NATCOR Course 'Heuristic Optimisation and Learning 2024' at University of Nottingham

Preliminary Reading Materials

This document includes supplementary materials recommended by some of the instructors, as well as the preliminary reading provided in the following pages.

Please note that all preliminary reading materials and course teaching materials for each session will be available in the NATCOR UoN 2024 team once it is set up in Microsoft Teams. Access to materials may vary; for some sessions, they will be available in advance, while for others, they will be available until the session starts.

Session 'Complexity Theory Part 1'

- **Reading 1:** *Ian Stewart on Minesweeper* [PDF]

Session 'Design of Heuristic Algorithms'

- **Reading 2:** *An Overview of Heuristic Solution Methods.* Edward Allen Silver. *Journal of the Operational Research Society, 55,* 936-956, 2004. [PDF]

Session 'Automated Configuration of Optimisation Algorithms'

These are not mandatory reading, but they could be useful if you wish to get a preliminary idea of topics that we will cover.

- **Reading 3:** *The irace package: Iterated Racing for Automatic Algorithm Configuration.* Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. Operations Research Perspectives, 3:43–58, 2016. [PDF]
- **Reading 4:** *Automatically Improving the Anytime Behaviour of Optimisation Algorithms.* Manuel López-Ibáñez and Thomas Stützle. European Journal of Operational Research, 235(3):569–582, 2014. [PDF]
- **Reading 5:** *From Grammars to Parameters: Automatic Iterated Greedy Design for the Permutation Flow-shop Problem with Weighted Tardiness.* Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle. In P. Pardalos and G. Nicosia, editors, Learning and Intelligent Optimization, 7th International Conference, LION 7, volume 7997 of Lecture Notes in Computer Science, pages 321–334. Springer, Heidelberg, Germany, 2013. [PDF]

In addition, it would be useful to install the **irace** and **iraceplot** packages. See the instructions available here: `https://lopez-ibanez.eu/2024-natcor-aac/#setup`

Session 'Big Data and Machine Learning'

There will be some demos using Jupyter Notebooks and pyspark in this session. It will be optional for students to replicate these demos. In that case, they might want to install these tools beforehand. There are lots of guides online, but these two seem good starting points:

- Guide to install spark and use pyspark from jupyter in windows (Windows)
- Get Started with PySpark Jupyter Notebook in 3 Minutes(Unix)

Session 'Approaches for Real World Air Transportation Problems'

- **Reading 6:** *Pruning Rules for Optimal Runway Sequencing.* Geert De Maere, Jason A.D. Atkin, Edmund K. Burke. *Transportation Science, 52(4),* 739-1034, 2018. [PDF]
- **Reading 7:** *A Multi-objective Approach for Robust Airline Scheduling.* Edmund K. Burke, Patrick De Causmaecker, Geert De Maere, Jeroen Mulder, Marc Paelinck, Greet Vanden Berghe. *Computers & Operations Research, 37(5),* 822-832, 2010. [PDF]

Preliminary Reading Material

This section contains an overview of the basic concepts that are needed in preparation to participate in the course. Section 1 provides an overview of several classic problem models widely studied in the field. These models are essential for grasping the fundamental concepts of the course. Section 2 covers the fundamentals of complexity theory, while Section 3 delves into the basics of heuristic search for optimisation.

In addition, it will be useful for students to have some experience with Java programming, Python programming and probability theory, but this is desirable, not a mandatory pre-requisite. At the start of the course, students will be asked to indicate their level of expertise in these topics to inform the delivery of the course.

Contents

1 Classic Optimisation Problems

1.1 Assignment problem

The assignment problem is a fundamental combinatorial optimisation problem. This problem arises in real-world scenarios like job scheduling, resource allocation, and logistics optimisation.

In its most general form, the assignment problem involves assigning a number of agents to a number of tasks with associated costs, aiming to minimise total assignment cost while ensuring each agent handles at most one task and vice versa.

1.2 Set Cover Problem

The set cover problem involves a finite set of N items and a set of available features U. The problem is to select the smallest subset of items that together include the subset $F \subseteq U$ of desirable features. Assume that $c_{ij} = 1$ if item i includes feature j, otherwise $c_{ij} = 0$.

The formulation of the Set Cover Problem is elaborated more in Section 3.4.

1.3 Boolean Satisfiability Problem (SAT)

SAT is a classic problem in computer science and mathematical logic. It mainly has three components:

- A set of m variables: $x_1, x_2, ...x_m$.
- A set of literals: A literal is either a variable (in which case it is called a positive literal) or a negation of a variable (called a negative literal).
- A set of n distinct clauses: $C_1, C_2, ..., C_n$. Each clause consists of only literals combined by just logical $or(\vee)$ connectors.

The goal of the SAT is to determine whether a given Boolean formula can be satisfied by assigning truth values to its variables.

In other words, to decide whehter the following conjunctive normal form (CNF) formula is satisfiable:

$$C_1 \wedge C_2 \wedge ... \wedge C_n$$

where $\wedge$ is a logical *and* connector.

