Chapter 1
Introduction

In this introductory chapter we start with a preliminary part and present then two classical exact algorithms breaking the triviality barrier. The first one, from the nineteen sixties, is the dynamic programming algorithm of Bellman, Held and Karp to solve the TRAVELLING SALESMAN problem [16, 17, 111]. The second is a branching algorithm to compute a maximum independent set of a graph. The main idea of this algorithm can be traced back to the work of Miller and Muller [155] and Moon and Moser [161] from the nineteen sixties.

The history of research on exact algorithms for these two NP-hard problems is contrasting. Starting with the algorithm of Tarjan and Trojanowski [213] from 1977 there was a chain of dramatic improvements in terms of the running time of an algorithm for the MAXIMUM INDEPENDENT SET problem. For the TRAVELLING SALESMAN problem, despite many attempts, no improvement on the running time of the Bellman-Held-Karp's algorithm was achieved so far.

1.1 Preliminaries

$\mathcal{O}^*$ notation. The classical big-O notation is defined as follows. For functions $f(n)$ and $g(n)$ we write $f = \mathcal{O}(g)$ if there are positive numbers n_0 and c such that for every $n > n_0$, $f(n) < c \cdot g(n)$. In this book we use a modified big-O notation that suppresses all polynomially bounded factors. For functions f and g we write $f(n) = \mathcal{O}^*(g(n))$ if $f(n) = \mathcal{O}(g(n)poly(n))$, where $poly(n)$ is a polynomial. For example, for $f(n) = 2^n n^2$ and $g(n) = 2^n$, $f(n) = \mathcal{O}^*(g(n))$. This modification of the classical big-O notation can be justified by the exponential growth of $f(n)$. For instance, the running time $(\sqrt{2})^n poly(n)$ is sandwiched between running times 1.4142135^n and 1.4142136^n for every polynomial $poly(n)$ and sufficiently large n. In many chapters of this book when estimating the running time of algorithms, we have exponential functions where the base of the exponent is some real number. Very often we round the base of the exponent up to the fourth digit after the decimal point. For example, for running time $\mathcal{O}((\sqrt{2})^n)$, we have $\sqrt{2} = 1.414213562...$, and $(\sqrt{2})^n poly(n) =$

$\mathcal{O}(1.4143^n)$. Hence when we round reals in the base of the exponent, we use the classical big-O notation. We also write $f = \Omega(g)$, which means that $g = \mathcal{O}(f)$, and $f = \Theta(g)$, which means that $f = \Omega(g)$ and $f = \mathcal{O}(g)$.

Measuring quality of exact algorithms. The common agreement in polynomial time algorithms is that the running time of an algorithm is estimated by a function either of the input length or of the input "size". The input length can be defined as the number of bits in any "reasonable" encoding of the input over a finite alphabet; but the notion of input size is problem dependent. Usually every time we speak about the input size, we have to specify what we mean by that. Let us emphasize that for most natural problems the length of the input is not exactly the same as what we mean by its "size". For example, for a graph G on n vertices and m edges, we usually think of the size of G as $\Theta(n+m)$, while the length (or the number of bits) in any reasonable encoding over a finite alphabet is $\Theta(n+m\log n)$. Similarly for a CNF Boolean formula F with n variables and m clauses, the size of F is $\Theta(n+m)$ and the input length is $\Theta(n+m\log m)$.

So what is the appropriate input "size" for exponential time algorithms? For example for an optimization problem on graphs, the input "size" can be the number of vertices, the number of edges or the input length. In most parts of this book we follow the more or less established tradition that

- Optimization problems on graphs are analyzed in terms of the number of vertices;
- Problems on sets are analyzed in terms of the number of elements;
- Problems on Boolean formulas are analyzed in terms of the number of variables.

An argument for such choices of the "size" is that with such parameterization it is often possible to measure the improvement over the trivial brute-force search algorithm. Every search version of the problem L in NP can be formulated in the following form:

Given x, find y so that $|y| \le m(x)$ and $R(x,y)$ (if such y exists).

Here x is an instance of L, $|y|$ is the length (the number of bits in the binary representation) of certificate y, $R(x,y)$ is a polynomial time decidable relation that verifies the certificate y for instance x, and $m(x)$ is a polynomial time computable and polynomially bounded complexity parameter that bounds the length of the certificate y. Thus problem L can be solved by enumerating all possible certificates y of length at most $m(x)$ and checking for each certificate in polynomial time if $R(x,y)$. Therefore, the running time of the brute-force search algorithm is up to a polynomial multiplicative factor proportional to the number of all possible certificates of length at most $m(x)$, which is $\mathcal{O}^*(2^{m(x)})$.

Let us give some examples.

