Optimisation and tools

Linear programming, Integer programming and combinatorial optimisation

AALBORG UNIVERSITY
DENMARK

Mohamed EL YAFRANI

- Postdoc researcher @ Operations Research group - AAU
- Background and research interests:
  - Computer science
  - Combinatorial optimisation
  - Metaheuristics and evolutionary computation

AALBORG UNIVERSITY
DENMARK

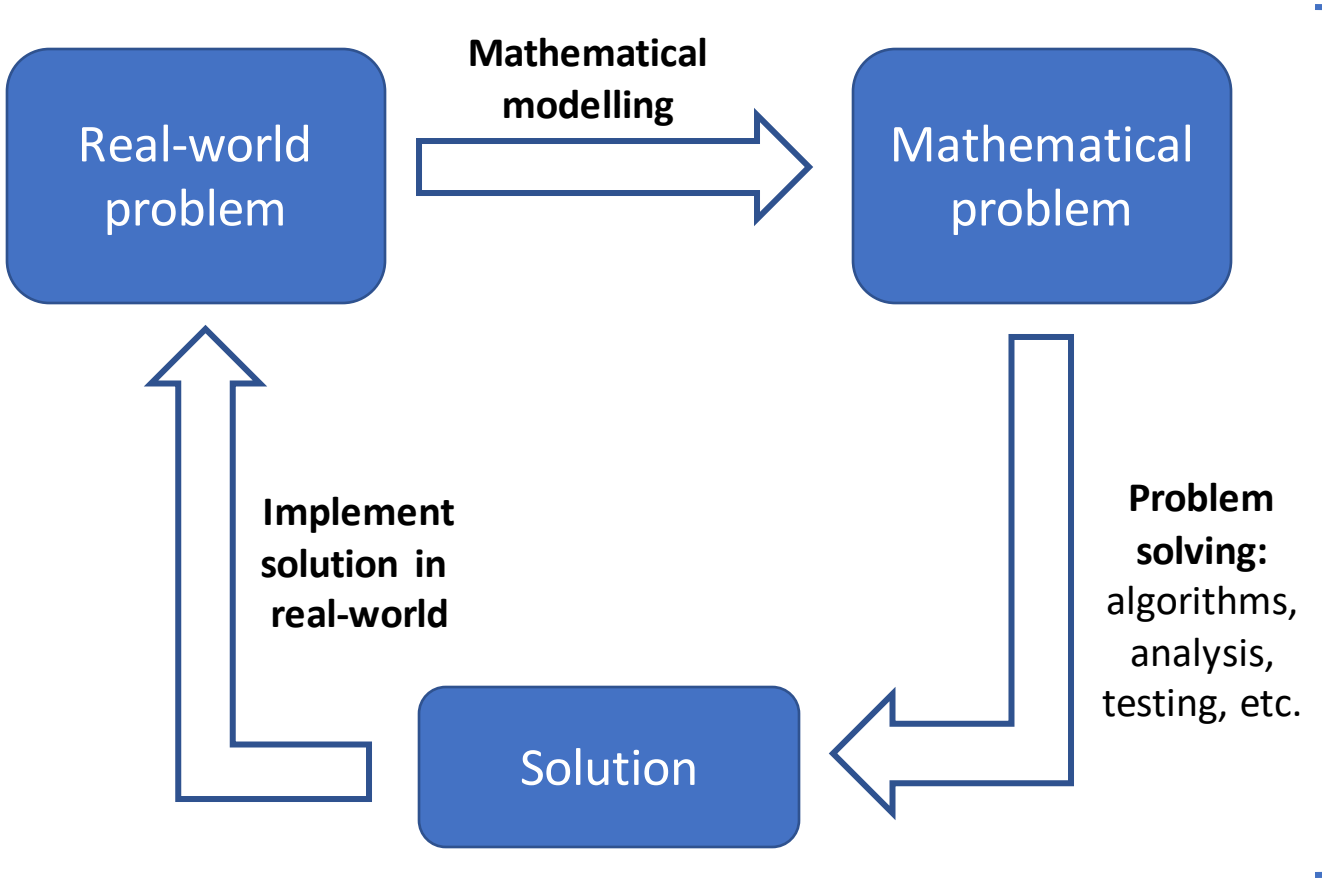
Outline

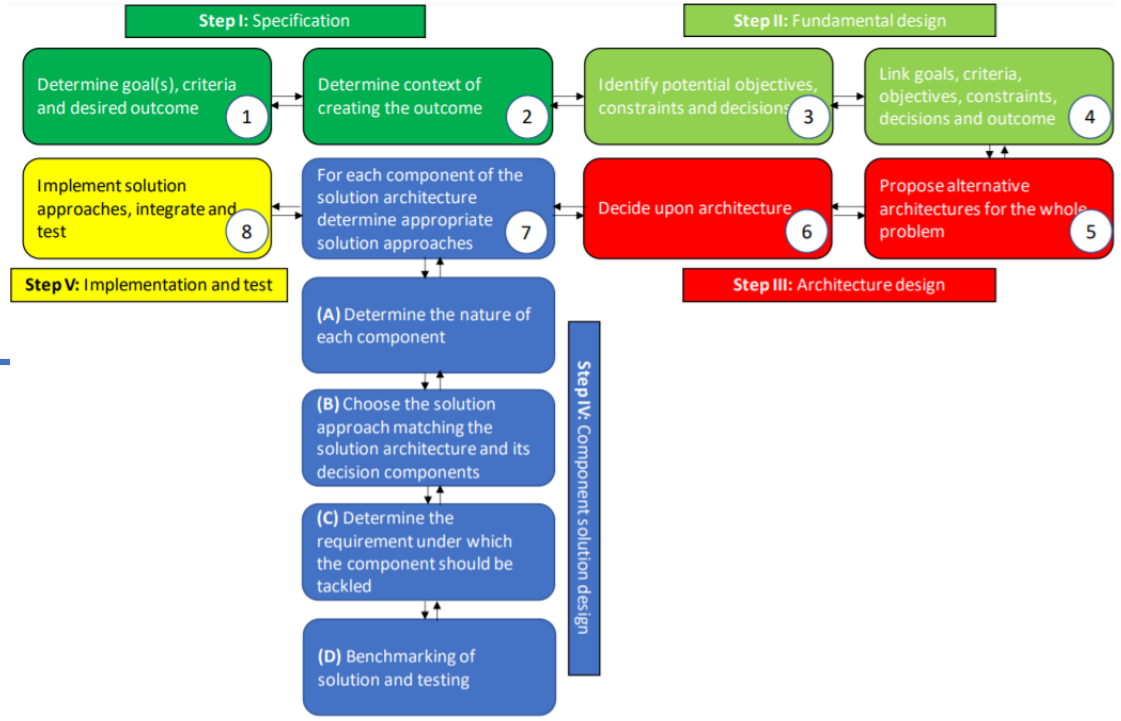
- Linear programming
  - Gentle introduction with simple examples
- Integer programming
  - Gentle introduction
  - The branch & bound algorithms
- Combinatorial optimisation and metaheuristics
  - Combinatorial optimisation and the computational burden
  - Constructive algorithms and local search
  - Metaheuristics

Linear programming

AALBORG UNIVERSITY
DENMARK

Problem solving cycle





Linear programming

- Linear programming (LP) is
  - a widely used mathematical technique for modelling and problem solving
  - designed to help managers in planning and decision-making
  - a technique that can help in resource allocation, routing, scheduling, etc.
- **Linear** means the equations and functions used in modelling the problems are linear
- **Programming** refers to modelling and solving a problem mathematically

Linear programming – properties

- All problems seek to maximise or minimise some quantity
  - e.g. maximise profit, minimise cost, minimise risk, etc.
  - This quantity is called the **objective function**
- The presence of restrictions limits the degree to which we can pursue our objective
  - These limits are called **constraints**
  - Constraints are expressed as equations or inequalities
- There must be alternative courses of action to choose from
  - The problem has more than one possible solution
- The objective and constraints in linear programming problems must be expressed in terms of **linear** equations or inequalities

Linear programming – important assumptions

Certainty:

- Values in the objective function and constraints are known with certainty and do not change during the period being studied

Proportionality:

- Exists in the objective and constraints
- Constancy between production increases and resource utilisation

Additivity:

- The total of all activities equals the sum of the individual activities

Divisibility:

- Solutions do not need to be whole numbers (integers)
- Solutions are divisible, and may take any fractional value

Non-negativity:

- All answers (decision variables) are ≥ zero
- Negative values of physical quantities are impossible

Linear programming – formulation

- Formulating a linear program involves developing a mathematical model to represent the problem.
- Once the problem is understood, begin to develop the mathematical statement of the problem.
- Steps in formulating LP problems:
    1. Completely understand the problem being faced
    2. Identify the objective and the constraints
    3. Define the decision variables
    4. Use the decision variables to write mathematical expressions for the objective function and the constraints

Linear programming – a simple example

A typical example of a LP problem is the **Product Mix** problem:

- Two or more products are usually produced using **limited resources** such as personnel, machines, raw materials, and so on.
- The profit that the firm seeks to maximise is based on the profit contribution per unit of each product.
- The company would like to determine how many units of each product it should produce so as to **maximise overall profit** given its limited resources.

Linear programming – a simple example

Product Mix problem Tables and Chairs

- The problem faced is **what to produce?**
  - **More concretely:** how many tables and how many chairs to produce?

Hours required to produce one unit

Department	Tables	Chairs	Available hours
Carpentry	4	3	240
Painting & varniching	2	1	100
	$7	$5	

Resource limits

Profit per unit

Linear programming – a simple example

- Identifying the objectives:
    - In this example, the objective is to **maximise profit**
- And the constraints:
    - Hours of carpentry time used ≤ 240 hrs. per week
    - Hours of painting & varnishing used ≤ 100 hrs. per week.

Linear programming – a simple example

Define the decision variables:

Let:

T be the number of tables produced each week

C be the number of chairs produced each week

Linear programming – a simple example

Maximise $7T + 5C$

Subject to: $4T + 3C \leq 240$ (Carpentry)

$2T + 1C \leq 100$ (Painting & Varnishing)

$T \geq 0$ (1st nonnegative cons)

$C \geq 0$ (2nd nonnegative cons)

Notice that:

- the units have been disregarded
- decision variables are kept on one side of the inequality

Linear programming – a simple example

Maximise 7T + 5C

Right Hand Side (RHS)

Subject to:	4T + 3C	≤ 240	(Carpentry)
	2T + 1C	≤ 100	(Painting & Varnishing)
	T	≥ 0	(1st nonnegative cons)
	C	≥ 0	(2nd nonnegative cons)

Notice that:

- the units have been disregarded
- decision variables are kept on one side of the inequality

Linear programming – graphical method

- When only two decision variables exist, the simplest method for solving the problem might be the graphical solution approach.
- The graphical method works only when there are two decision variables, but it provides valuable insight into how larger problems are structured.
- Most real-world problems involve multiple decision variables and sometimes multiple objectives as well.