1.4 SUBSET SUM (SSUM)

Subset-sum is a classic problem in computer science complexity theory. It is the following decision problem:

INSTANCE

- A set S of n integers: $a_1, a_2, ...a_m$.
- A "target" integer K

QUESTION: Does there exist a subset T of S, such that the sum of the elements in T is precisely equal to K

1.5 Bin Packing Problem

The bin packing problem involves a set of items $U = \{u_1, u_2, \ldots, u_n\}$ each with size given by a rational number s_i and a set of bins each with capacity equal to 1. The problem is to find a partition of U into disjoint subsets $U_1, U_2, \ldots, U_k$ such that the sum of sizes of the items in each U_i is no more than 1 and such that k is as small as possible. That is, the goal is to pack all the items in as few bins as possible.

1.6 Travelling Salesman Problem (TSP)

TSP is one of the most intensively studied problems in mathematics and computer science.

The classical TSP resolves around given a set of c cities $1, 2, \ldots, c$ and the distance $d(i, j)$ (a non-negative integer) between two cities i and j, finding the shortest tour that visits each city exactly once and returns to the starting city.

Asymmetric and symmetric variations exist within the TSP:

- Symmetric TSP: The distance between two cities remains consistent regardless of the direction of travel, resulting in an undirected graph. This symmetry reduces the number of potential solutions by half.
- Asymmetric TSP: In this scenario, paths may not be traversable in both directions, or the distances between cities may vary, creating a directed graph. Real-world factors such as traffic flow, one-way streets, and variable transportation costs may give rise to an asymmetric TSP.

1.7 Vehicle Routing Problem (VRP)

VRP is a widely investigated problem in logistics and transportation, serving as a generalisation of the TSP.

The basic VRP involves a single depot and multiple customer locations dispersed geographically. The aim is to optimise vehicle routes from the depot to satisfy customer demands while minimizing transportation costs.

The constraints commonly encountered in VRP include:

- Visit Constraints: Each customer must be visited exactly once by exactly one vehicle.
- Route Constraints: Each vehicle must start and end its route at the depot.
- Capacity Constraints: Each vehicle has a limited capacity that cannot be exceeded during the route.
- Time Windows: Some VRP variants include time windows, specifying the allowable time range for servicing each customer or reaching the final destination.

1.8 Nurse Rostering Problem (NRP)

NRPs represent extensively studied combinatorial optimisation challenges within healthcare institutions. These problems entail allocating nurses to designated shifts over defined work periods to fulfil operational needs while adhering to specific constraints.

In NRPs, hard constraints are mandatory, while soft constraints can be violated if necessary. The quality of the roster is evaluated based on the satisfaction of soft constraints.

Due to the diverse constraints among healthcare institutions, variations are considerable. Constraints may include limitations on consecutive assignments, prohibitions on simultaneous assignments for different personnel, and additional organisation-specific requirements tailored to the operational contexts.

1.9 Examination Timetabling Problems

Examination Timetabling Problems involve scheduling a set of exams taken by different students within a limited time frame. The aim is to minimise soft constraint violations while adhering to all hard constraints.

The constraints in exam timetabling problems can vary widely depending on the educational institution and the specific requirements of the exams, making exam timetabling a complex optimisation problem.

Some common hard constraints include:

- Room availability: Ensuring that each exam has an appropriately sized and available room.
- Student allocation: Exams with overlapping student enrollments cannot be scheduled simultaneously.

Soft constraints often include:

- Spread of exams: Ensuring that each student's exams are evenly distributed across the exam period to avoid consecutive exams.
- Priority scheduling: Assigning larger or more critical exams earlier in the schedule to balance workload and optimise resource allocation.
- Staff availability: Considering the availability of invigilators and other personnel necessary for exam administration.

Additional Resources for Problem Models

For a more comprehensive description and examples of the problem models, below is a quick reference to the Wikipedia links, some book chapters or surveys corresponding to each problem model.

Problem Models	**Additional Resources**
Assignment Problem	Assignment Problem Wikipedia
Set Cover Problem	Set Cover Wikipedia
Boolean Satisfiability Problem	SAT Wikipedia
Subset Sum	SSUM Wikipedia
Bin Packing Problem	BP Wikipedia, A survey of approximation algorithms for BP
Traveling Salesman Problem	TSP Wikipedia, A book chapter
Vehicle Routing Problem	VRP Wikipedia, A taxonomic review of VRPs
Nurse Rostering Problem	A bibliographic survey for NRP
Timetabling Problem	A survey of methodologies for exam timetabling

2 Basics of Complexity Theory

2.1 Computability

Computability focuses on answering questions such as:

- What class of problems can be solved using a computational algorithm?
- Is there an efficient computational algorithm to solve a given problem?

There have been many attempts by computer scientists to define precisely what is an effective procedure or computable function.

Intuitively, a computable function can be implemented on a computer, whether in a reasonable computational time or not.