- *Subset problems.* In a subset problem every feasible solution can be specified as a subset of an underlying ground set. If the cardinality of the ground set is n, then every subset S of the ground set can be encoded by a binary string of length n. The ith element of the string is 1 if and only if the ith element of the instance x is in S. In this case $m(x) = n$ and the brute-force search can be done in time $\mathcal{O}^*(2^n)$. For instance, a truth assignment in the SATISFIABILITY problem corresponds to selecting a subset of TRUE variables. A candidate solution in this case is the subset of variables, and the size of each subset does not exceed the number of variables, hence the length of the certificate does not exceed n. Thus the brute-force search enumerating all possible subsets of variables and checking (in polynomial time) whether the selected assignment satisfies the formula takes $\mathcal{O}^*(2^n)$ steps. In the MAXIMUM INDEPENDENT SET problem, every subset of the vertex set is a solution candidate of size at most n, where n is the number of vertices of the graph. Again, the brute-force search for MAXIMUM INDEPENDENT SET takes $\mathcal{O}^*(2^n)$ steps.

- *Permutation problems.* In a permutation problem every feasible solution can be specified as a total ordering of an underlying ground set. For instance, in the TRAVELLING SALESMAN problem, every tour corresponds to a permutation of the cities. For an instance of the problem with n cities, possible candidate solutions are ordered sets of n cities. The size of the candidate solution is n and the number of different ordered sets of size n is $n!$. In this case $m(x) = \log_2 n!$ and the trivial algorithm runs in time $\mathcal{O}^*(n!)$.
- *Partition problems.* In a partition problem, every feasible solution can be specified as a partition of an underlying ground set. An example of such a problem is the GRAPH COLORING problem, where the goal is to partition the vertex set of an n-vertex graph into color classes. In this case $m(x) = \log_2 n^n$ and the brute-force algorithm runs in $\mathcal{O}^*(n^n) = \mathcal{O}^*(2^{n \log n})$ time.

Intuitively, such a classification of the problems according to the number of candidate solutions creates a complexity hierarchy of problems, where subset problems are "easier" than permutation problems, and permutation problems are "easier" than partition problems. However, we do not have any evidences that such a hierarchy exists; moreover there are permutation problems solvable in time $\mathcal{O}^*((2-\varepsilon)^n)$ for some $\varepsilon > 0$. There are also some subset problems for which we do not know anything better than brute-force search. We also should say that sometimes such classification is ambiguous. For example, is the HAMILTONIAN CYCLE problem a permutation problem for vertices or a subset problem for edges? One can argue that on graphs, where the number of edges m is less than $\log_2 n!$, the algorithm trying all possible edge subsets in time $\mathcal{O}^*(2^m)$ is faster than $\mathcal{O}^*(n!)$, and in these cases we have to specify what we mean by the brute-force algorithm. Fortunately, such ambiguities do not occur often.

Parameterized complexity. The area of exact exponential algorithms is not the only one dealing with exact solutions of hard problems. The parameterized complexity theory introduced by Downey and Fellows [66] is a general framework for a refined analysis of hard algorithmic problems. Parameterized complexity measures complexity not only in terms of input length but also in terms of a parameter which is a numerical value not necessarily dependent on the input length. Many parameterized algorithmic techniques evolved accompanied by a powerful complexity theory. We refer to recent monographs of Flum and Grohe [78] and Niedermeier [164] for overviews of parameterized complexity. Roughly speaking, parameterized complexity seeks the possibility of obtaining algorithms whose running time can be bounded by a polynomial function of the input length and, usually, an exponential function of the parameter. Thus most of the exact exponential algorithms studied in this book can be treated as parameterized algorithms, where the parameter can be the number of vertices in a graph, the number of variables in a formula, etc. However, such a parameterization does not make much sense from the point of view of parameterized complexity, where the fundamental assumption is that the parameter is independent of the input size. In particular, it is unclear whether the powerful tools from parameterized complexity can be used in this case. On the other hand, there are many similarities between the two areas, in particular some of the basic techniques like branching, dynamic programming, iterative compression and inclusion-exclusion are used in both areas. There are also very nice connections between subexponential complexity and parameterized complexity.

1.2 Dynamic Programming for TSP

Travelling Salesman Problem. In the TRAVELLING SALESMAN problem (TSP), we are given a set of distinct cities $\{c_1, c_2, \ldots, c_n\}$ and for each pair $c_i \neq c_j$ the distance between c_i and c_j, denoted by $d(c_i, c_j)$. The task is to construct a tour of the travelling salesman of minimum total length which visits all the cities and returns to the starting point. In other words, the task is to find a permutation π of $\{1, 2, \ldots, n\}$, such that the following sum is minimized

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}).$$

How to find a tour of minimum length? The easy way is to generate all possible solutions. This requires us to verify all permutations of the cities and the number of all permutations is $n!$. Thus a naive approach here requires at least $n!$ steps. Using dynamic programming one obtains a much faster algorithm.

