NP-Complete Problems

The argument at the end of the last section, which takes us from the NP-completeness of CNF-SAT to the NP-completeness of 3SAT can be formalised in terms of the composition of reductions. That is, if for any languages L_1, L_2 and L_3, we have that $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then it must be the case that $L_1 \leq_P L_3$. In other words, the relation $\leq_P$ between languages is transitive. The reason is that if f is a reduction from L_1 to L_2, and g a reduction from L_2 to L_3, then their composition $g \circ f$ is a reduction from L_1 to L_3. Moreover, if both f and g are polynomial time computable, then so is their composition, by the algorithm that first computes f on its input x and then computes g on the result $f(x)$. There is a polynomial bound on the running time of this algorithm because the length of $f(x)$ must be bounded by a polynomial in the length of x, so even though the input to the algorithm computing g is $f(x)$, the total running time is bounded by a polynomial in the length of x.

It is also possible to show that logarithmic space reductions are closed under compositions, so the relation $\leq_L$ is transitive as well, but this is a bit more involved to prove. One cannot simply carry out one logarithmic space computation after another to compute the composed reduction $g \circ f$, as one does not have the space to store the intermediate result $f(x)$. Rather, the algorithm computing g, whenever it requires a symbol from its input has to restart the computation of f until the required symbol is produced. This is similar to the construction used in the proof that Reachability can be done in space $O((\log n)^2)$.

By the transitivity of reducibility, and our previous proofs of NP-completeness, it follows that if we show, for any language A in NP, that SAT $\leq_P A$ or that 3SAT $\leq_P A$, it immediately follows that A is NP-complete. We now use this to establish the NP-completeness of a number of natural combinatorial problems.

Graph Problems

We begin by looking at problems involving graphs.

Independent Set Let $G = (V, E)$ be an undirected graph with a set V of vertices, and E of edges. We say that $X \subseteq V$ is an *independent set* if there are no edges (u, v) in E, for any $u, v \in X$. The definition gives rise to a natural algorithmic problem, namely, given a graph G, find the largest independent set. Instead of this optimisation problem, we will consider a decision problem which we call IND, which is defined as

> The set of pairs (G, K), where G is a graph, and K is an integer, such that G contains an independent set with K or more vertices.

That is, we turn the question into a yes/no question by explicitly setting a target size in the input.

The problem IND is clearly in NP. We can nondeterministically generate an arbitrary subset X of the vertices, and then in polynomial time check that X has at least K elements and that it is an independent set.

To show that IND is NP-complete, we construct a reduction from 3SAT to IND. The reduction maps a Boolean expression ϕ in 3CNF with m clauses to the pair (G, m) where G is a graph, and m the target size. G is obtained from the expression ϕ as follows.

G contains m triangles, one for each clause of ϕ, with each node representing one of the literals in the clause.

Additionally, there is an edge between two nodes in different triangles if they represent literals where one is the negation of the other.

As an example, if ϕ is the expression

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_2 \vee \neg x_1)$$

we obtain a graph G with six nodes, connected by edges as in Figure 5, where the triangle of vertices at the top corresponds to the first clause and the triangle at the bottom to the second clause of ϕ.

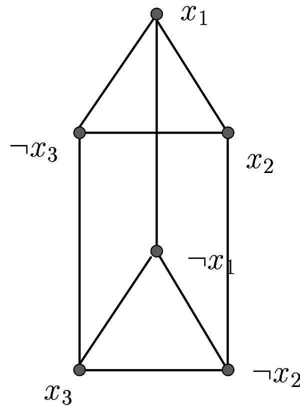

Figure 5: Reduction from 3SAT to IND

To prove that this is a polynomial time reduction from 3SAT to IND, we need to show that the transformation from ϕ to (G, m) can be carried out by a polynomial time algorithm. This is not too difficult to see, as G is really just a direct encoding of ϕ as a graph, and m is obtained by counting the clauses of ϕ. We also need to check that G contains an independent set of m vertices *if, and only if,* ϕ is satisfiable.

