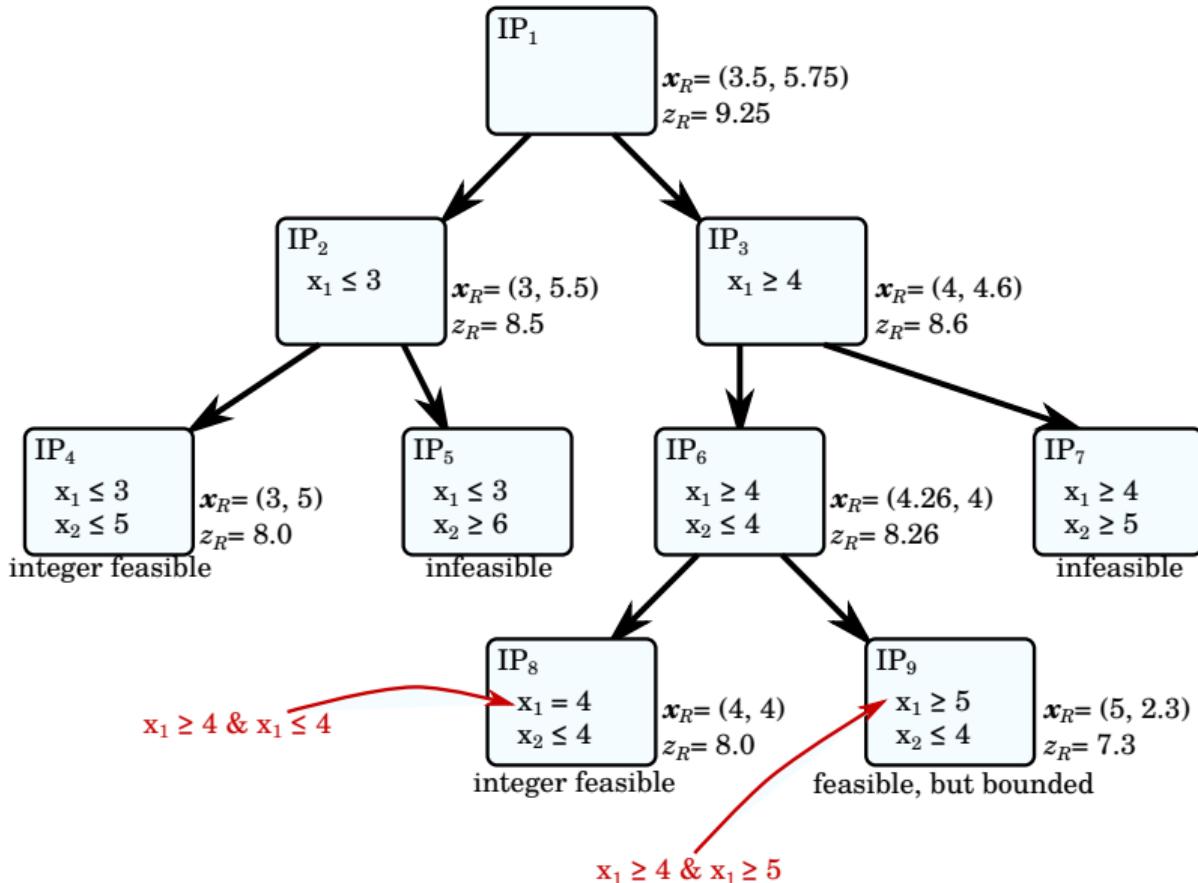
and

$$(LP_3) \quad \begin{aligned} \max z &= x_1 + x_2 \\ \text{s.t.} \quad &-x_1 + 2x_2 \leq 8 \\ &23x_1 + 10x_2 \leq 138 \\ &x_1 \geq 4 \\ &x_1, x_2 \geq 0 \end{aligned}$$

Example B&B



THE UNIVERSITY
of ADELAIDE



Example (continued)

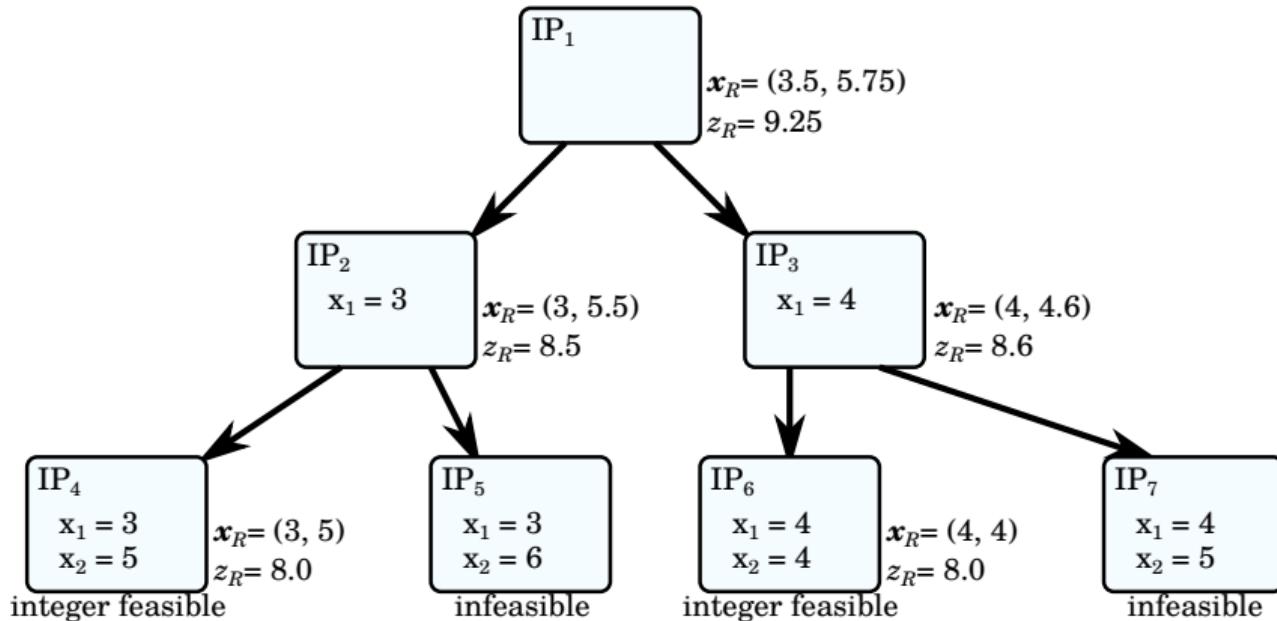
We still have two 2 (LP) problems to solve, but they are “tighter”