Linear programming – graphical method

Corner Point Solution Method

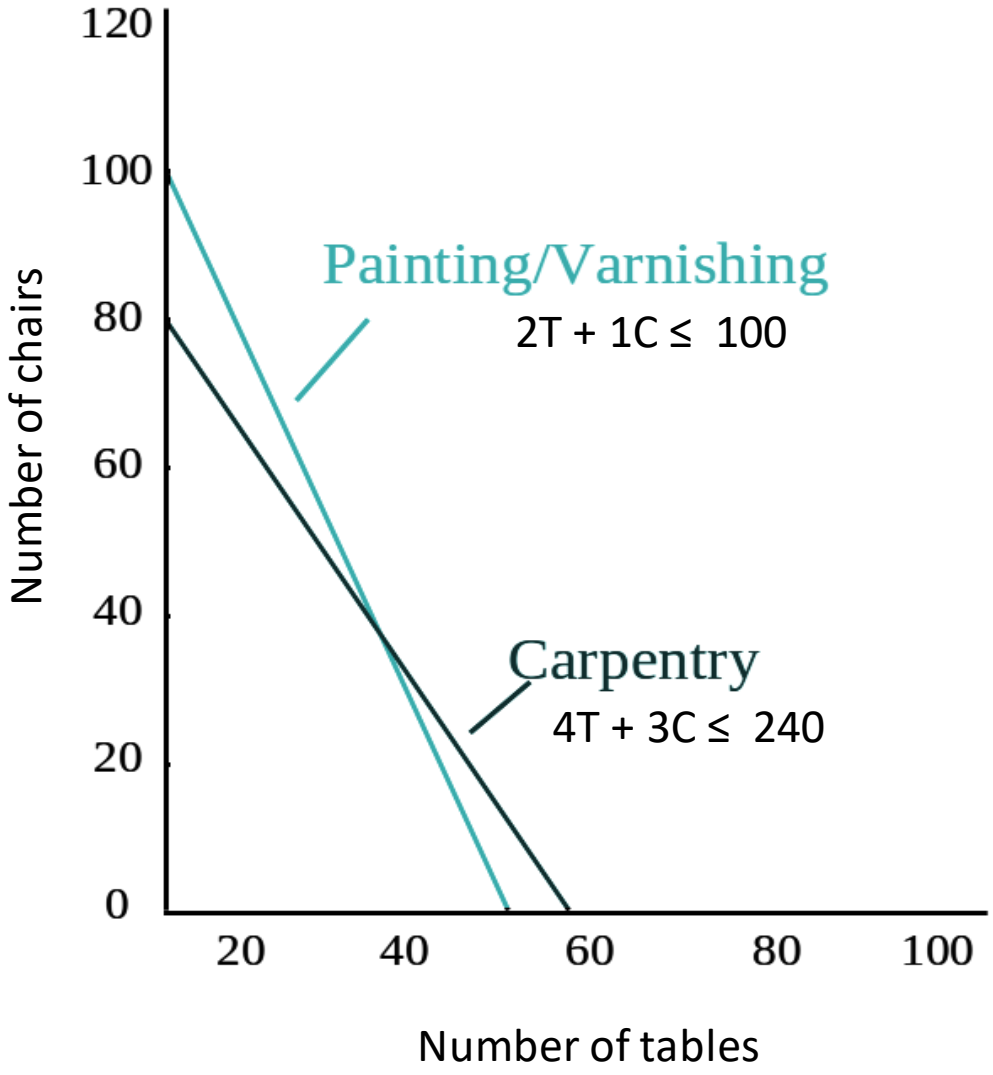
- Approach to solve LP problems with two decision variables
- It involves looking at the profit at every corner point of the feasible region
- The mathematical theory behind LP is that the optimal solution must lie at one of the corner points in the feasible region

Linear programming – graphical method

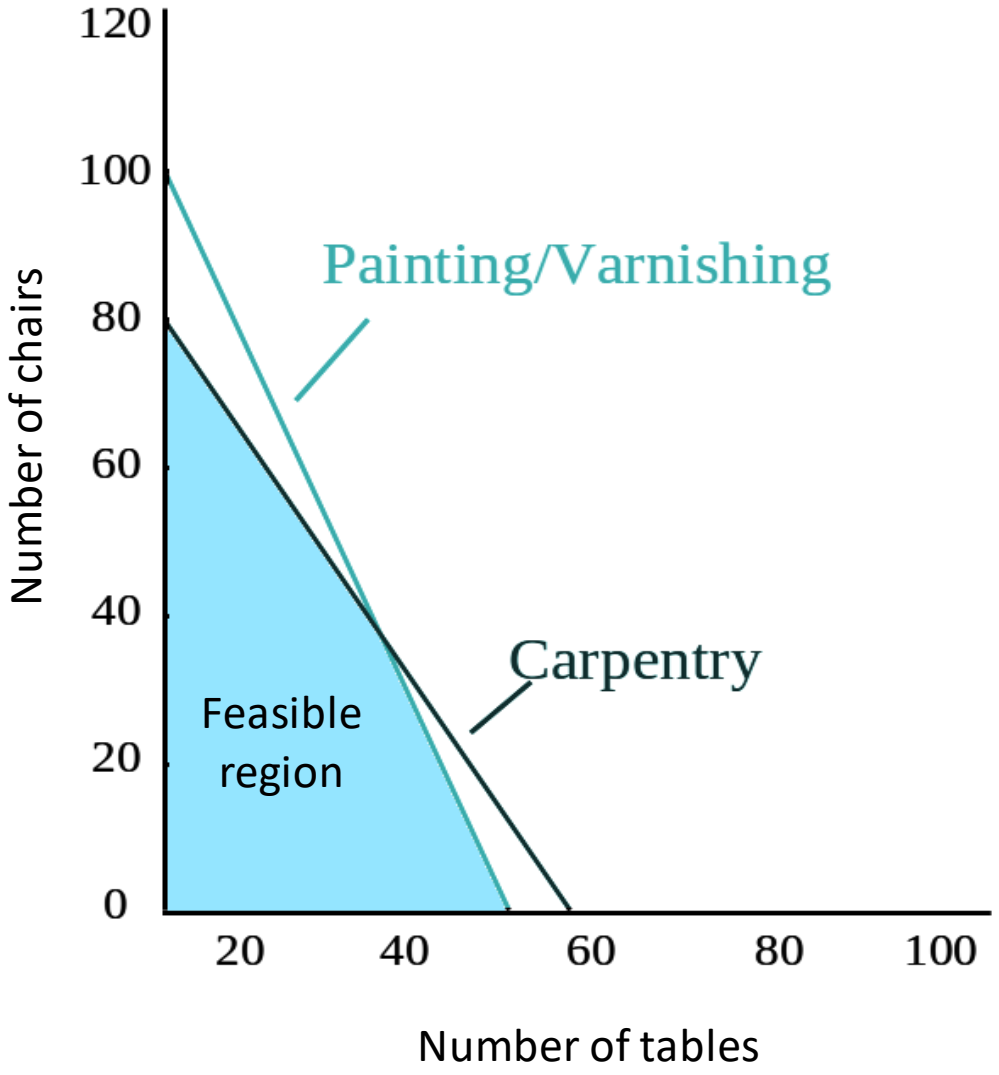
Steps in using the corner point method for solving LP problems

1. Graph all constraints and find the feasible region
2. Find the corner points of the feasible region
3. Compute the profit (or cost) at each of the feasible corner points
4. Select the corner point with the best value of the objective function found in step 3. This is the optimal solution

Linear programming – graphical method



Linear programming – graphical method



Linear programming – graphical method

- The feasible region for the problem is a four-sided polygon with four corner, or extreme, points.
- These points are labeled 1, 2, 3 and 4 on the next graph.
- To find the (T, C) values producing the maximum profit, find the coordinates of each corner point and test their profit levels.

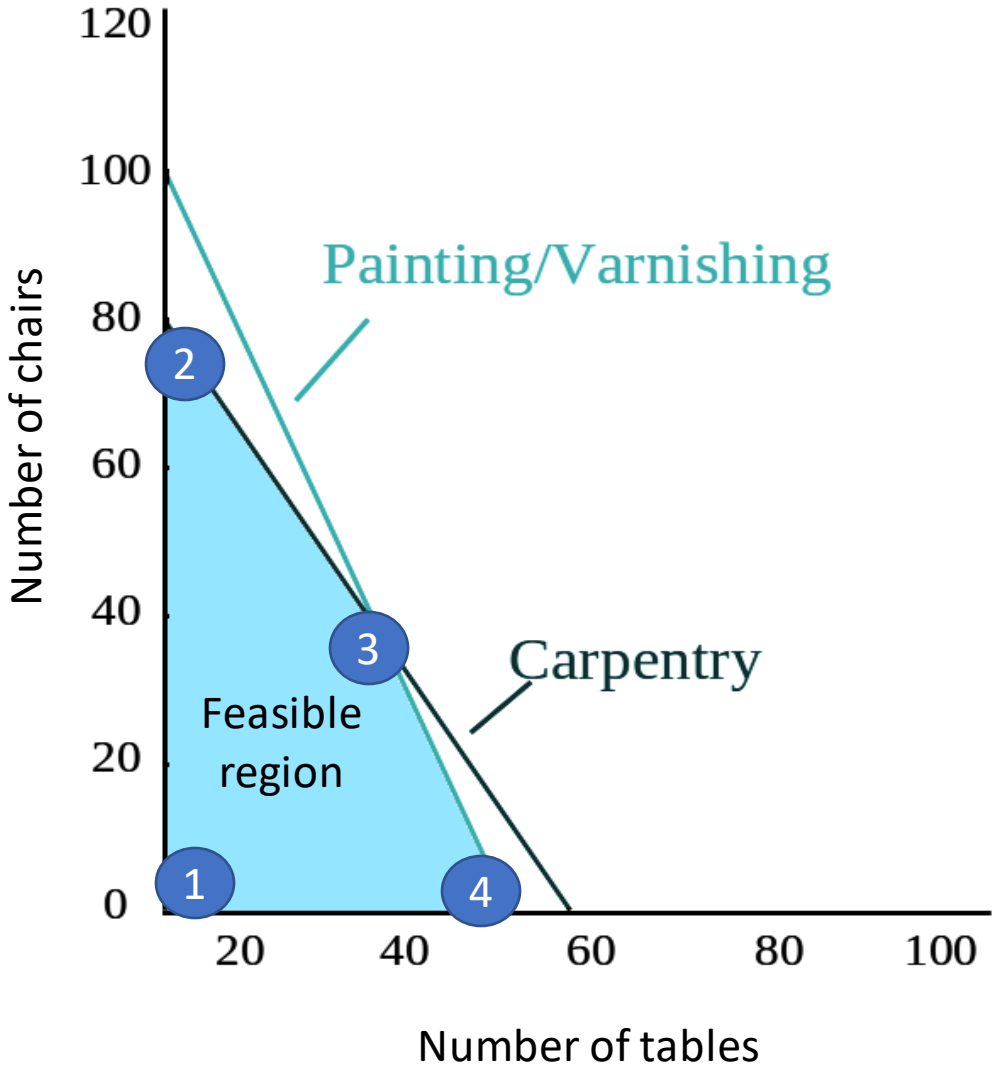
Point 1: (T = 0,C = 0) profit = $7(0) + $5(0) = $0

Point 2: (T = 0,C = 80) profit = $7(0) + $5(80) = $400

Point 3: (T = 30,C = 40) profit = $7(30) + $5(40) = **$410**

Point 4: (T = 50, C = 0) profit = $7(50) + $5(0) = $350

Linear programming – graphical method



Linear programming – Special cases

- Several special cases might occur when applying the graphical solution to a particular LP:
    1. Infeasibility
    2. Unbounded Solutions
    3. Multiple Optimal Solutions
    4. Redundancy in constraints

Integer programming

AALBORG UNIVERSITY
DENMARK

Integer programming

- Integer programming is the extension of LP that solves problems requiring integer solutions
- Many situations exists where decimal values of a particular decision variable do not give a feasible solution
- E.g., consider the optimal ordering policy problem: ordering 2.42 computers is not feasible
  - The number of computer should be a whole number (integer)

Integer programming – variants

There are three types of integer programs:

- **(Pure) Integer Programming:** all variables are required to have integer values
- **Mixed-Integer Programming (MIP):** some, but not all, of the decision variables are required to have integer values
- **0-1 Integer Programming:** a special case in which all the decision variables must have integer solution values of 0 or 1

Integer programming – solving IPs

- Rounding off is one way to reach integer solution values, but it often does not yield the best solution
- An important concept to understand is that an integer programming solution can never be better than the solution to the same LP problem
  - The integer problem is usually worse in terms of higher cost or lower profit

Integer programming – solving IPs

- Consider the following mixed product problem:

Maximise 7X1 + 6X2

subject to:

2X1 + 3X2 ≤ 12 (wiring hours)

6X1 + 5X2 ≤ 30 (assembly hours)

X1, X2 ≥ 0 (nonnegative)

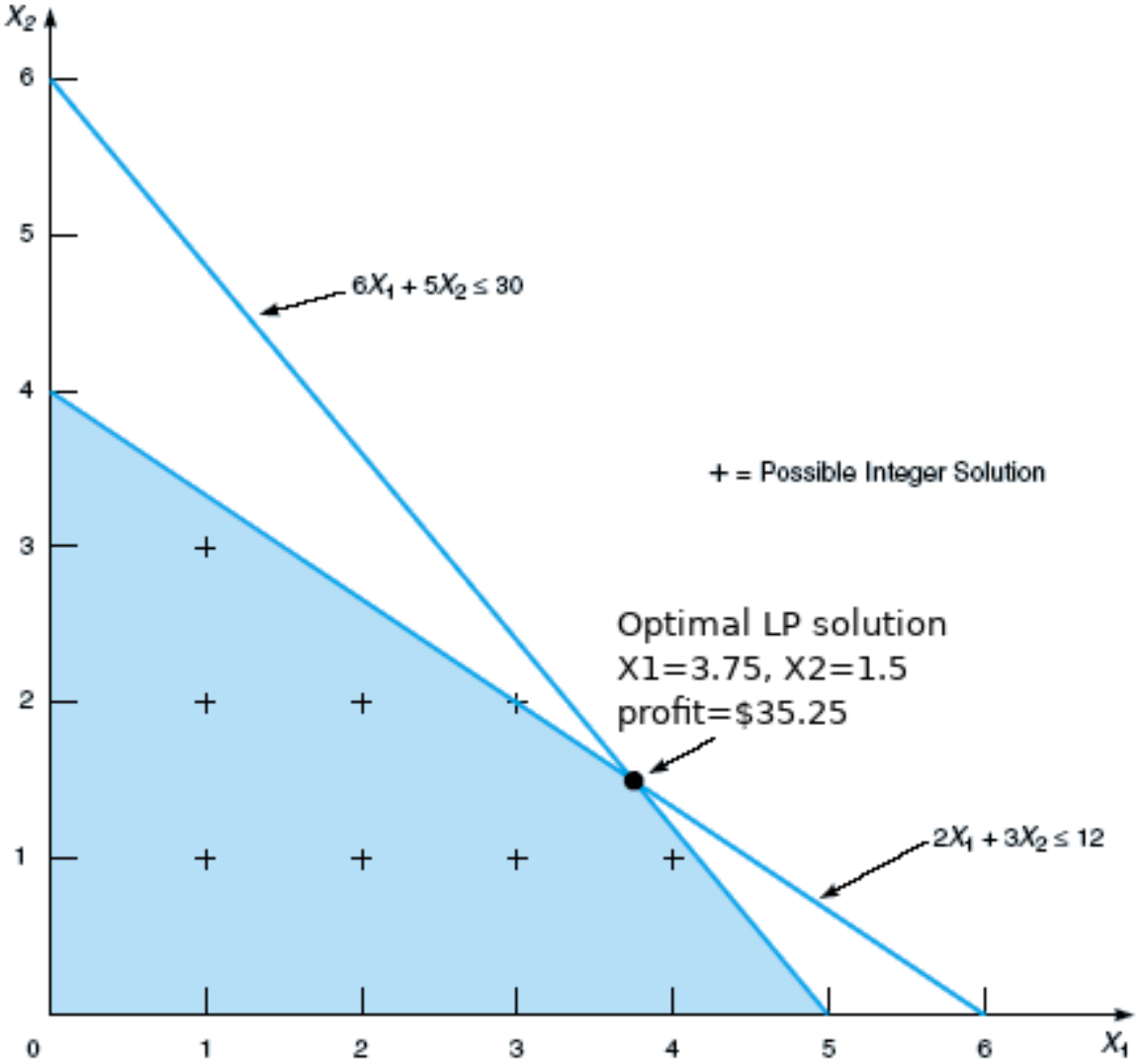
where:

X1 = number of chandeliers produced

X2 = number of ceiling fans produced

- Since we only have two decision variables, we can use the graphical approach to solve the LP
- This will also amply illustrate the main problem with rounding off as a method for solving Integer Programming problems

Integer programming – solving IPs



CHANDELIERS (X_1)	CEILING FANS (X_2)	PROFIT ($\$7X_1 + \$6X_2$)	
0	0	$0	
1	0	7	
2	0	14	
3	0	21	
4	0	28	
5	0	35	← True optimal integer solution
0	1	6	
1	1	13	
2	1	20	
3	1	27	
4	1	34	← Solution if rounding off is used
0	2	12	
1	2	19	
2	2	26	
3	2	33	
0	3	18	
1	3	25	
0	4	24	

Integer programming – solving IPs

- Conclusion: rounding off does not guarantee obtaining the optimal solution
- Instead of rounding off, we introduce the Branch and Bound method
- The Branch and Bound method breaks the feasible solution region into sub-problems until an optimal solution is found
- There are Six Steps in Solving Integer Programming Maximisation Problems by Branch and Bound

Integer programming – solving IPs – B&B

Branch and bound steps for a maximisation problem

1. Solve the original problem using LP:
   - If the answer satisfies the integer constraints, it is the optimal solution
   - If not, this value provides an initial upper bound
2. Find any feasible integer solution that meets the integer constraints for use as a lower bound
   - Usually, rounding down each variable will accomplish this
3. Branch on one variable from Step 1 that does not have an integer value.
   - Split the problem into two sub-problems based on integer values that are immediately above and below the non-integer value.
   - For example, if $X_1 = 3.75$ was in the final LP solution, introduce the constraint $X_1 \geq 4$ in the first sub-problem and $X_1 \leq 3$ in the second sub-problem

Integer programming – solving IPs – B&B

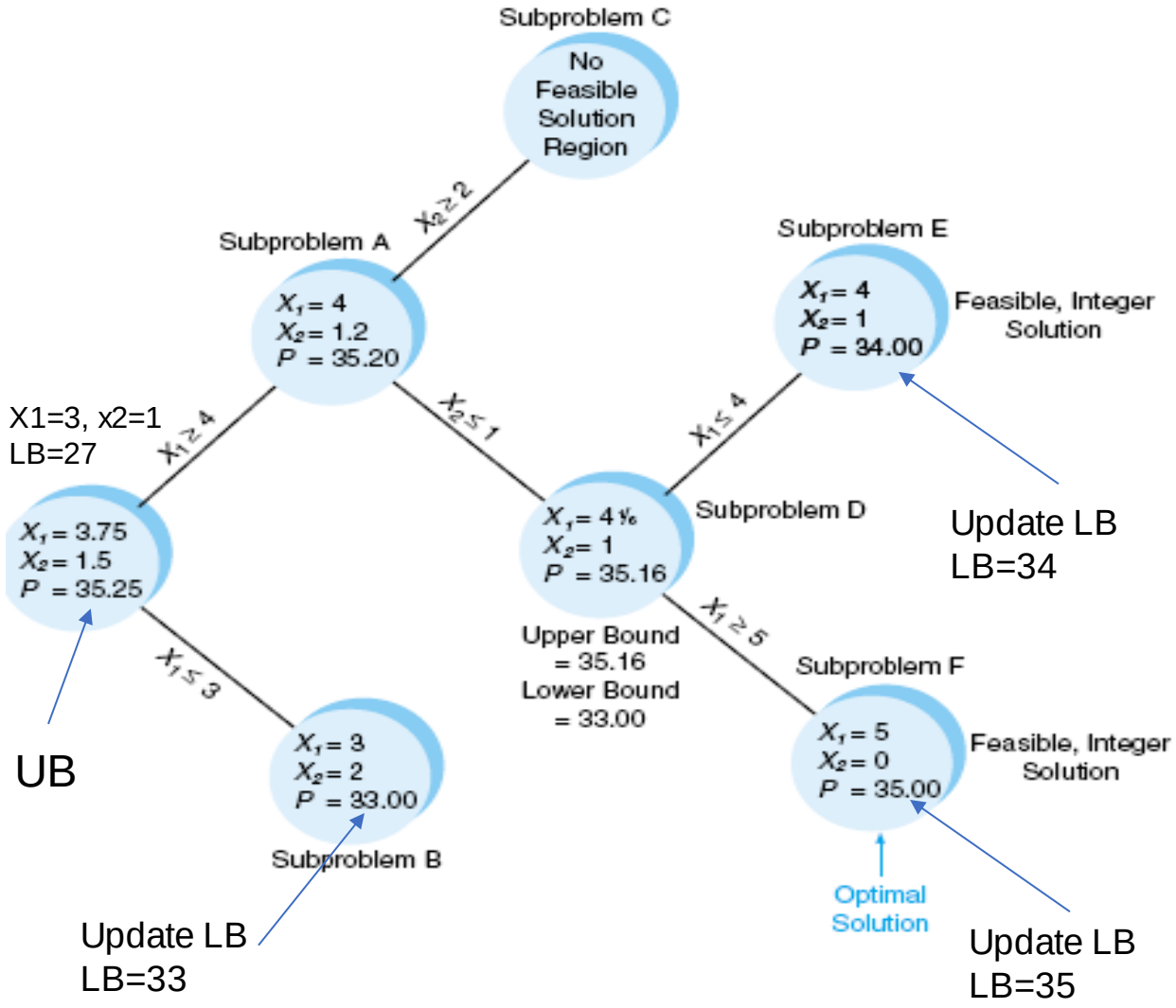
4. Create nodes at the top of these new branches by solving the new problems.
5. Branch termination
   - If a branch yields a solution to the LP problem that is not feasible, terminate the branch.
   - If a branch yields a solution to the LP problem that is feasible, but not an integer solution, go to step 6.
   - If the branch yields a feasible integer solution, examine the value of the objective function. If this value equals the upper bound, an optimal solution has been reached.
   - If it is not equal to the upper bound, but exceeds the lower bound, set it as the new lower bound and go to step 6.
   - Finally, if it is less than the lower bound, terminate this branch.

Integer programming – solving IPs – B&B

6. Examine both branches again and set the upper bound equal to the maximum value of the objective function at all final nodes.
   - If the upper bound equals the lower bound, stop.
   - If not, go back to step 3.

If we are solving a minimisation problem, simply reverse the roles of upper and lower bound. Alternatively, convert the minimisation problem into a maximisation problem.