An ideal formulation of an effective procedure or computable function involves:

1. A language in which a set of rules can be expressed.
2. A rigorous description of a single machine which can interpret the rules in the given language and carry out the steps of a specified process.

2.2 Computational Problems

A computational problem (e.g. the classical TSP introduced in Section 1.6) is generally described by:

- Its parameters (e.g. set of cities, distance between cities)
- Statement of the required properties in the solution (e.g. ordering of the cities such that the travelling distance is minimised)

An instance of a computational problem gives specific values to the problem parameters and a solution to that instance has a specific value.

The size of a problem instance is defined in terms of the amount of input data required to describe its parameters.

In practice, the instance size is conveniently expressed in terms of a single variable.

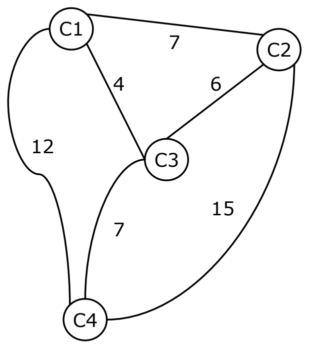
For example, the size of an instance of the TSP can be expressed by:

- A finite input string describing exactly all parameter values (cities and distances)
- A number intending to reflect the size of the input string (number of cities)

An encoding scheme is a mapping between the problem instance and the finite input string that describes the instance. A problem can be described or represented using different encoding schemes.

Example of some encoding schemes that can be used to describe instances of the TSP. Consider the problem instance shown below for these coding schemes.

Examples of encoding schemes are:



a) Vertex list-edge list
$\{c1, c2, c3, c4\}$
$\{(c1, c2), (c1, c3), (c1, c4), (c2, c3), (c2, c4), (c3, c4)\}$

b) Neighbour list
$c1 - \{c2, c3, c4\}$
$c2 - \{c3, c4\}$
$c3 - \{c4\}$

c) Adjacency matrix

$$\begin{bmatrix} & c_1 & c_2 & c_3 & c_4 \\ c_1 & 0 & 1 & 1 & 1 \\ c_2 & 1 & 0 & 1 & 1 \\ c_3 & 1 & 1 & 0 & 1 \\ c_4 & 1 & 1 & 1 & 0 \end{bmatrix}$$

2.3 Computational Algorithms

A computational algorithm is a deterministic step-by-step procedure to solve a computational problem. That is, the computational algorithm should produce a solution for any instance of the computational problem.

The interest is to find an efficient algorithm, in terms of computing resources, to solve a given problem.

Practical computing time is typically the factor which determines if an algorithm can be considered efficient or not.

The time complexity of a computational algorithm refers to the largest amount of time required by the algorithm to solve a problem instance of a given size.

For a given problem, how to assess if a computational algorithm is efficient?

Computational perspective: polynomial time vs. exponential time complexity.

The complexity of a function $f(n)$ is $O(g(n))$ if $|f(n)| \leq c \cdot |g(n)|$ for all $n > 0$ and some constant c.

An algorithm has polynomial time complexity if its complexity function is given by $O(g(n))$ where $g(n)$ is a polynomial function for the input length n.

2.4 Intractable and Undecidable Problems

A computational problem is considered intractable if no computationally efficient algorithm can be given to solve it, or in other words if no polynomial time algorithm can be found to solve it.

Although polynomial time algorithms are preferred, some exponential time algorithms are considered good for solving some problems. This is because the algorithm may have exponential complexity in the worst case but it performs satisfactorily on the average.

The tractability of a given problem is independent of the encoding scheme and the computer model.

The tractability of a problem can refer to:

- An exponential amount of computational time is needed to find a solution. For example, solving the TSP with algorithm A3.
- A solution is too long that cannot be described with a polynomial expression on the length of the input. For example, in a TSP instance, asking for all tours that have length L or less.

A computational problem is considered undecidable if there is no algorithm at all that can be given to solve it. Therefore, an undecidable problem is also intractable.

Many intractable problems are decidable and can be solved in polynomial time with a non-deterministic computing model.

The NP-complete class of problems include decision problems (the answer is either YES or NO) that can be solved by a non-deterministic computing model in polynomial time.

The conjecture is that NP-complete problems are intractable but this is a very much open question.

But at least, if the problem is NP-complete there is no known polynomial time algorithm to solve it.

2.5 Time Complexity

We attempt to separate the practically solvable problems from those that cannot be solved in a reasonable time.
When constructing a computer algorithm it is important to assess how expensive the algorithm is in terms of storage and time.
Algorithm complexity analysis tells how much time does an algorithm take to solve a problem.

The size of a problem instance is typically measured by the size of the input that describes the given instance.

Algorithm complexity analysis is based on counting primitive operations such as arithmetic, logical, reads, writes, data retrieval, data storage, etc.

$time_A(n)$ expresses the time complexity of algorithm A and is defined as the greatest number of primitive operations that could be performed by the algorithm given a problem instance of size n.

