Coping with hard computational problems

A large number of optimisation problems which are required to be solved in practice are NP-hard.

These problems are unlikely to have an efficient algorithm.

This does not obviate the need for solving these problems.

Observe that NP-hardness only means that, if P is not equal to NP, we cannot find algorithms which will find the optimal solution to all instances of the problem in time which is polynomial in the size of the input.

There are three possibilities for relaxing the requirements outlined above.

Super-polynomial time heuristics

In some cases there are algorithms which are just barely super-polynomial and run reasonably fast in practice.

There are techniques such as branch-and-bound or dynamic programming which are useful from this point of view.

For example, the Knapsack problem is NP-complete but it is considered “easy” since there is a “pseudo-polynomial” time algorithm for it.

A problem with this approach is that very few problems are susceptible to such techniques and for most NP-hard problems the best algorithm we know runs in truly exponential time.

Probabilistic analysis of heuristics

In some applications, it is possible that the class of input instances is severely constrained and for these instances there is an efficient algorithm which will always do the trick.

Consider for example the problem of finding Hamiltonian cycles (i.e. simple cycles that visit every vertex in the given graph) in graphs. This is NP-hard. However, it can be shown that there is an algorithm which will find a Hamiltonian cycle in “almost every” graph which contains one.

Such results are usually derived using a probabilistic model of the constraints on the input instances. It is then shown that certain heuristics will solve the problem with very high probability.

Unfortunately, it is usually not very easy to justify the choice of a particular input distribution.

Approximation algorithms

In practice, it is usually hard to tell the difference between an optimal solution and a near-optimal solution.

It seems reasonable to devise algorithms which are really efficient in solving NP-hard problems, at the cost of providing solutions which in all cases is guaranteed to be only slightly sub-optimal.

In some situations, this relaxation of the requirements for solving a problem appears to be the most reasonable.

This results in the notion of the “approximate” solution of an optimisation problem.

The stress will be (AS ALWAYS) on algorithms for which one can PROVE some kind of *performance guarantee*.

I.e. there may be more refined algorithms that seem to do better in practice, but for which nobody is able to prove anything!

Preliminaries

The definition of optimisation problem was given informally in Lecture 1.

From now on we will be concerned with a particular class of optimisation problems.

An *NP optimisation problem* (NPO) is defined by four components $(\mathcal{I}, \mathcal{SOL}, c, opt)$ where:
(1) $\mathcal{I}$ is the set of the *instances*. Membership in $\mathcal{I}$ is decidable in polynomial time.
(2) For each $x \in \mathcal{I}$, $\mathcal{SOL}(x)$ is the set of *solutions* of x. Membership in $\mathcal{SOL}(x)$ is decidable in polynomial time and for each $y \in \mathcal{SOL}(x)$, the length of y, $|y|$ is polynomial in the length of x.
(3) For each $x \in \mathcal{I}$ and each $y \in \mathcal{SOL}(x)$, $c(x, y)$ is an integer, non-negative function, called the *objective* or *cost* function.
(4) $opt \in \{\max, \min\}$ is the *optimisation criterion* and tells if the problem P is a maximisation or a minimisation problem.

Example: vertex covers

If $\mathcal{I}$ is the set of undirected graphs, $\mathcal{SOL}(G)$ is, for every $G \in \mathcal{I}$, the collection of all sets of vertices $U \subseteq V(G)$ such that every edge in G has at least one end point in U, $c(G, U) = |U|$ for every $U \in \mathcal{SOL}(G)$, and $opt = \min$ then the problem under consideration is that of finding a, so called, *vertex cover* of the edges of minimum cardinality (denoted by MINVC).

The number of vertices of this optimal cover is a graph parameter normally denoted by $\tau(G)$.

Example: bin-packing

We are given a collection of items, each with an associated size (a number between 0 and 1). We are required to pack them into bins of unit size so as to minimise the number of bins used. Thus, we have the following minimisation problem.

Instances Sets $\{s_1, s_2, \ldots, s_n\}$ of numbers with $s_i \in [0, 1]$ for all i (for example $\{0.1, 0.8, 0.3, 0.5\}$ and $\{0.2, 0.2, 0.7, 0.9\}$ are two instances of size four).

Solutions A collection of subsets $\{B_1, B_2, \ldots, B_k\}$ which is a partition of $\mathcal{I}$, such that for all i $\sum_{j \in B_i} s_j \leq 1$ (for example $B_1 = \{0.1, 0.8\}$ and $B_2 = \{0.3, 0.5\}$ is a solution to the first instance given above).

Objective The cost of a solution is the number of bins used.

Optimisation Criterion Minimisation!

Technical assumptions

We would like to specify at the outset that an underlying assumption throughout this book will be that the optimisation problems satisfy the following two technical conditions.

1. The range of the cost function and all the numbers in $\mathcal{I}$ have to be integers. Note that we can easily extend this to allow rational numbers since those can be represented as pairs of integers. For example, in the Bin Packing problem we will assume all item sizes are rationals.
2. For any $y \in \mathcal{SOL}(x)$, $c(x, y)$ is polynomially bounded in the size of any number which appears in $\mathcal{I}$.