Integer programming – solving IPs – B&B



Maximise $7X_1 + 6X_2$

subject to:

$2X_1 + 3X_2 \leq 12$ (wiring hours)

$6X_1 + 5X_2 \leq 30$ (assembly hours)

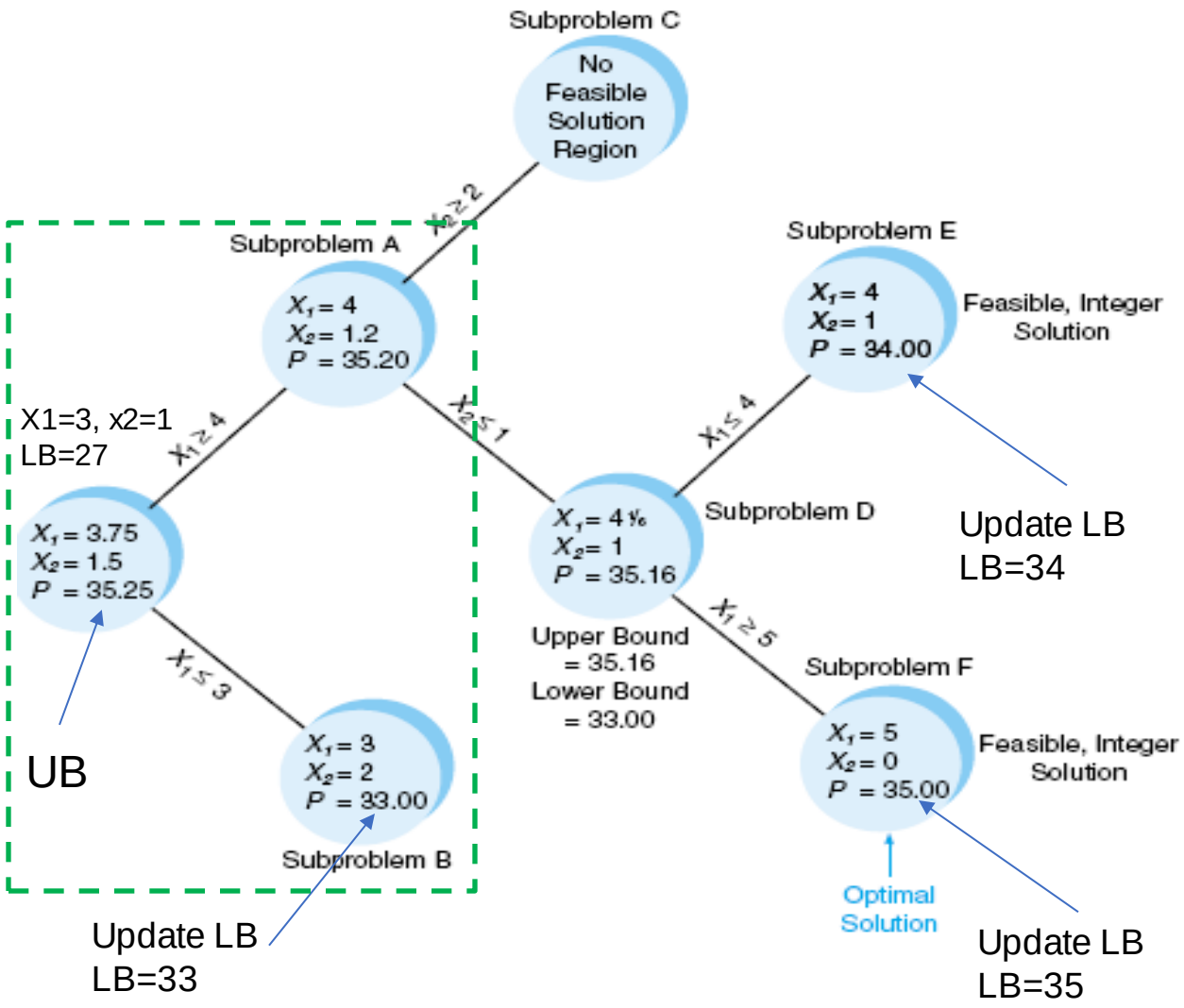
$X_1, X_2 \geq 0$ (nonnegative)

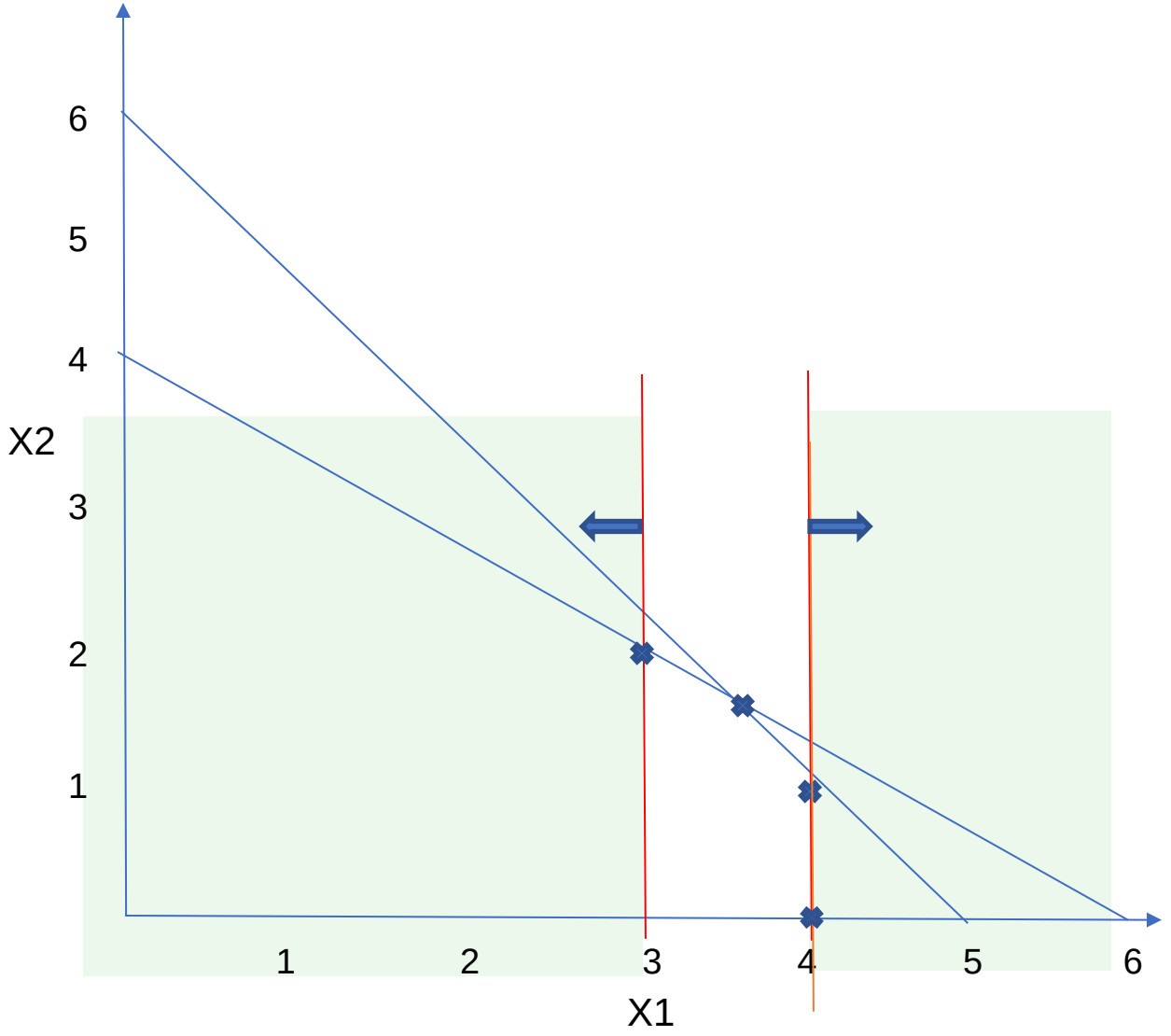
where:

X_1 = number of chandeliers produced

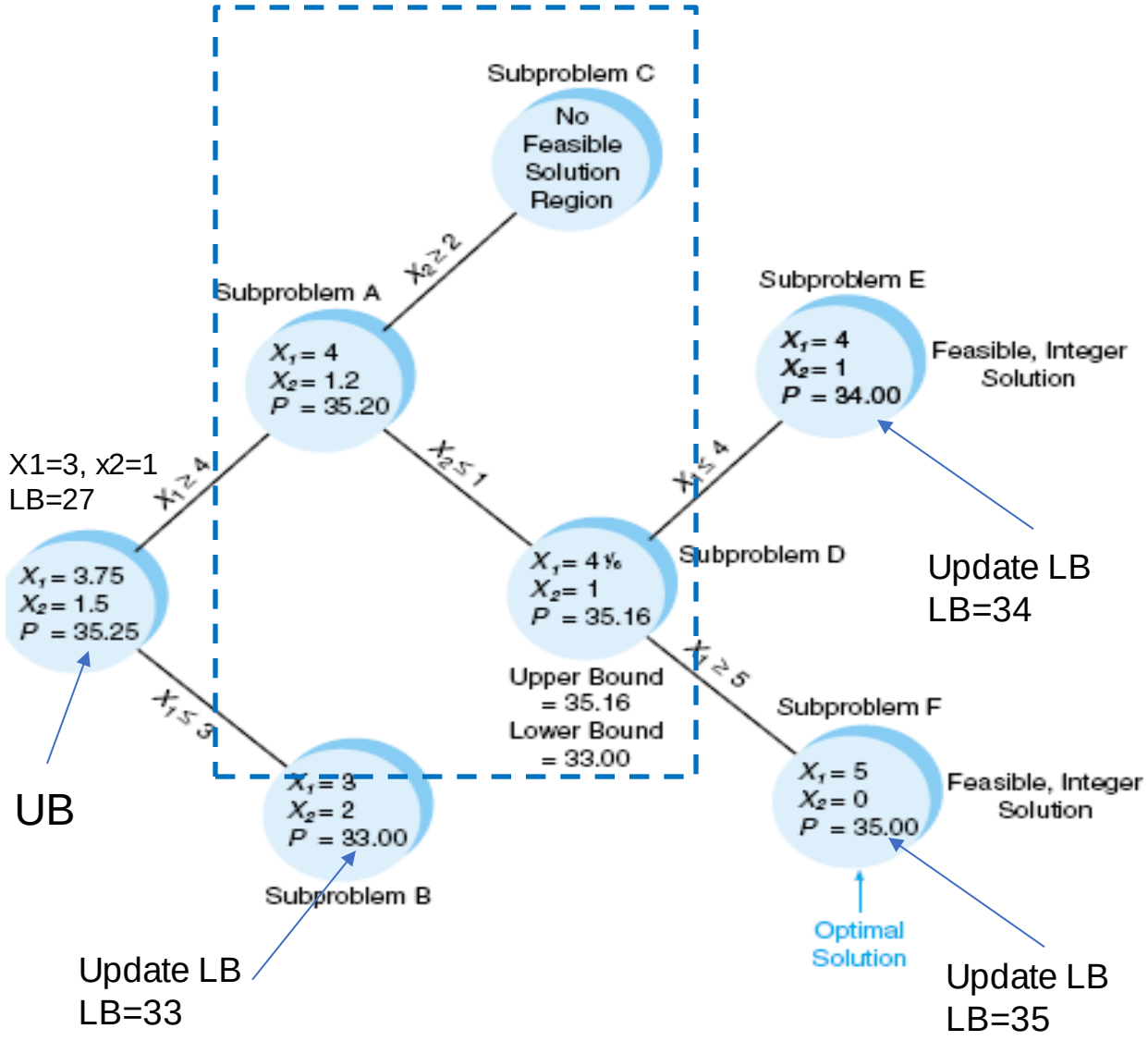
X_2 = number of ceiling fans produced

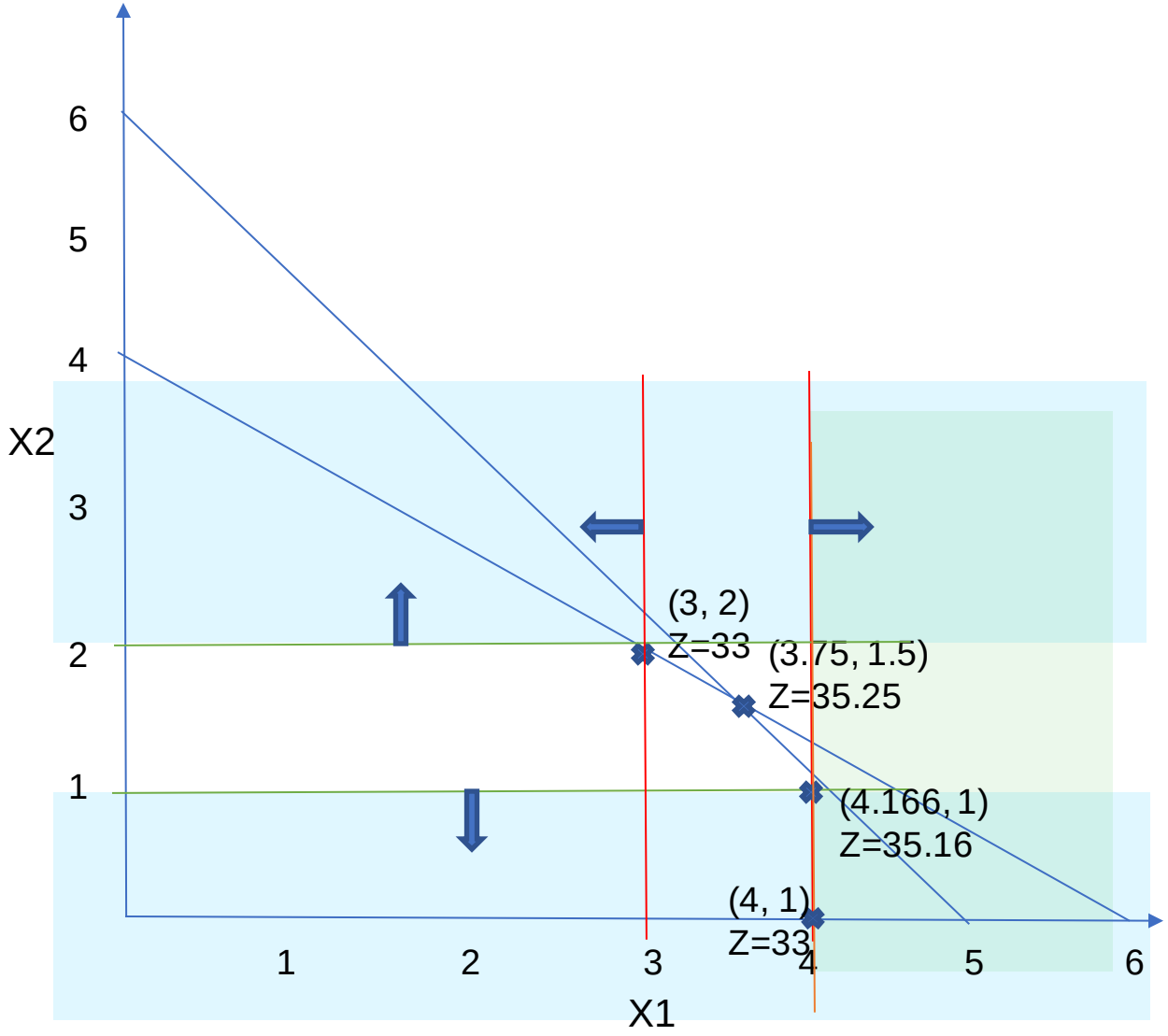
Integer programming – solving IPs – B&B



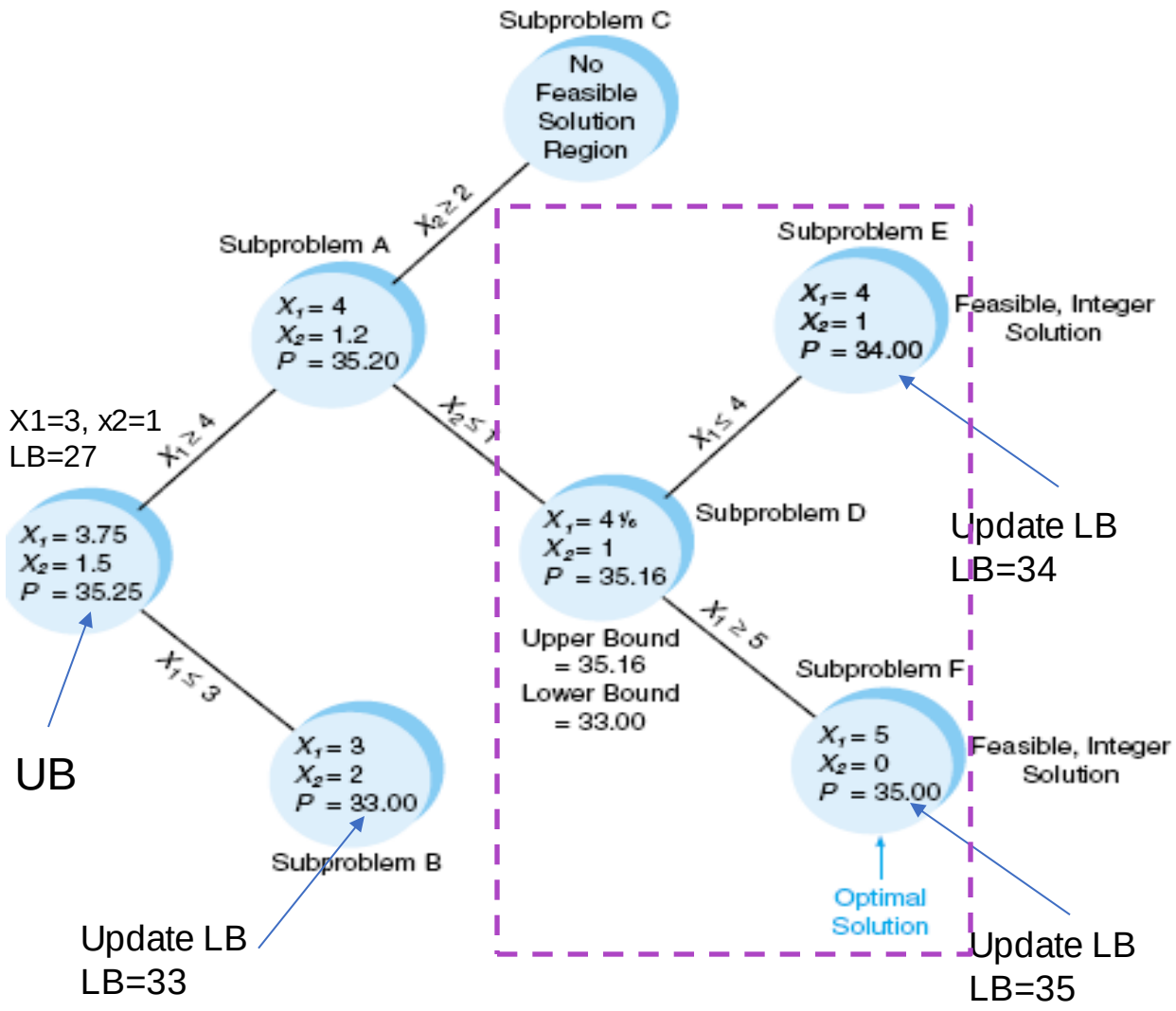


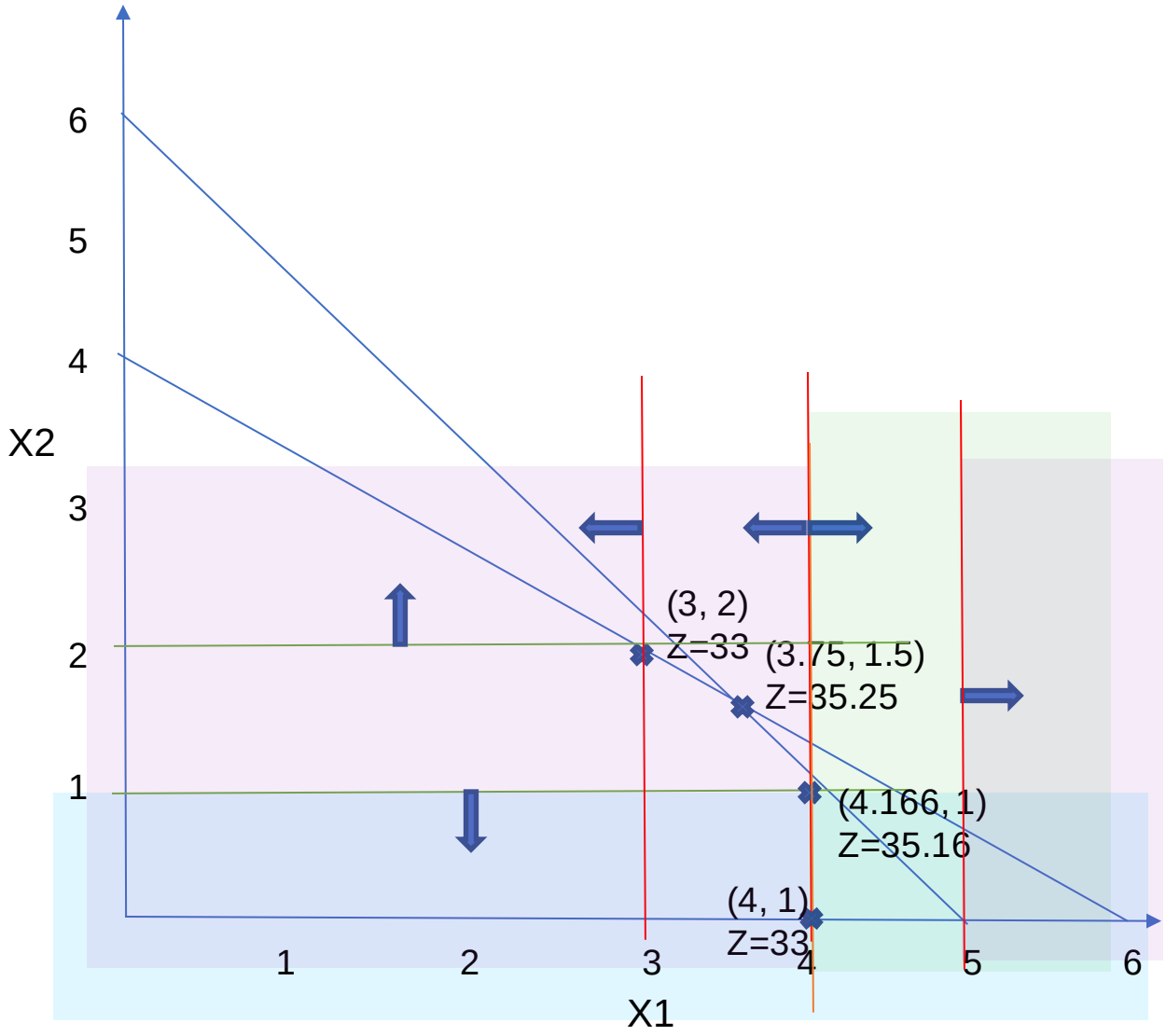
Integer programming – solving IPs – B&B





Integer programming – solving IPs – B&B





Combinatorial optimisation and metaheuristics

AALBORG UNIVERSITY
DENMARK

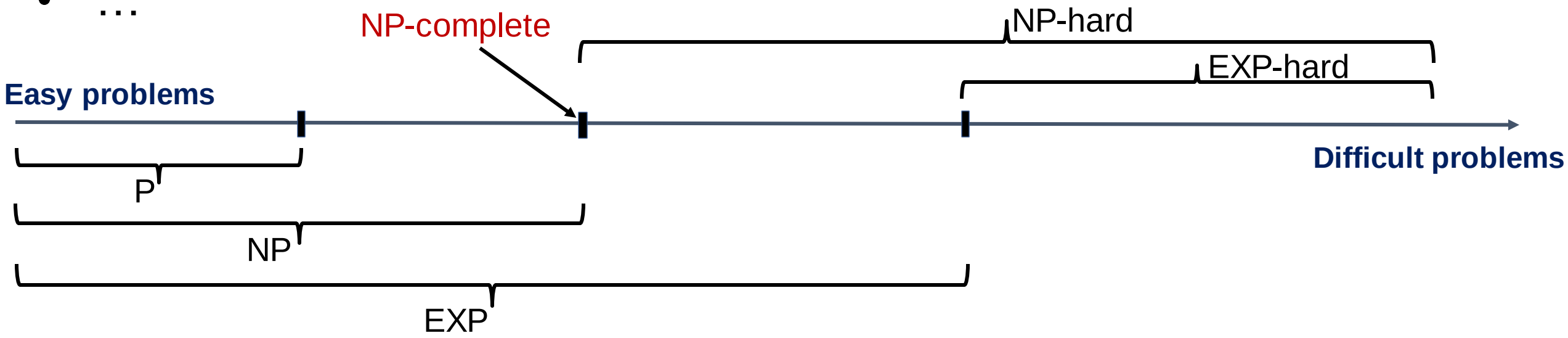
Combinatorial optimisation

- Combinatorial optimisation is a topic that consists of studying optimisation problems with a finite set of solutions
- Many important problem in OR are combinatorial optimisation problems:
  - Scheduling problems
  - Transportation problems
  - Path planning
  - Etc.