For one direction, suppose ϕ is satisfiable, and let T be a truth assignment that satisfies it. For each clause in ϕ, choose exactly one literal in the clause which is made true by T (there must be at least one in each clause, since T satisfies ϕ). Let X be the set of vertices in G corresponding to the literals we have just chosen. X cannot contain a vertex labeled by a variable x and another vertex labeled by $\neg x$, otherwise T would not be a consistent truth assignment. Also, X cannot contain two vertices from the same triangle, since we chose exactly one vertex in each clause. So, there are no edges between vertices in X. Furthermore, since we chose one vertex from each triangle, there are m vertices in X.

In the other direction, we have to show that if G has an independent set with m vertices, then ϕ is satisfiable. Let X be such an independent set. Any two vertices arising from the same clause are part of a triangle, and so cannot be both in X. So, X must contain exactly one vertex from each triangle. We now define a truth assignment T for ϕ as follows. For any variable x, if there is a vertex labeled x in X, then let $T(x) = \texttt{true}$, and if there is a vertex

labeled $\neg x$ in X, then let $T(x) = \texttt{false}$. If X contains neither a vertex labeled x nor a vertex labeled $\neg x$, then set $T(x)$ arbitrarily. We can see that this is a consistent assignment of truth values to the variables of ϕ, since X cannot contain both a vertex labeled x and a vertex labeled $\neg x$ as there would be an edge between them. Finally, we note that T is a truth assignment that satisfies ϕ, since for each clause of ϕ, there is one literal corresponding to a vertex in X, and which is therefore made true by T.

Clique A graph problem closely related to IND, and perhaps more commonly mentioned, is the problem of finding a *clique* in a graph. Once again, we begin with a definition. Given a graph $G = (V, E)$, a subset $X \subseteq V$ of the vertices is called a *clique*, if for every $u, v \in X$, (u, v) is an edge.

Once again, there is a natural optimisation problem of finding the largest clique in a graph, but we will consider a decision problem, which we cal CLIQUE.

> The set of pairs (G, K), where G is a graph, and K is an integer, such that G contains a clique with K or more vertices.

As with IND, it is easy to see that CLIQUE is in NP. There is an algorithm which, on input (G, K) guesses a subset X of the vertices of G containing K elements, and then verifies that X forms a clique. To see that CLIQUE is NP-complete, it suffices to prove that IND $\leq_P$ CLIQUE. This is easily seen by the reduction that maps a pair (G, K) to the pair $(\bar{G}, K)$, where $\bar{G}$ is the *complement graph* of G. That is, $\bar{G}$ has the same set of vertices as G, and a pair (x, y) is an edge of $\bar{G}$ if, and only if, it is not an edge of G. Clearly then, any independent set of G is a clique in $\bar{G}$, and conversely any clique in $\bar{G}$ is an independent set of G, so G contains an independent set with K elements if, and only if, $\bar{G}$ contains a clique with K elements. The reduction can, quite obviously, be carried out by a polynomial time algorithm.

Graph Colourability If we are given a graph $G = (V, E)$, that is a set of vertices V along with a set of edges E, we call a *colouring* of G an assignment of colours to the vertices of G such that no edge connects two vertices of the same colour. We say that G is k-colourable if there is a colouring of G which uses no more than k colours. More formally, we say that G is *k-colourable*, if there is a function

$$\chi : V \to \{1, \ldots, k\}$$

such that, for each $u, v \in V$, if $(u, v) \in E$,

$$\chi(u) \neq \chi(v).$$

This sets up a decision problem for each k. Namely,

> given a graph $G = (V, E)$, is it k-colourable?