We are usually more interested in the rate of growth of $time_A$ than in the actual number of operations in the algorithm.

Thus, to determine whether it is practical or feasible to implement A we require a not too large upper bound for $time_A(n)$.

Time complexity of algorithms is expressed using the "Big-Oh" notation.

The complexity of a function $f(n)$ is $O(g(n))$ if $|f(n)| \leq c \cdot |g(n)|$ for all $n > 0$ and some constant c.

Let f and g be functions on n, then:

$f(n) \in O(g(n))$ means that "f is of the order of g", that is, "f grows at the same rate of g or slower".

$f(n) \in \Omega(g(n))$ means that "f grows faster than g".

$f(n) \in \Theta(g(n))$ means that "f and g grow at the same rate".

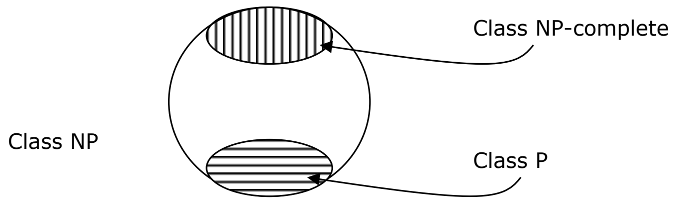
An algorithm is said to perform in polynomial time if and only if its time complexity function is given by $O(n^k)$ for some $k \geq 0$.

That is, an algorithm A is polynomial time if $\mathrm{time}_A(n) \leq f(n)$ and f is a polynomial, i.e., f has the form: $a_1 n^k + a_2 n^{k-1} + \ldots + a_m$.

Then, the class **P** is defined as the class of problems which can be solved by polynomial time deterministic algorithms. The notion is that problems in **P** can be solved 'quickly'.

Then, the class **NP** is defined as the class of problems which can be solved by polynomial time non-deterministic algorithms. The notion is that problems in **NP** can be verified 'quickly'.

Then, within the class **NP** there are easy problems, those in **P**, and very much harder problems called **NP-complete** problems.



Intuitively, the class **NP-complete** are the hardest problems in **NP** as defined by Cook's theorem.

2.6 Tackling NP-Complete Problems

There are two approaches:

1. Search techniques that although of exponential time complexity aim for optimality and are an improvement over exhaustive search. These include:
   (a) techniques that construct partial solutions, e.g. branch and bound, and
   (b) techniques that reduce the worst time complexity by making clever choices while searching the tree, e.g. specific algorithms for some NP-complete problems.

2. Search techniques that do not aim to find the optimal solutions but instead aim for sub-optimal solutions, that is:
   (a) approximation solution that is optimal up to a small constant factor, e.g. approximation algorithms, or
   (b) good enough solution reasonably fast, e.g. local search heuristics.

A combinatorial optimisation problem (maximization or minimization) $\prod$ consists of three parts:

1. A set of instances D
2. For each instance I, a finite set $S_D(I)$ of candidate solutions to I
3. An evaluation function F that assigns to each instance I and each candidate solution $s \in S_D(I)$ a positive rational number $F(I, s)$ called the solution value of s.

If $\prod$ is a maximisation problem then:
An optimal solution s^* is one that for all $s \in S_D(I)$, $F(I, s^*) \leq F(I, s)$ then OPT(I) is used to denote the value for an optimal solution to the instance I.

An algorithm A is called an approximation algorithm if it finds a solution $s \in S_D(I)$ with value A(I). The algorithm is called an optimisation algorithm if A(I)=OPT(I).

If $\prod$ is NP-complete, no polynomial time algorithm can be found unless **P=NP**.

The option then is to find an approximation algorithm or heuristic, called algorithm A, that runs in 'reasonable time' and for each instance I of $\prod$, A(I) is 'close' to OPT(I).

3 Basics of Heuristic Search for Optimisation

3.1 Optimisation Problems

Basic ingredients of optimisation problems:

- A set of decision variables which affect the value of the objective function. In the TSP, the variables might be of the type $X_{ij}\{0, 1\}$ where the value of the variable indicates whether city j is visited after city i or not; in fitting-the-data problem, the variables are the parameters that define the model.
- An objective function which we want to minimise or maximise. It relates the decision variables to the goal and measures goal attainment. For instance, in the TSP, we want to minimise the total distance travelled for visiting all cities in a tour; in fitting experimental data to a user-defined model, we might minimise the total deviation of observed data from predictions based on the model.
- A set of constraints that allow the variables to take on certain values but exclude others. For some variations of the TSP, it may not be allowed to visit some cities in a certain sequence, so restricting the values that certain decision variables can take in a solution to the problem instance.

The optimisation problem is defined as follows:

Find values for all the decision variables that minimise or maximise the objective function while satisfying the given constraints.

Formal representation of optimisation problems are described by mathematical programming.

Mathematical programming is used to describe problems of choosing the best element from some set of available alternatives or in OR terms to describe problems how to allocate limited resources.