Problem complexity classes – how "difficult" is your problem?

In computational complexity, there are many problem complexity classes:

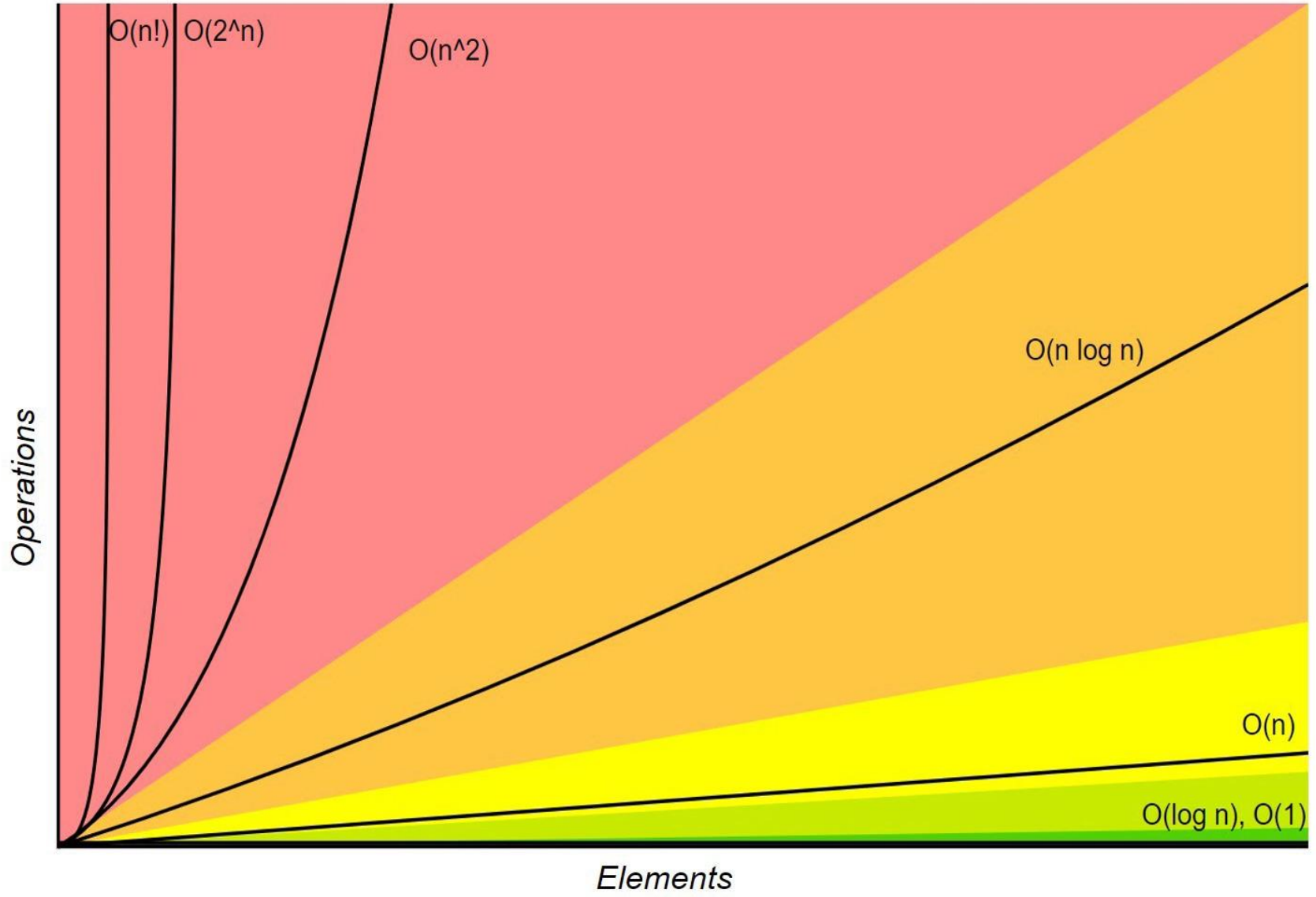
- P : the set of problems solvable in polynomial time
- EXP : the set of problems solvable in exponential time
- NP : the set of problems with solutions verifiable in polynomial time
- NP-hard : the set of problems that are "at least as hard as the hardest problem in NP".
- NP-complete = NP ∩ NP-hard
- …



Problem complexity classes

- NP-complete problems are not solvable in polynomial time, unless P=NP.
  - "P=NP**?**" is to this day an open question in mathematics and CS
  - Most researchers "believe" that P≠NP
- Many logistics and managerial problems are either in P or NP-complete.
- Integer programming is NP-complete, while Pure Linear programming is in P.
  - This is the fundamental reason why solving ILP problems is more challenging than solving pure LP problems.

Complexity chart – how "fast" is your algorithm?



The knapsack problem

We are given:

- A set of *n* items numbered from 1 to *n*, where each item *i* has a weight w_i and a value v_i.
- A knapsack with a maximum weight capacity *W.*

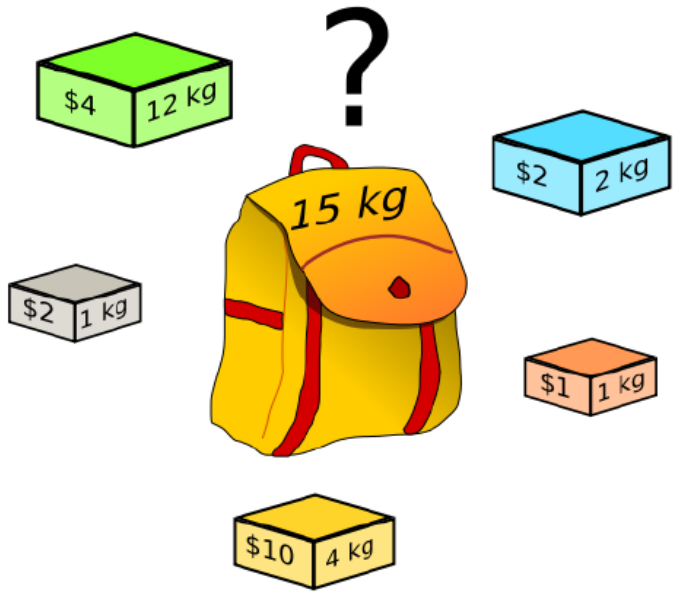
The goal is to select some items in order to maximise the total value, while satisfying the maximum capacity constraint:

Maximise $$\sum_{i=1}^{n} v_i x_i$$

Subject to: $$\sum_{i=1}^{n} w_i x_i \leq W$$

$$x_i \in \{0, 1\}$$

1: packed
0: not packed



The knapsack problem

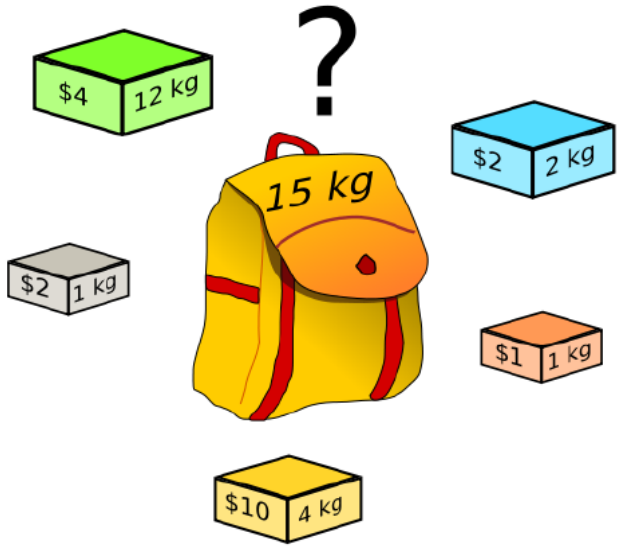
- The knapsack problem in a classic NP-complete problem
- It is one of the most intuitive problems
- Many exact approaches can be used to solve the problem (find an optimal solution):
  - Dynamic programming
  - Branch and bound
  - …
- However, since the problem is NP-complete, the running time of these approaches increases exponentially w.r.t. the size of the problem!

Alternative solution approaches

- Instead of using exact approaches, we will look at alternative approaches able to find feasible and "satisfactory" solutions, with no guarantee of optimality.
- These approaches can be categorised as follows:
  - **Greedy/constructive algorithms**
    - Construct a solution from scratch while ensuring feasibility
  - **Local search algorithms**
    - Start from an arbitrary solution, then try to improve it iteratively by generating "neighbouring" solutions using small changes (called moves)
  - **Metaheuristics**
    - High-level heuristics approaches that make few or no assumptions about the problem
    - Try to improve a given solution iteratively by sampling new solutions

Greedy/constructive algorithms

- A greedy algorithm is any algorithm that tries to optimise a given problem by making locally optimal choices
- Greedy algorithms are fast, but do not guarantee optimality nor good quality solutions
- Example for the knapsack problem:
  1. Sort the items from the most valuable to the less valuable ones
  2. Select the items to put in the knapsack in that order
  3. Skip the items that exceed the total capacity
  4. Stop when we don't have any more items to evaluate, or maximum capacity is reached



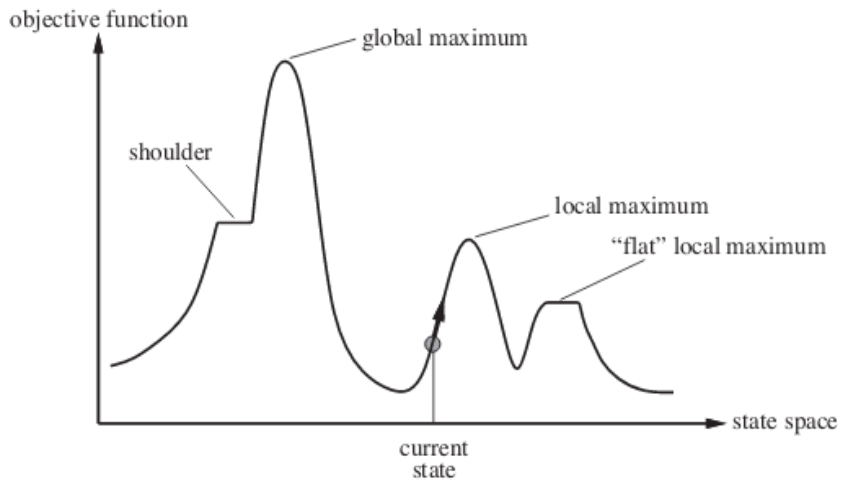
Hill climbing/neighbourhood search

- Hill climbing is a technique belonging to the family of local search algorithm.
- It is an iterative algorithm that starts with an arbitrary solution, then attempts to find a better solution by making an incremental change to the solution.
- If the change produces a better solution, another incremental change is made to the new solution, and so on.
- The algorithm stops when no further improvements can be found.