The problem 2-Colourability is in P. However, for all $k > 2$, k-colourability is NP-complete. Note that here, unlike in the cases of IND and CLIQUE considered above, the

number k is not part of the input presented for the algorithm. Rather, we are considering k as a number fixed before hand and the input is just a graph. So, we will show that the problem 3-colourability is NP-complete. The problem is clearly in NP, since we can guess a colour (one of a fixed set of three, say `red, blue, green`) for each vertex, and then verify that the colouring is valid by checking for each edge that its endpoints are differently coloured. The checking can be done in polynomial time, so this algorithm establishes that 3-colourability is in NP.

To complete the proof that 3-colourabilityis NP-complete, we construct a reduction from 3SAT to 3-colourability. The reduction maps a Boolean expression ϕ in 3-CNF to a graph G so that G is 3-colourable if, and only if, ϕ is satisfiable. Suppose ϕ has m clauses and n distinct variables. G will have 2 special vertices, which we call a and b, one vertex for each variable x and one for its negation $\bar{x}$. For each x, the vertices a, x and $\bar{x}$ are connected in a triangle. In addition, there is an edge connecting a and b, and for each clause in ϕ, there are five new vertices connected in the pattern shown in Figure 6, with the vertex b, and the vertices corresponding to the three literals l_1, l_2 and l_3 that appear in the clause.

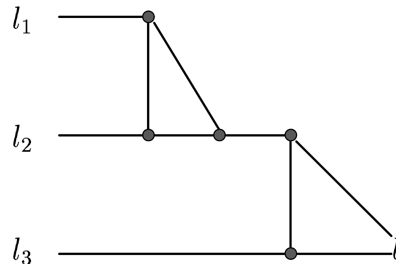

Figure 6: Gadget for the reduction from 3SAT to 3-Colourability

We now need to check that G is 3-colourable if, and only if, ϕ is satisfiable. In any colouring of G, the vertex a must be assigned a colour, let's call it `red`. Since there is an edge between a and b, b must have a different colour, let's call it `blue`. Furthermore, the vertices corresponding to the literals must be coloured `blue` or `green`, as they all have edges to a. Also, of these two colours, each vertex x must be of the opposite colour as its negation $\bar{x}$. We now claim that if there is a valid colouring of the whole graph G, then the truth assignment that makes a variable x `true` if, and only if, it is coloured `blue` (i.e. the same colour as b) is a satisfying truth assignment of ϕ. Conversely, from any satisfying truth assignment T of ϕ, we can obtain a valid 3-colouring of G by colouring `blue` all vertices corresponding to literals that are made `true` by T and colouring `green` all literals that are made `false` by T. To see that this is the case, we only need to check that, in the gadget shown in Figure 6, if b is `blue`, and l_1, l_2, l_3 are all either `blue` or `green`, then there is a valid colouring of the remaining vertices if, and only if, at least one of l_1, l_2 or l_3 is `blue`. This can be checked by examining all possibilities, and is left as an exercise.

Hamiltonian Graphs In a graph G with a set of vertices V, and a set of edges E, a *Hamiltonian cycle* is a path, starting and ending at the same vertex, such that every node in V appears on the cycle *exactly once*. A graph is called *Hamiltonian* if it contains a

Hamiltonian cycle.[1] We define HAM to be the decision problem of determining whether a given graphs is Hamiltonian. As an example, consider the two graphs in Figure 7. The first graph is not Hamiltonian, while the second one is.

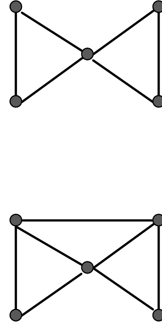

Figure 7: Which graph is Hamiltonian?

It is not too difficult to see that HAM is a problem in NP. An algorithm for solving it can guess a permutation $v_1, v_2, \ldots, v_n$ of the vertices in V, and then verify that, for each i, there is an edge from v_i to v_{i+1} and finally that there is an edge from v_n to v_1. Since the verification step can be done by a polynomial time algorithm, we conclude that HAM is in NP.