$$\begin{aligned} (LP_2) \quad \max z &= x_1 + x_2 \\ \text{s.t.} \quad -x_1 + 2x_2 &\leq 8 \\ 23x_1 + 10x_2 &\leq 138 \\ x_1 &= 3 \quad \text{note the equality} \\ x_1, x_2 &\geq 0 \end{aligned}$$

and

$$\begin{aligned} (LP_3) \quad \max z &= x_1 + x_2 \\ \text{s.t.} \quad -x_1 + 2x_2 &\leq 8 \\ 23x_1 + 10x_2 &\leq 138 \\ x_1 &= 4 \quad \text{note the equality} \\ x_1, x_2 &\geq 0 \end{aligned}$$

Example B&B



- ▶ Note that now we don't have to solve as many sub-problems as previous case
- ▶ In fact, we only need to go two steps down any branch, at worst

5 More sophisticated algorithms

5.12 Branch and bound: practical considerations

Dr Mike Chen
School of Mathematical Sciences
The University of Adelaide



THE UNIVERSITY
*of*ADELAIDE

The order of problem selection matters!

- ▶ I have just done them in the order added
- ▶ could do other simple approaches: e.g., depth first or breadth first
- ▶ other approaches
 - ▶ **Best bound rule:**
We partition the subset with the lowest bound, hoping that this gives the best chance of an optimal solution and of being able to discard other, larger, subsets by fathoming.
 - ▶ **Newest bound rule:**
We partition the most recently created subset, breaking ties with the best bound rule. This has book-keeping advantages, in that we don't need to jump around the tree too often. It can also save some computational effort involved in calculating bounds.

But there is no universal “best” order.



- ▶ B&B can be made even faster
 - ▶ sometimes we know something specific about the problem that helps derive better bounds
 - ▶ we don't solve each relaxation from scratch – we already have a starting point (e.g., see sensitivity analysis when we add a constraint)
- ▶ B&B is a very general algorithm
 - ▶ as described above we seek the optimum
 - ▶ can also be used as part of a heuristic
- ▶ different strategies available for each step above
 - ▶ can use heuristics inside B&B
 - ▶ pre-processing of the problem can be good
- ▶ no single strategy stands out as best for all problems
 - ▶ but sometimes we can exploit properties of a particular problem to do better

- ▶ We have a variety of approaches to attack ILPs
 - ▶ heuristics
 - ▶ simple to program
 - ▶ very problem dependent
 - ▶ often quite fast
 - ▶ B&B
 - ▶ more general (solves general ILPs)
 - ▶ harder work to program (not too much harder)
 - ▶ potentially slow for big problems
 - ▶ others ...
- ▶ But there is no “one size fits all” solution

Further reading I

-  E.W. Dijkstra, *A note in two problems in connexion with graphs*, Numerische Mathematik **1** (1959), 269–271.
-  Martin Gardner, *Mathematical games*, Scientific American (1972), no. 2, 106.
-  F. Glover, *Tabu search: A tutorial*, Interfaces **20** (1990), no. 4, 74–94.
-  F. Gray, *Pulse code communication*, U. S. Patent 2 632 058, March 17 1953.
-  J. J. Grenfenstette (ed.), *Proceedings of the first international conference on genetic algorithms and their applications*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1985.
-  J. H. Holland, *Adaptation in natural and artificial systems*, University of Michigan Press, Ann Arbor, MI, 1975.
-  Bernhard Korte and Jens Vygen, *Combinatorial optimization*, Springer, 2000.

Further reading II

-  Eva K. Lee and John Mitchell, *Encyclopedia of optimization*, ch. Branch-and-bound methods for integer programming, Kluwer Academic Publishers, 2001,
<http://www.rpi.edu/~mitchj/papers/leeejem.html>.
-  J. D. Schaffer (ed.), '*proceedings of the third international conference on genetic algorithms*', Morgan Kaufmann Publishers, Inc., 1989.
-  Aman Shaikh and Albert Greenberg, *Experience in black-box OSPF measurement*, Proc. ACM SIGCOMM Internet Measurement Workshop, 2001, pp. 113–125.