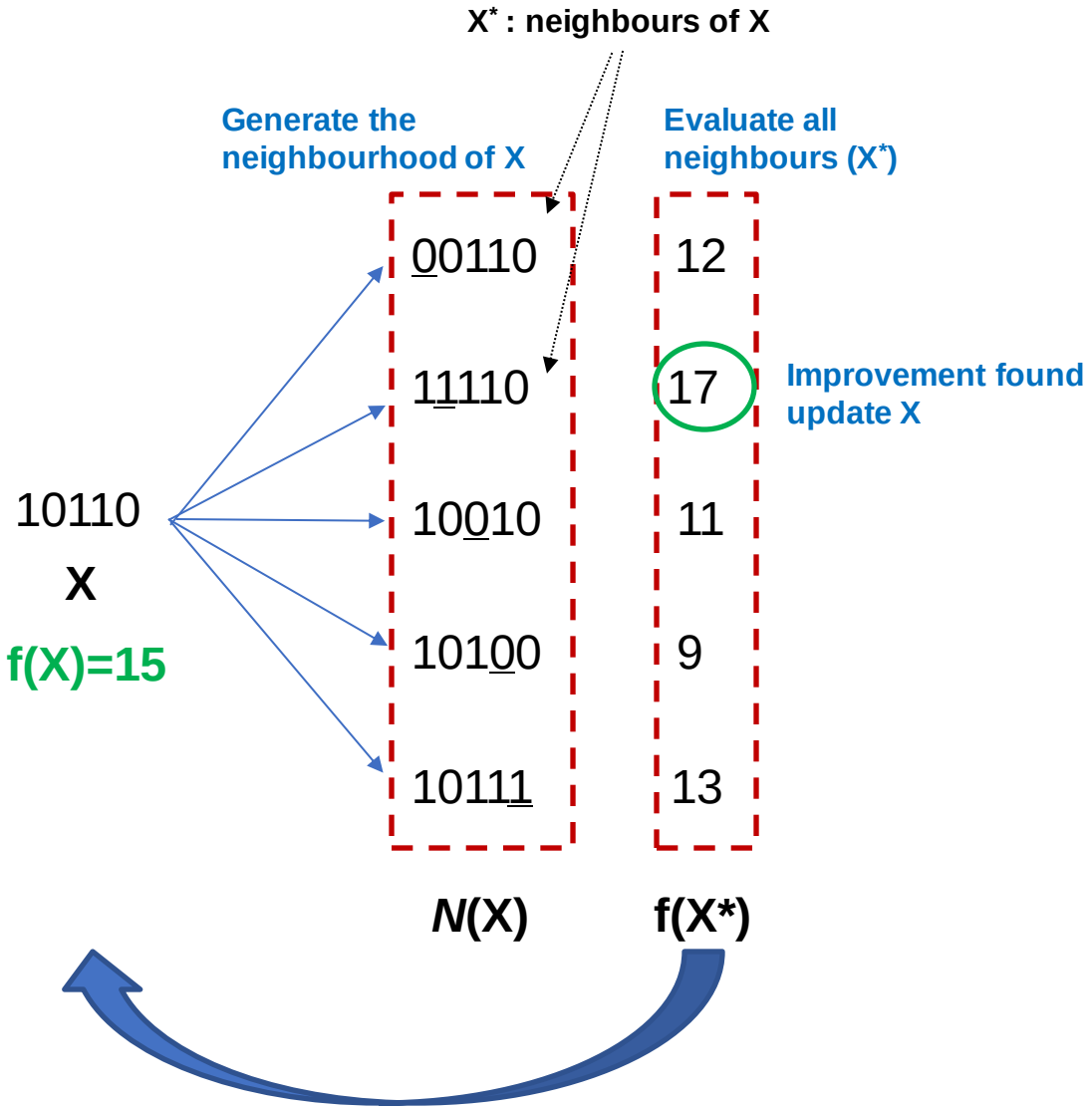
Hill climbing/neighbourhood search

- The neighbourhood *N(X)* is generated by applying a small change on the current solution
- Example for KP:
  - We can use the bit-flip operator:
    If the DV is 0, flip to 1
    Flip to 1 otherwise
  - One issue with this approach: improvement will occur only when flipping 0 to 1
  - Alternative operators: swap operator (swap two DVs, i.e., pack one and unpack the other)
- Given a solution X, A neighbouring solution X' is usually faster to evaluate : $f(X') = f(X) \pm \Delta$

1: $X \leftarrow$ initial solution
2: **repeat**
3: **for** $X^* \in \mathcal{N}(X)$ **do**
4: **if** $f(X^*) > f(X)$ **then**
5: $X \leftarrow X^*$
6: **end if**
7: **end for**
8: **until** there is no improvement



Hill climbing/neighbourhood search



1: $X \leftarrow$ initial solution
2: **repeat**
3: **for** $X^* \in \mathcal{N}(X)$ **do**
4: **if** $f(X^*) > f(X)$ **then**
5: $X \leftarrow X^*$
6: **end if**
7: **end for**
8: **until** there is no improvement

Stochastic hill climbing

- In stochastic hill climbing, instead of generating the entire neighbourhood, we generate only one neighbouring solution at random.
- If the neighbour is better than the current solution, we update the current best solution. Otherwise, we repeat the process.
- This process is repeated for a specified maximum number of iterations.

```
1: X ← initial solution
2: i ← 0
3: for i < max_iterations do
4:     X* ← random_neighbour(X)
5:     if f(X*) > f(X) then
6:         X ← X*
7:     end if
8:     i ← i + 1
9: end for
```

Heuristic design rules – the transition to metaheuristics

- To design an efficient heuristic, it is important to ensure a tradeoff between exploitation and exploration:
  - Exploitation consists of focusing the search in a particular area. This is what classical local search algorithms do.
  - Exploration consists of diversifying the search and exploring multiple areas to improve the solution further. For instance, ILS does this with perturbations.
- An algorithm that focuses more on exploitation will converge towards local search; and an algorithm that focuses more on exploration will converge towards random search.
- Incorporate problem knowledge whenever possible.

Extending local search

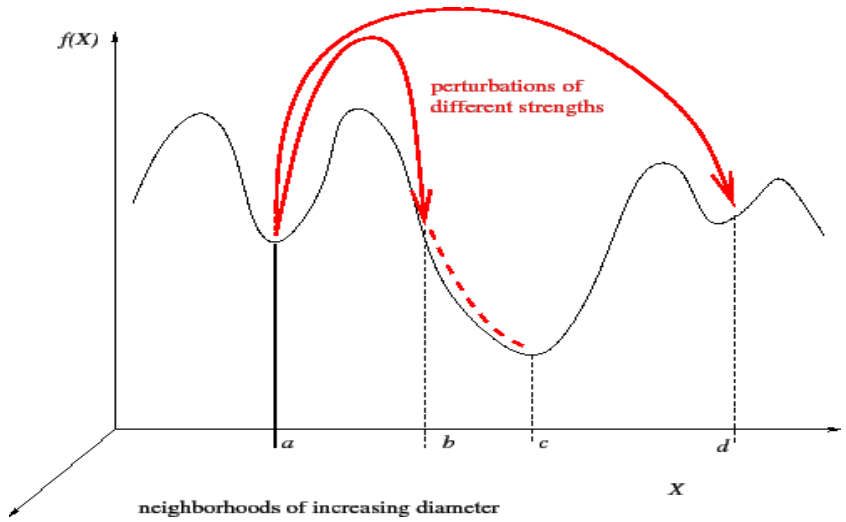
There are many metaheuristics that attempt to extend local search by incorporating exploration mechanisms:

- **Tabu search** accepts worst solutions using a tabu list to record already visited solutions
- **Variables neighbourhood search** uses multiple neighbourhood operators to escape from local optima
- **Iterated local search** tries to escape from local optima by applying stronger moves (perturbations)
- **Simulated annealing** accepts worst solutions using a probability based on Boltzmann criterion
- …

Extending local search - Iterated Local Search

- The issue with most local search algorithms is that they get stuck in local optima
- Iterated local search tries to solve this problem by applying "perturbations" to the current solution in order to escape from the local optima
- The perturbation must be strong enough to explore another local optimum area, but not too strong (random restart)

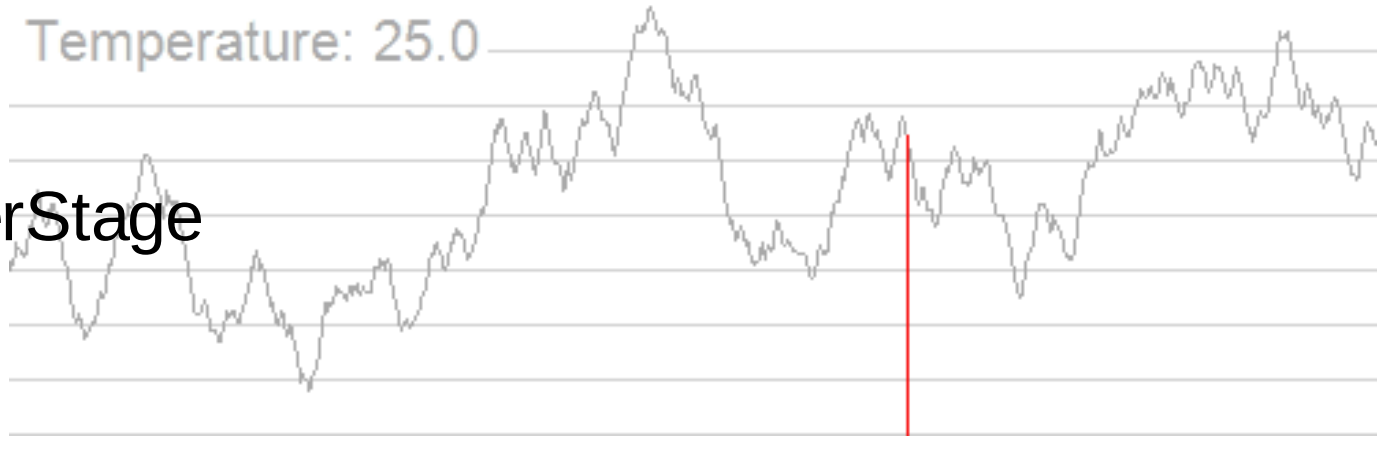
procedure *Iterated Local Search*
  $s_0 = \text{GenerateInitialSolution}$
  $s^* = \text{LocalSearch}(s_0)$ % optional
  repeat
    $s' = \text{Perturbation}(s^*)$
    $s^{*\prime} = \text{LocalSearch}(s')$
    $s^* = \text{AcceptanceCriterion}(s^*, s^{*\prime})$
  until termination condition met
end



Extending local search - Simulated annealing

- **Simulated annealing** is a stochastic optimisation algorithm attempting to "balance" between exploitation and exploration by accepting non-improving solution with a decreasing probability (slow cooling)
- It is an extension/adaptation of the Metropolis-Hastings algorithm for sampling
- Parameters to tune:
  - Initial temperature: T_{init}
  - Cooling rate: α
  - Number of inner iterations: IterStage

procedure *Simulated_Annealing(inital solution s)*
  $Best \leftarrow s$
  $T \leftarrow T_{init}$, choose $\alpha \in]0,1[$, choose $IterStage$
  repeat
    for $i = 1$ **to** $IterStage$
      Choose $s' \in N(s)$
      $\Delta \leftarrow f(s') - f(s)$
      $s \leftarrow s'$ { if $\Delta < 0$ or with the probability $\exp \frac{-\Delta}{T}$ } Update $Best$
    end for
    $T \leftarrow \alpha \times T$
  until stopping conditions are met
end

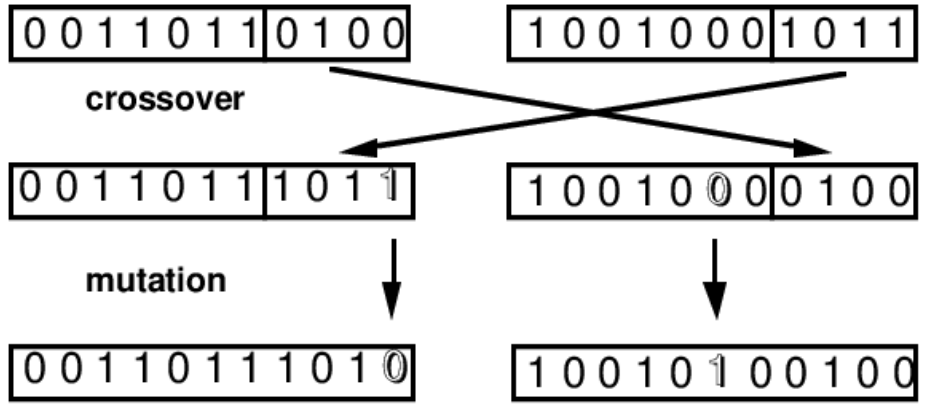


Source: https://upload.wikimedia.org/wikipedia/commons/d/d5/Hill_Climbing_with_Simulated_Annealing.gif