Mathematical formulation:

minimize $f(x)$

subject to

$g_i(x) \geq b_i, \quad i = 1, \dots, m$

$h_j(x) = c_j, \quad j = 1, \dots, n$

where $\mathbf{x}$ is a vector of decision variables, $f(\cdot), g_i(\cdot), h_i(\cdot)$ are general functions.

Similar formulation of the problem holds when the objective function has to be maximised.

Placing restrictions on the type of functions under consideration and on the values that the decision variables can take leads to specific classes of optimisation problems.

A solution which satisfies all constraints is called a feasible solution.

3.2 Linear Programming (LP)

- In an LP formulation, $f(\cdot)$, $g_i(\cdot)$, $h_i(\cdot)$ are linear functions of decision variables.
- LP is one of the most important tools in OR.
- The typical LP problem involves:
  - limited resources
  - competing activities
  - measure of solution quality
  - constraints on the use of resources
  - the optimal solution
- LP is a mathematical model with:
  - decision variables and parameters
  - linear algebraic expressions (objective function and constraints)
  - planning activities

A standard LP formulation for the problem of finding the optimal allocation of limited resources to competing activities. Note that constraints can involve inequalities $\geq$ or $\leq$, strict or not, and equalities.

$$\boxed{\begin{array}{ll} \text{Maximise}: & Z = c_1x_1 + c_2x_2 + \dots c_nx_n \\ \text{Subject to}: & \\ & a_{11}x_1 + a_{12}x_2 + \dots a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \dots a_{2n}x_n \leq b_2 \\ & \vdots \qquad\qquad\qquad\qquad\qquad \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \dots a_{mn}x_n \leq b_m \\ & x_1 \geq 0, x_2 \geq 0 \dots x_n \geq 0 \end{array}}$$

Functional constraints

Non-negativity constraints

3.3 Integer Programming (IP)

In many applications, the solution of an optimisation problem makes sense only if decision variables are integers.

Therefore, a solution $X \in Z^n$, where Z^n is the set of n-dimensional integer vectors. In mixed-integer programs, some components of X are allowed to be real. In general, integer linear problems are much more difficult than regular linear programming.

Integer programming problems, such as TSP, are often expressed in terms of binary variables.

Linear and integer programming have proved valuable for modelling many and diverse types of problems in manufacturing, routing, scheduling, assignment, and design. Industries that make use of LP and its extensions include transportation, energy, telecommunications, and manufacturing of many kinds.

3.4 Discrete/Continuous/Combinatorial Optimisation

In discrete optimisation, the variables used in the mathematical programming formulation are restricted to assume only discrete values, such as the integers.

In continuous optimisation, the variables used in the objective function can assume real values, e.g., values from intervals of the real line.

In combinatorial optimisation, decision variables are discrete, which means that the solution is a set, or sequence of integers or other discrete objects. The number of possible solutions is a combinatorial number (an extremely large number, such as 10^{50}). Combinatorial optimisation problems include problems on graphs, matroids, and other discrete structures.

Let's illustrate with an example using the set cover problem (refer to Section 1.2).

Mathematical programming formulation for the set covering problem:

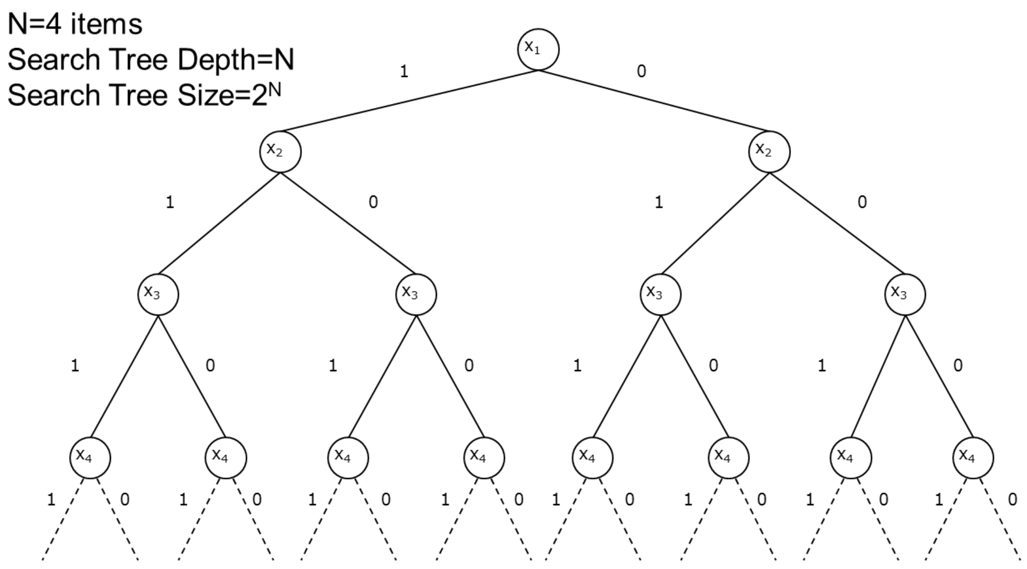
$$\begin{array}{ll}
\text{Minimise:} & Z = \sum_{i=1}^{N} x_i \\
\text{Subject to:} & \sum_{i=1}^{N} c_{ij} x_i \geq 1 \;\; \text{for } j = 1,2,\ldots,F \qquad (1) \\
 & x_i = 1 \;\; \text{if item } i \text{ is selected, 0 otherwise} \\
 & c_{ij} = 1 \;\; \text{if item } i \text{ has feature } j\text{, 0 otherwise}
\end{array}$$