Approximation algorithm: the definition

Given an NP-hard optimisation problem, it is clear that we cannot find an algorithm which is guaranteed to compute an optimal solution in polynomial time for all input instances, unless P=NP. We now relax the requirement of optimality and ask for an approximation algorithm. This is defined as follows.

An *approximation algorithm* A, for an optimisation problem, is a $\boxed{\text{polynomial time}}$ algorithm such that given an input instance x, it will output $\boxed{\text{some}}$ $A(x) \in \mathcal{SOL}(x)$.

Note that we are only interested in polynomial time algorithms.

Example

Consider the Bin Packing problem. Let DA (Dumb Algorithm) be an algorithm which packs each item into a bin by itself.

Clearly, this is an approximation algorithm for the problem BP (!)

Of course it is not a very good approximation algorithm in the sense that the number of bins it uses need not be close to the optimal number of bins.

Thus, we need some way of comparing approximation algorithms and analysing the quality of solutions produced by them.

Moreover, the "measure of goodness" of an approximation algorithm must relate the optimal solution to the solution produced by the algorithm.

Such measures are referred to as performance guarantees and the exact choice of such a measure is not obvious a priori. We will explore two[a] notions of performance guarantees in what follows.

[a] What do you think is the most natural choice of such a measure?

Absolute Performances

We know that packing a collection of items into the smallest possible number of bins is "impossible".

So what is the next best solution that we could obtain?

Clearly, this would be a solution which uses at most one extra bin when compared to the optimal solution.

In general, it would be desirable to have a solution whose value differs from the optimal by some small constant. This is formalised in the *absolute performance measure*.

An *absolute approximation algorithm* A for an NPO optimisation problem is an approximation algorithm such that for some constant $k > 0$,

$$\forall x \in \mathcal{I}, \quad |c(x, A(x)) - \mathrm{opt}(x)| \leq k$$

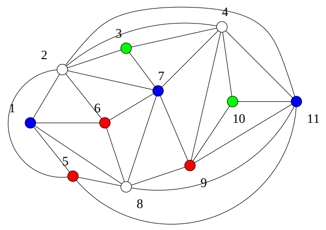
This is clearly the best we can expect from an approximation algorithm for any NP-hard problem. But can we find such algorithms? We give below a couple of examples where such algorithms are possible to find.

Example 1: Planar Vertex Colouring

Consider the problem of colouring the vertices of a graph such that no two adjacent vertices have the same colour.

The goal is to minimise the number of colours used. We denote by $\chi(G)$ the minimum number of colours needed to colour G (this is also known as the *chromatic number* of G).

The decision version of this problem is NP-hard even when restricted to graphs that are cubic and planar



We will show that the planar graph colouring problem has an absolute approximation algorithm.

Simple facts about planar graphs

- (The magic Euler formula) If G is a connected planar graph on n vertices, m edges then any plane embedding of G on the plane has

$$\ell = m - n + 2$$

  faces.
- Every planar graph on n vertices has at most $3n - 6$ edges.
- Every planar graph has a vertex of degree at most 5.
- (Exercise) If G is a planar graph then $\chi(G) \leq 6$.

Every planar graph is 5-colorable[a]

By induction on the number of vertices n.

BASE: For any planar graph with at most 5 vertices the result is trivial (just give each vertex a different colour).

STEP: We assume that all planar graphs with at most n vertices are 5-colourable. Let G be a planar graph with $n + 1$ vertices. Assume w.l.o.g. that the given graph is connected (if not the inductive hypothesis applies directly to each component).

Let v be a vertex of degree at most 5 in G, and let $N(v) = \{v_i : 1 \leq i \leq 5\}$. By the induction hypothesis $G \setminus v$ is 5-colourable. Let c be one such a colouring.

Case 1. (the easy one) If one of the 5 colours is not used to colour $N(v)$ we can colour v with it and complete the colouring of G.

[a] In fact, the (in)famous Four Color Theorem for planar maps [AH] tells us that every planar graph is 4-colorable.

Case 2. (using "Kempe chains") Without loss of generality assume $c(v_i) = c_i$.

Let G_{13} be the subgraph of $G \setminus v$ induced by the vertices coloured c_1 and c_3.

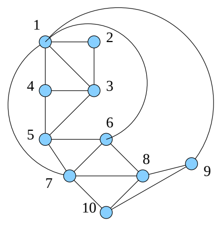
If v_1 and v_3 belong to different components of G_{13} then interchange the colours of the vertices in the component containing v_1. Vertex v can now be coloured c_1.

Otherwise if v_1 and v_3 belong to the same component then there exists a path P between v_1 and v_3 such that $P + v$ forms a cycle which necessarily encloses the vertex v_2 or both v_4 and v_5. We can then complete the colouring using G_{24} and assigning c_2 to v.

Theorem. There is an absolute approximation algorithm for the planar graph colouring problem.

Proof. Consider the algorithm that computes a 5-colouring of the given planar graph (in polynomial time). It is easy to see that A never uses more than 2 extra colours.

- Describe using pseudo-code the 5-colouring algorithm.
- What is the running time of this algorithm?
- Run your algorithm on the graph below


1
2
3
4
5
6
7
8
9
10