Beyond local search

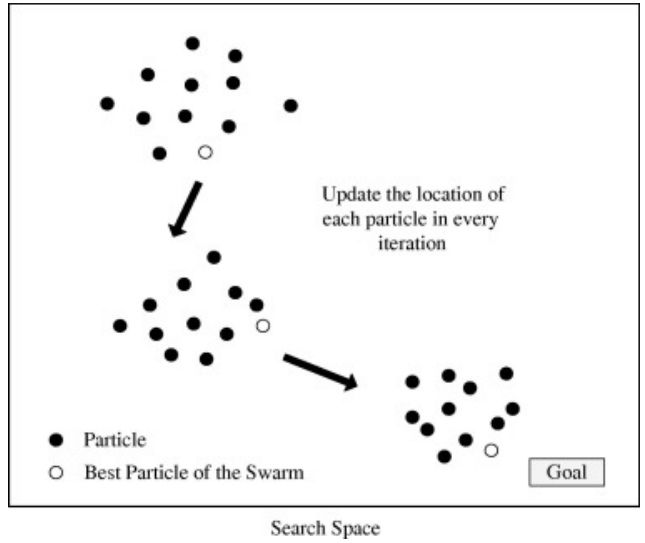
There are many metaheuristics that are not extensions of local search algorithms

- **Genetic algorithms** adopt the idea of neo-Darwinian evolution to iteratively "evolve" a population of solutions using genetic operators:
  - Crossover: generating a solution by combining two "parent" solutions
  - Mutation: applying small changes based on a small probability
  - (Artificial) Selection: tournament selection, random selection, roulette wheel selection, etc.

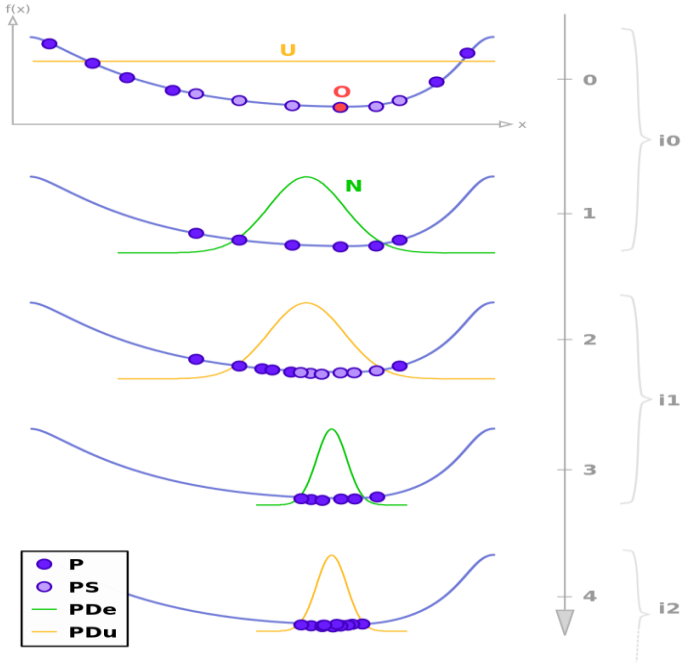


Beyond local search

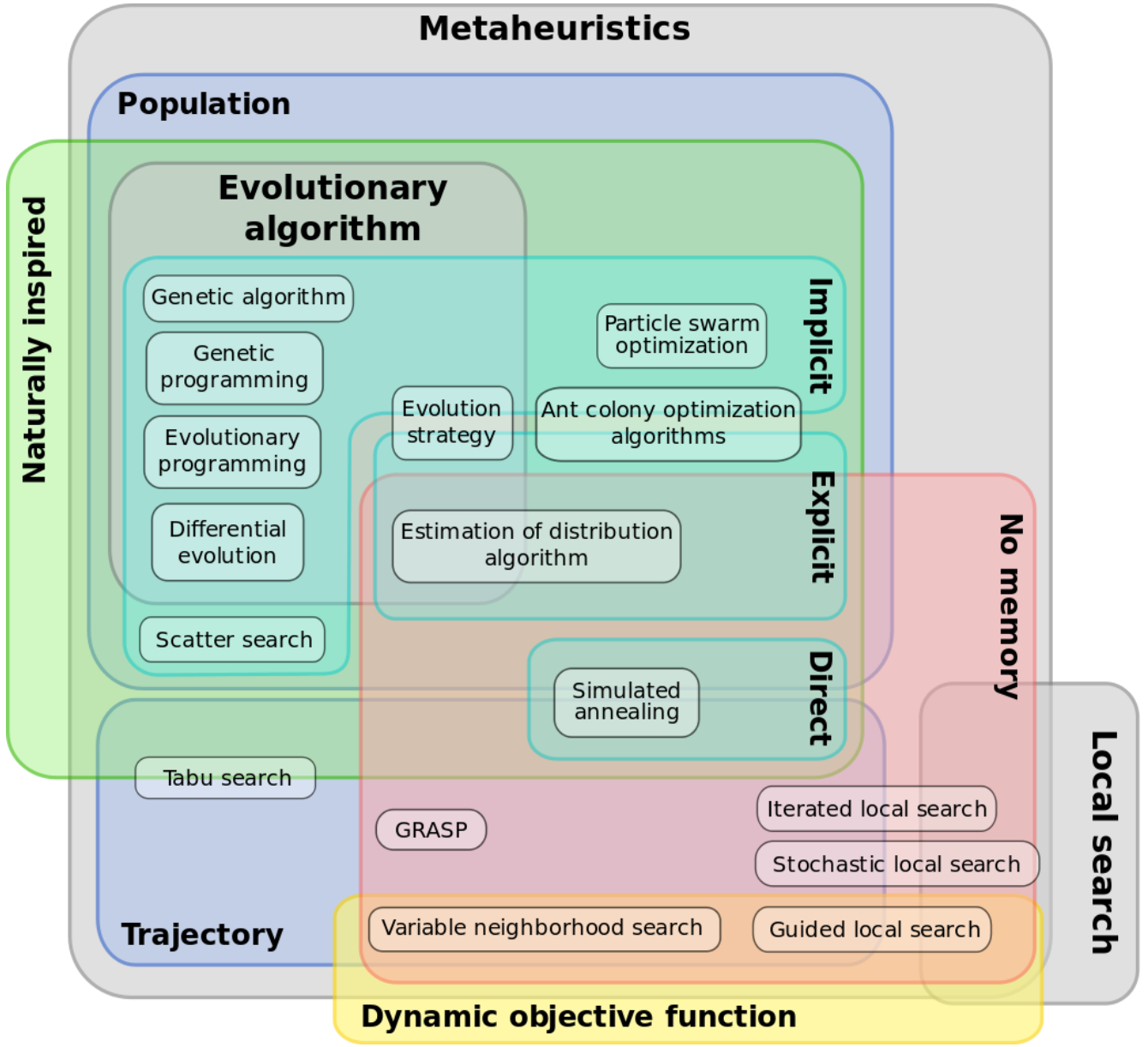
- **Particle swarm optimisation** uses principles from collective behaviours of decentralised systems to iteratively improve a population of solutions.



- **Estimation of Distribution Algorithms** are methods that guide the search for the optimum by building and sampling explicit probabilistic models of promising candidate solutions.



The bigger picture...

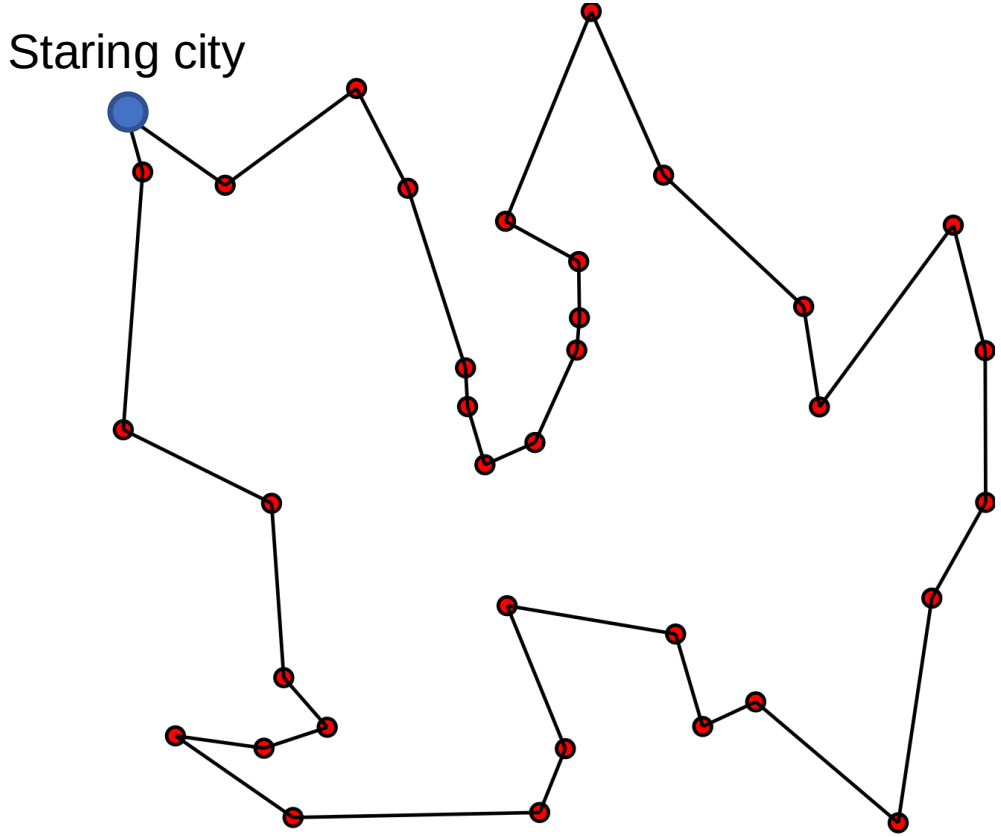


Source: https://en.wikipedia.org/wiki/Metaheuristic#/media/File:Metaheuristics_classification.svg

Quick exercise – The travelling salesman problem

Let us now consider the Travelling Salesman Problem (TSP):

- Given a set of N cities, a salesman must visit each city exactly once before going back to the starting city
- Let $\{d_{ij}\}$ be the distance matrix between cities i and j
- **Goal:** find a tour that minimises the total travelling distance



Assuming solutions are encoded as permutations (sequence in which the cities will be visited):
- how to design a constructive/greedy algorithm for TSP?
- how to design an operator to generate a neighbourhood?

Before using a metaheuristic

- **No guarantee of optimality -** how important is optimality for your problem?
- **No free lunch -** the choice of the best metaheuristic for your problem is not an easy one
- **Parameters -** depending on which metaheuristic you opt for, there might be multiple parameter to tune...
  - You could use a metaheuristic to tune a metaheuristic, that tunes a metaheuristic...
  - Heuristic parameter tuning in itself is an active area of research - check out:
    - **The irace package:** Iterated racing for automatic algorithm configuration
    - **SMAC:** Learning the empirical hardness of optimization problems: The case of combinatorial auctions
    - **MATE:** A Model-based Algorithm Tuning Engine

Additional resources

- Zbigniew Michalewicz and David B. Fogel. "*How to solve it: modern heuristics"*. Springer Science & Business Media, 2013.

- El-Ghazali Talbi. "*Metaheuristics: from design to implementation"*. Vol. 74. John Wiley & Sons, 2009.