The mathematical formulation for a specific set covering problem instance:

$$\begin{array}{ll}
\multicolumn{2}{l}{\text{BIP model for problem instance for } N = 8 \text{ and } F = 4} \\
\text{Minimise:} & Z = x_1 + x_2 + x_3 + x_4 + x_5 + x_5 + x_7 + x_8 \\
\text{Subject to:} & c_{11}x_1 + c_{21}x_2 + c_{31}x_3 + c_{41}x_4 + c_{51}x_5 + c_{61}x_6 + c_{71}x_7 + c_{81}x_8 \geq 1 \\
 & c_{12}x_1 + c_{22}x_2 + c_{33}x_3 + c_{42}x_4 + c_{52}x_5 + c_{62}x_6 + c_{72}x_7 + c_{82}x_8 \geq 1 \\
 & c_{13}x_1 + c_{23}x_2 + c_{33}x_3 + c_{43}x_4 + c_{53}x_5 + c_{63}x_6 + c_{73}x_7 + c_{83}x_8 \geq 1 \\
 & c_{14}x_1 + c_{24}x_2 + c_{34}x_3 + c_{44}x_4 + c_{54}x_5 + c_{64}x_6 + c_{74}x_7 + c_{84}x_8 \geq 1 \\
 & x_i \in \{0,1\} \;\; \text{for } i = 1 \text{ to } N
\end{array}$$

Why is Set Covering a difficult optimisation problem to solve?

For a Set Covering instance with $N = 4$ decision variables x_{ij}, the corresponding search tree is shown below. Each branch in the tree represents a potential solution, and a search algorithm needs to explore this search space. As the number of decision variables increases, the search tree may grow exponentially.



3.5 Approximation Algorithms

Let's use the bin packing problem, as introduced in Section 1.5, to explore some approximation algorithms for its solution.

The 'First Fit' (FF) algorithm:

- Given the sequence of unit-capacity bins $B_1, B_2, B_3, \ldots$
- Place the items into the bins, one at a time. Item u_i is placed in the lowest-indexed bin with enough remaining capacity for s_i, which is the item size.

It has been proven that the worst performance: $\text{FF}(I) \leq 1.7\text{OPT}(I) + 2$.

The 'Best Fit' (BF) algorithm:

- Given the sequence of unit-capacity bins $B_1, B_2, B_3, \ldots$
- Place the items into the bins, one at a time. Item u_i is placed in the bin with enough remaining capacity closest to but not exceeding $1 - s_i$.

It has been proven that the worst performance: $\text{BF}(I) \leq 1.7\text{OPT}(I) + 2$.

The performance of the FF and BF approximation algorithms for Bin Packing depends very much on the ordering of the items, in particular whether larger or smaller items appear before in the sequence.

The 'First Fit Decreasing' (FFD) algorithm:

- Given the sequence of unit-capacity bins $B_1, B_2, B_3, \ldots$.
- Order the items in non-increasing order of their size.
- Place the items into the bins, one at a time. Item u_i is placed in the lowest-indexed bin with enough remaining capacity for s_i, which is the item size (as in FF).

It has been proven that the worst performance: $\text{FFD}(I) \leq 1.22\text{OPT}(I) + 4$.

3.6 Greedy Algorithms and Neighbourhood Search

Real-world optimisation problems (combinatorial or continuous) are difficult to solve for several reasons:

- Size of the search space is very large (exhaustive search impractical)
- Simplified models are needed to facilitate an answer
- Difficult to design an accurate evaluation function
- Existence of a large number of constraints
- Difficult to collect accurate data
- Human limitations to construct solutions

Hard constraints must be satisfied for a solution to be feasible.

Soft constraints are not mandatory but desirable conditions in good quality solutions.

Given search space S and a set of feasible solutions $F \subseteq S$, a search problem is to find a solution $x \in S$ such that:

$fitness\_function(x) \leq fitness\_function(y) \quad \forall y \in F$

Two solutions x, y are said to be neighbors if they are 'close' to each other in the sense that the distance between their corresponding encodings or representations is within a certain limit.

Some search algorithms work with complete solutions, i.e., they try to go from the current solution to a better one. Example: neighborhood search.

Other search algorithms work with partial solutions, i.e., they construct a complete solution step by step. Example: greedy algorithms.