We can show that HAM is NP-hard by a reduction from 3SAT. This involves coding a Boolean expression ϕ as a graph G in such a way that every satisfying truth assignment of ϕ corresponds to a Hamiltonian cycle in G. The reduction is much more involved then the ones we have seen for IND and 3-Colourability. The details are left out of the present notes, and can be found in standard textbooks on Complexity Theory.

Travelling Salesman Problem Recall that the *Travelling Salesman Problem* is an optimisation problem specified as, given

- V — a set of vertices.
- $c : V \times V \to \mathbb{N}$ — a cost matrix.

Find an ordering $v_1, \ldots, v_n$ of V for which the total cost:

$$c(v_n, v_1) + \sum_{i=1}^{n-1} c(v_i, v_{i+1})$$

is the smallest possible.

In order to analyse this with the theory of NP-completeness we are building up, we can turn this too into a decision problem, by putting in an explicit target t for the cost of the

[1] The name comes from William Hamilton, a nineteenth century Irish mathematician, who considered whether there were ways of traversing the edges of platonic solids in such a way as to visit each corner exactly once.

tour. That is, the problem TSP is the set of triples $(V, c : V \times V \to \mathbb{N}, t)$, such that there is an ordering $v_1, \ldots, v_n$ of V for which

$$c(v_n, v_1) + \sum_{i=1}^{n-1} c(v_i, v_{i+1}) \leq t.$$

It is fairly easy to see that if there were a polynomial time solution to the optimisation problem, there would also be a polynomial time solution to the decision problem TSP. We could just compute the optimal solution and then check whether its total cost was within the budget t. Thus, a proof that TSP is NP-complete is a strong indication that there is no polynomial time solution to the optimisation problem.

To show that TSP is NP-hard, we note that there is a simple reduction to it from HAM. The reduction maps a graph $G = (V, E)$ to the triple $(V, c : V \times V \to \mathbb{N}, n)$, where n is the number of vertices in V, and the cost matrix c is given by:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ 2 & otherwise \end{cases}$$

Now, since a tour must visit all n cities, it must traverse exactly n edges in this matrix. Thus, if it is within budget, i.e. it has a total cost at most n, then it must only use entries with value 1 in the matrix. In other words, it only traverses edges in the original graph G and, therefore, describes a Hamiltonian cycle in the graph. Conversely, if there is a Hamiltonian cycle in G, there is a way of touring all the cities using only edges of cost 1, and this gives a tour of total cost n.

Sets, Numbers and Scheduling

So far, the NP-complete problems we have looked at either concern the satisfiability of formulas, or properties of graphs. However, NP-completeness is not about formulas and graphs. Literally hundreds of naturally arising problems have been proved NP-complete, in areas involving network design, scheduling, optimisation, data storage and retrieval, artificial intelligence and many others, and new ones are found every day. Such problems arise naturally whenever we have to construct a solution within constraints, and the most effective way appears to be an exhaustive search of an exponential solution space. In this section, we examine three more NP-complete problems, whose significance lies in that they have been used to prove a large number of other problems NP-complete, through reductions. They have been chosen as representative of a large class of problems dealing with sets, numbers and schedules.

3D Matching 3D matching is an extension into 3 dimensions of the well known bipartite matching problem. The latter is defined as the problem of determining, given two sets B and G of equal size, and a set $M \subseteq B \times G$ of pairs, whether there is a *matching*, i.e. a subset $M' \subseteq M$ such that each element of B and each element of G each appear in exactly one pair M' (note that this implies that M' has exactly n elements). The bipartite matching problem is solvable by a polynomial time algorithm.

The problem *3D Matching*, also sometimes called *tripartite matching* is defined by:

Given three disjoint sets X, Y and Z, and a set of triples $M \subseteq X \times Y \times Z$, does M contain a matching?

I.e. is there a subset $M' \subseteq M$, such that each element of X, Y and Z appears in exactly one triple of M'?

This problem is NP-complete. We prove the NP-hardness by a reduction from 3SAT.

We are given a Boolean expression ϕ in 3CNF with m clauses and n variables. For each variable v, we include in the set X, m distinct elements $x_{v1}, \ldots, x_{vm}$ and in Y also m elements $y_{v1}, \ldots, y_{vm}$. We also include in Z $2m$ elements for each variable v. We call these elements $z_{v1}, \ldots, z_{vm}, \bar{z}_{v1}, \ldots, \bar{z}_{vm}$. The triples we include in M are (x_{vi}, y_{vi}, z_{vi}) and $(x_{vi}, y_{v(i+1)}, \bar{z}_{vi})$ for each $i < m$. In the case where $i = m$, we put 1 instead of $i + 1$ in the last triple. The situation for $m = 4$ is displayed in Figure 8, where the triangles represent triples of M.

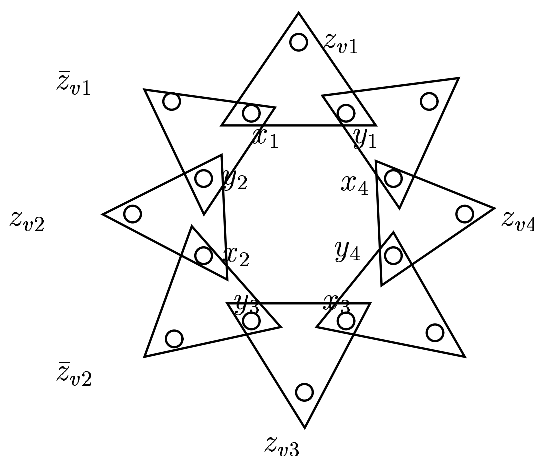

Figure 8: Matching gadget for each variable

In addition, for each clause c of ϕ, we have two elements $x_c \in X$ and $y_c \in Y$. These elements are additional to the m elements for each variable mentioned earlier. If, for some variable v, the literal v occurs in clause c, we include the triple

$$(x_c, y_c, z_{vc})$$

in M, while if the literal $\neg v$ occurs in c, we include the triple

$$(x_c, y_c, \bar{z}_{vc})$$

in M.

Note that, so far, the sets X and Y each contain $mn + m$ elements (m elements x_{vi} for each variable v, and one element x_c for each clause c), and the set Z contains $2mn$ elements, $2m$ for each variable. To get a valid instance of 3DM we need three sets of equal size. We achieve this by adding $m(n-1)$ additional elements to X and Y. These are *dummy* elements, in the sense that we don't want them to constrain possibly matchings in any way, so for each dummy x and y, and *every* element of z, we include the triple (x, y, z) in M. The result is

that if we can find a matching for all elements except the dummies, we can find a matching for the whole set.

It remains to show that there is a matching for the instance we have constructed if, and only if, the expression ϕ is satisfiable. Note that, for any variable v, the element x_{v1} of X appears in only two triples: (x_{v1}, y_{v1}, z_{v1}) and $(x_{v1}, y_{v2}, \bar{z}_{v1})$. A successful matching must include exactly one of these. Moreover, once we have made the choice, this constrains the choice for all other x_{vi}, as can be seen from Figure 8. In fact, there are only two ways that all the x_{vi} can be matched, for a particular v. We can either use all the z_{vi} or we can use all the $\bar{z}_{vi}$. We can think of this choice as the two possible truth values that the variable v can take.

Consider any truth assignment T satisfying the expression ϕ. For each variable v that is set to `true` by T, we select all triples of the form $(x_{vi}, y_{v(i+1)}, \bar{z}_{vi})$ for our matching, and for each v that is set to false, we select the triples $(x_{vi}, y_{v(i)}, z_{vi})$. The result is that, if v is `true`, the elements z_{vi} are available to satisfy x_c and y_c for clauses c in which v appears as a positive literal. Similarly, if v is `false`, the elements $\bar{z}_{vi}$ are available to satisfy those clauses where $\neg v$ appears as a literal. Thus, from a satisfying truth assignment, we obtain a matching. Conversely, we can argue that any matching yields a satisfying truth assignment. The details of the argument are left as an exercise.