Example. Consider a symmetric n-city TSP over a complete graph with solutions encoded as permutations.

permutation of integers 1 to n	e.g., 13546278
total number of permutations: $n!$	
ignoring symmetric tours: $\frac{n!}{2}$	e.g., 13546278 = 87264531
ignoring shifted identical tours: $\frac{(n-1)!}{2}$	e.g., 13546278 = 54627813

Evaluation Function

S is set of solutions and a solution $s = c_1c_2c_3...c_n$ then
$f(s) = d(c_1, c_2) + d(c_2, c_3) + .. + d(c_{n-1}, c_n) + d(c_n, c_1)$

Objective

find $s \in S$ such that $f(s) \leq f(s') \quad \forall s' \in S$
i.e., find a tour with the minimum length

Neighbourhood Solutions

2-opt move: interchanges 2 non-adjacent edges
current:13546278, neighbours: 13246578, 13746258, 13586274, etc.

2-right_insert move: moves a city 2 positions to the right
current: 13546278, neighbours: 15436278, 13542768, etc.

Greedy Algorithm 1

1. Select random starting city
2. Proceed to nearest unvisited city
3. Repeat step 2 until all cities are visited
4. Return to starting city

Greedy Algorithm 2

1. Find shortest edge (c_i, c_j) and add cities c_i, c_j to the tour
2. Select next cheapest edge (c_r, c_t) (making sure no city is visited more than once)
3. Add cities c_r, c_t to the tour
4. Repeat steps 2-3 until a tour is formed

In combinatorial optimisation problems, the neighbourhood of a solution x in the search space S can be defined as:

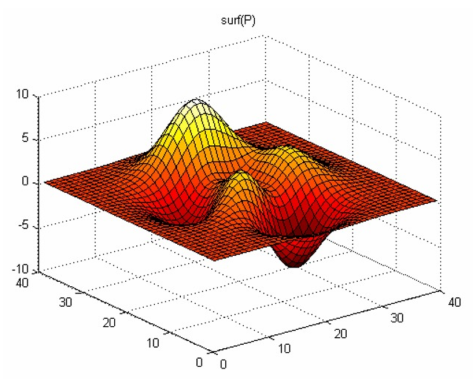
$N(x) = \{y \in S | y = \Phi\}$, where Φ is a move or sequence of moves.

A feasible solution x is a local optimum with respect to $N(x)$ if:

$f(x) \leq f(y) \quad \forall y \in N(x)$ (assuming minimization)

If the inequality above is strict then x is a strict local optimum.

The graph below illustrates a fitness landscape with local optima, global optima, hills, valleys and plateaus.



Neighbourhood search refers to exploring solutions within the neighbourhood of the current solution with the goal of finding the best solution in the vicinity, i.e. a local optimum.

Trade-off in neighbourhood search: $|N(x)|$ vs. Search Time

Note: defining appropriate neighbourhoods according to the problem domain and the search strategy is crucial in heuristic local search.

Neighbourhood Size

The k-exchange neighbourhood is widely used in many combinatorial optimisation problems.

Typically, |k-exchange neighbourhood| $= O(n^k)$

The Neighbourhood pruning: consider only a subset of neighbours which are selected based on the state of the current solution.

Some Strategies for Escaping Local Optima

- Re-initialize the search from a different starting solution
- Accept non-improving candidate solutions to replace the current solution in a deterministic or probabilistic manner
- Explore only a fraction of the neighborhood (e.g., pruning)
- Use more than one neighborhood to generate candidate solutions
- Design composite (independent or not) neighborhoods
- Maintain a memory of already visited solutions or regions
- Modify the fitness function to change the search landscape
- Perturb local optima in a deterministic or probabilistic manner
- Improve more than one solution simultaneously

3.7 Basics of Meta-heuristics

Further preliminary reading: `http://www.scholarpedia.org/article/Metaheuristics`

Greedy heuristics and simple local search have some limitations with respect to the quality of the solution that they can produce.

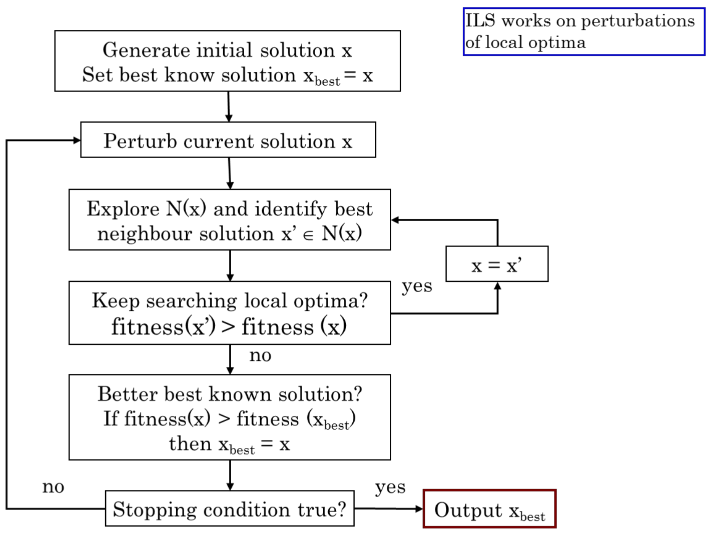
A good balance between intensification and diversification is crucial in order to obtain high-quality solutions for difficult problems.