Set Covering A further two well-known NP-complete problems are established by straightforward reductions from 3DM. The first is *Exact Cover by 3-Sets* which is defined by:

> Given a set U with $3n$ elements, and a collection $S = \{S_1, \ldots, S_m\}$ of three-element subsets of U, is there a sub collection containing exactly n of these sets whose union is all of U?

The straightforward reduction maps an instance (X, Y, Z, M) of 3DM to the pair (U, S), where $U = X \cup Y \cup Z$, and S consists of all the three element sets $\{x, y, z\}$, where $(x, y, z) \in M$.

A more general problem is *Set Covering*, which is defined by:

> Given a set U, a collection of $S = \{S_1, \ldots, S_m\}$ subsets of U and an integer budget B, is there a collection of B sets in S whose union is U?[2]

The reduction from *Exact Cover by 3-Sets* to *Set Covering* maps a pair (U, S) to the triple (U, S, n), where $3n$ is the number of elements in U.

Knapsack Knapsack is one of the most famous NP-complete problems because it is a natural generalisation of many scheduling and optimisation problems, and through a variety of reductions has been used to show many such problems NP-hard. While the optimisation problems we have seen so far all involve attempting to either minimize some measure of cost or maximize some quantitative benefit. Many optimisation problems arising in practice, however, involve tradeoffs between cost and benefit. Knapsack captures this intuition by

[2] The use of an integer budget B in the definition of the problem is a clear indication that this is the decision version of a natural optimisation problem.

involving both a maximisation and a minimisation element. Formally, the problem is defined by:

> We are given n items, each with a positive integer value v_i and weight w_i. We are also given a maximum total weight W, and a minimum total value V.
>
> Can we select a subset of the items whose total weight does not exceed W, and whose total value exceeds V?

To prove that Knapsack is NP-complete, we construct a reduction from the problem of *Exact Cover by 3 Sets* (we omit the argument that Knapsack is in NP, which is easy).

We are given a set $U = \{1, \ldots, 3n\}$, and a collection of 3-element subsets of U, $S = \{S_1, \ldots, S_m\}$. We map this to an instance of KNAPSACK with m elements each corresponding to one of the S_i, and having weight and value

$$\Sigma_{j \in S_i} m^j$$

and set the target weight and value both to

$$\Sigma_{j=0}^{3n-1} m^j.$$

The idea is that we represent subsets of U as strings of 0s and 1s of length $3n$. We treat these strings as representations of numbers, not in base 2, but in base m. This guarantees that when we add numbers corresponding to the sets S_i, we never get carry from one place to the next, as there are only m sets. Thus, the only way we can achieve the target number (represented by 1s in all positions), is if the union of the sets we have chosen is all of U, and no element of U is represented more than once—as this would result in a value greater than 1 in some place. It follows that the instance of Knapsack has a solution if, and only if, the original pair (U, S) has an exact cover by 3-sets.

Indeed, the reduction we have constructed produces instances of Knapsack of a rather special kind. All weights and values are equal, and the target weight is the same as the target value. While this does prove that the general Knapsack problem is NP-complete, it also establishes the NP-completeness of a restriction of the problem to instances where weights and values are always equal. This problem has a particularly simple formulation:

> Given a collection of numbers $v_1, \ldots, v_n$ and a target t, is there a subcollection of the numbers which adds up exactly to t.

This simple looking problem turns out to be NP-complete.

Scheduling The problem Knapsack has been used to prove a wide variety of scheduling problems NP-complete. A few examples are given here as illustration.