A meta-heuristic can be defined as "an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high-quality solutions. It may manipulate a complete (or incomplete) single solution or a collection of solutions at each iteration. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method" (Voss et al. 1999).

A meta-heuristic method is meant to provide a generalized approach that can be applied to different problems. Meta-heuristics do not need a formal mathematical model of the problem to solve.

An example of a basic meta-heuristic is the iterated local search approach, an intuitive method to combine intensification and diversification. The local search and perturbation operators should be carefully designed so that they work well in combination.

Flow Diagram


ILS works on perturbations of local optima
Generate initial solution x
Set best know solution x_{best} = x
Perturb current solution x
Explore N(x) and identify best neighbour solution x' ∈ N(x)
x = x'
yes
Keep searching local optima? fitness(x') > fitness (x)
no
Better best known solution? If fitness(x) > fitness (x_{best}) then x_{best} = x
no
Stopping condition true?
yes
Output x_{best}

Some Types of Meta-heuristics

- Single-solution method vs. Population-based methods
- Nature-inspired method vs. Non-nature-inspired methods
- 'Pure' methods vs. Hybrid methods
- Memory-based vs. Memory-less methods
- Deterministic vs. Stochastic methods
- Iterative vs. Greedy methods

The following algorithms are examples of meta-heuristics:

- Iterated local search
- Threshold acceptance
- Great deluge
- Simulated annealing
- Greedy randomized search procedure
- Guided local search
- Variable neighborhood search
- Tabu search
- Evolutionary algorithms
- Particle swarm optimisation
- Artificial immune systems
- Etc.

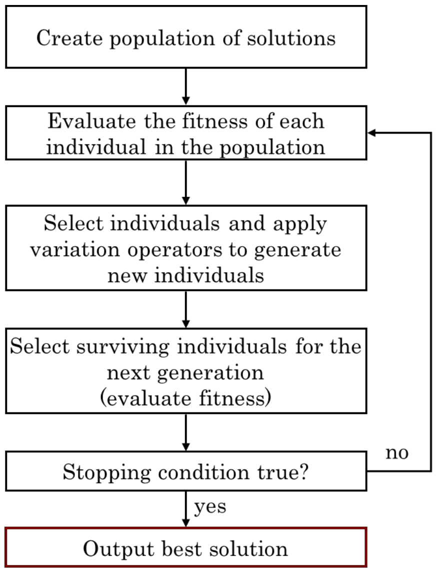
3.8 Basics of Evolutionary Algorithms

The rationale for evolutionary algorithms (EAs) is to maintain a population of solutions during the search. The solution (individuals) compete between them and are subject to selection, and reproduction operators during a number of generations.

Exploitation vs. Exploration

- Having many solutions instead of only one.
- Survival of the fittest principle.
- Pass on good solution components through recombination.
- Explore and discover new components through self-adaptation.
- Solutions are modified from generation to generation by means of reproduction operators (recombination and self-adaptation).

Typical Evolutionary Algorithm Cycle


Create population of solutions
Evaluate the fitness of each individual in the population
Select individuals and apply variation operators to generate new individuals
Select surviving individuals for the next generation (evaluate fitness)
Stopping condition true?
no
yes
Output best solution

Typical Components of an EA

- Solution representation
- Population size
- Initialization strategy
- Evaluation function
- Reproduction operators
- Selection mechanism
- Stopping condition to detect convergence

A diverse range of evolutionary procedures can be designed for the subject problem by tuning the above components.

It is crucial to design components and tune parameters while considering the choices made on the various parts of the algorithm.

Some Examples of Evolutionary Algorithms

- Genetic Algorithms (GA)
- Evolutionary Strategies (ES)
- Genetic Programming (GP)
- Ant Colony Optimisation (ACO)
- Memetic Algorithms (MA)
- Particle Swarm Optimisation (PSO)
- Differential Evolution (DE)
- Estimation of Distribution Algorithm (EDA)
- Cultural Algorithms (CA)

Since this is a very active research area, new variants of EAs and other population-based meta-heuristics are often proposed in the specialized literature.

In optimisation problems, the goal is to find the best feasible solution.

Issues When Dealing with Infeasible Solutions

- Evaluate quality of feasible and infeasible solutions
- Discriminate between feasible and infeasible solutions
- Decide if infeasibility should be eliminated
- Decide if infeasibility should be repaired
- Decide if infeasibility should be penalized
- Initiate the search with feasible solutions, infeasible ones or both
- Maintain feasibility with specialized representations, moves, and operators
- Design a decoder to translate an encoding into a feasible solution
- Focus the search on specific areas, e.g., in the boundaries

A constraint handling technique is required to help the effective and efficient exploration of the search space.