Timetable Design

> Given a set H of *work periods*, a set W of *workers* each with an associated subset of H (available periods), a set T of *tasks* and an assignment $r : W \times T \rightarrow \mathbb{N}$ of *required work*, is there a mapping $f : W \times T \times H \rightarrow \{0, 1\}$ which completes all tasks?

That is, for any $w \in W$ and $h \in H$ there is at most one $t \in T$ for which $f(w, h, t) = 1$, and there is one only if w is available at h. Moreover, for each $t \in T$, and each $w \in W$, there are at least $r(w, t)$ distinct h for which $f(w, h, t) = 1$.

Sequencing with Deadlines

Given a set T of *tasks* and for each task a *length* $l \in \mathbb{N}$, a release time $r \in \mathbb{N}$ and a deadline $d \in \mathbb{N}$, is there a work schedule which completes each task between its release time and its deadline?

Here, a schedule is an assignment to each task $t \in T$ a start time $s(t)$, such that $s \geq r(t)$, $d(t) \geq s(t) + l(t)$, and such that for any other t', $s(t') \geq s(t) + l(t)$.[3]

Finally, a multi-processor version of this is:
Job Scheduling

Given a set T of *tasks*, a number $m \in \mathbb{N}$ of processors a length $l \in \mathbb{N}$ for each task, and an overall deadline $D \in \mathbb{N}$, is there a multi-processor schedule which completes all tasks by the deadline?

Responses to NP-completeness

Having seen a number of NP-complete problems, and some varied proofs of their NP-completeness, we have acquired at least some ability to recognise new NP-complete problems when we see them. One question that arises is, what are we to do when we are confronted with one. Surely, the analysis that shows that a problem is NP-complete is not the end of the matter. We are still required to find a solution of some sort. There are a variety of possible responses.

It might be that we are trying to solve a single instance—we might have to construct a railway timetable, or an exam timetable, once only. The results of asymptotic complexity do not, of course, tell us much about single instances. An algorithm whose running time is exponential may run in reasonable time on instances up to some small size. In practice, though, an algorithm that is exponential in running time will become completely impractical on even reasonably small instances. One would certainly now want to use it on something as large as a railway timetable. The running time may, after all, double with the addition of each additional station

In general, if we are using a general purpose algorithm for the problem (rather than exploiting features of the particular single instance), then *scalability* is important. A program tested on small instances may completely seize up when confronted with an industrial scale example.

Thus, in order to get a solution that is scalable, one has to adopt a different approach to a brute force search. Often, this involves a closer examination of the problem at hand. Is it really an NP-complete problem in its full generality, or is it a *restriction* to some special class of instances. For instance, in many applications, the graphs that arise are necessarily planar. Not all the NP-complete graph problems we have considered remain NP-complete

[3]Note that this is non-preemptive scheduling.

when restricted to planar graphs. The CLIQUE problem is a case in point. No planar graph can have a clique of five or more elements. Thus, the problem of finding the largest clique in a graph reduces to checking whether the graph contains a clique of four elements, something that can be done by a polynomial time algorithm. While HAM and 3-Colourability are known to be NP-complete, even when restricted to planar graphs, 4-Colourability is trivial on planar graphs, since all planar graphs are 4-colourable.

Another approach often adopted in dealing with optimisation problems corresponding to NP-complete problems is to settle for approximate rather than exact solutions. Often, such problems admit polynomial time *approximation algorithms*, which are not guaranteed to find the best solution, but will produce a solution which is known to be within a known factor of the optimal. This is particularly useful in applications where we need to be able to give performance guarantees.

A final point is that, if we are using an algorithm with potentially exponential worst-case performance, using a backtracking search strategy, it is important to identify good *heuristics* for constraining the search. These heuristics will often arise from known limitations of the actual application area, and it is difficult to devise general purpose rules for them. However, good heuristics can dramatically cut down the search space and home in on a solution for a typical instance of the problem, while still requiring an exponential search in the worst-case.