The dynamic programming algorithm for TSP computes for every pair (S, c_i), where S is a nonempty subset of $\{c_2, c_3, \ldots, c_n\}$ and $c_i \in S$, the value $OPT[S, c_i]$ which is the minimum length of a tour which starts in c_1, visits all cities from S and ends in c_i. We compute the values $OPT[S, c_i]$ in order of increasing cardinality of S. The computation of $OPT[S, c_i]$ in the case S contains only one city is trivial, because in this case, $OPT[S, c_i] = d(c_1, c_i)$. For the case $|S| > 1$, the value of $OPT[S, c_i]$ can be expressed in terms of subsets of S:

$$OPT[S, c_i] = \min\{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i) \colon c_j \in S \setminus \{c_i\}\}. \qquad (1.1)$$

Indeed, if in some optimal tour in S terminating in c_i, the city c_j immediately precedes c_i, then

$$OPT[S,c_i] = OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i).$$

Thus taking the minimum over all cities that can precede c_i, we obtain (1.1). Finally, the value OPT of the optimal solution is the minimum of

$$OPT[\{c_2, c_3, \dots, c_n\}, c_i] + d(c_i, c_1),$$

where the minimum is taken over all indices $i \in \{2, 3, \dots, n\}$.

Such a recurrence can be transformed in a dynamic programming algorithm by solving subproblems in increasing sizes, which here is the number of cities in S. The corresponding algorithm `tsp` is given in Fig. 1.1.

```
Algorithm tsp($\{c_1, c_2, \dots c_n\}, d$).
Input: Set of cities $\{c_1, c_2, \dots, c_n\}$ and for each pair of cities $c_i, c_j$ the distance $d(c_i, c_j)$.
Output: The minimum length of a tour.

  for $i = 2$ to $n$ do
      $OPT[c_i, c_i] = d(c_1, c_i)$
  for $j = 2$ to $n-1$ do
      forall $S \subseteq \{2, 3, \dots, n\}$ with $|S| = j$ do
          $OPT[S, c_i] = \min\{OPT[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \colon c_k \in S \setminus \{c_i\}\}$
  return $\min\{OPT[\{c_2, c_3, \dots, c_n\}, c_i] + d(c_i, c_1) : i \in \{2, 3, \dots, n\}\}$
```

Fig. 1.1 Algorithm `tsp` for the TRAVELLING SALESMAN problem

Before analyzing the running time of the dynamic programming algorithm let us give a word of caution. Very often in the literature the running time of algorithms is expressed in terms of basic computer primitives like arithmetic (add, subtract, multiply, comparing, floor, etc.), data movement (load, store, copy, etc.), and control (branching, subroutine call, etc.) operations. For example, in the unit-cost random-access machine (RAM) model of computation, each of such steps takes constant time. The unit-cost RAM model is the most common model appearing in the literature on algorithms. In this book we also adapt the unit-cost RAM model and treat these primitive operations as single computer steps. However in some parts of the book dealing with computations with huge numbers such simplifying assumptions would be too inaccurate.

The reason is that in all known realistic computational models arithmetic operations with two b-bit numbers require time $\Omega(b)$, which brings us to the log-cost RAM model. For even more realistic models one has to assume that two b-bit integers can be added, subtracted, and compared in $\mathcal{O}(b)$ time, and multiplied in $\mathcal{O}(b \log b \log\log b)$ time. But this level of precision is not required for most of the results discussed in this book. Because of the $\mathcal{O}^*$-notation, we can neglect the difference between log-cost and unit-cost RAM for most of the algorithms presented in this book. Therefore, normally we do not mention the model used to analyze running times of algorithms (assuming unit-cost RAM model), and specify it only when the difference between computational models becomes important.

Let us come back to TSP. The amount of steps required to compute (1.1) for a fixed set S of size k and all vertices $c_i \in S$ is $\mathcal{O}(k^2)$. The algorithm computes (1.1) for every subset S of cities, and thus takes time $\sum_{k=1}^{n-1} \mathcal{O}(\binom{n}{k})$. Therefore, the total time to compute OPT is

$$\sum_{k=1}^{n-1} \mathcal{O}(\binom{n}{k} k^2) = \mathcal{O}(n^2 2^n).$$

The improvement from $\mathcal{O}(n!n)$ in the trivial enumeration algorithm to $\mathcal{O}^*(2^n)$ in the dynamic programming algorithm is quite significant.

For the analyses of the TSP algorithm it is also important to specify which model is used. Let W be the maximum distance between the cities. The running time of the algorithm for the unit-cost RAM model is $\mathcal{O}^*(2^n)$. However, during the algorithm we have to operate with $\mathcal{O}(\log nW)$-bit numbers. By making use of more accurate log-cost RAM model, we estimate the running time of the algorithm as $2^n \log W n^{\mathcal{O}(1)}$. Since W can be arbitrarily large, $2^n \log W n^{\mathcal{O}(1)}$ is not in $\mathcal{O}^*(2^n)$.

Finally, once all values $OPT[S, c_i]$ have been computed, we can also construct an optimal tour (or a permutation π) by making use of the following observation: A permutation π, with $\pi(c_1) = c_1$, is optimal if and only if

$$OPT = OPT[\{c_{\pi(2)}, c_{\pi(3)}, \dots, c_{\pi(n)}\}, c_{\pi(n)}] + d(c_{\pi(n)}, c_1),$$

and for $k \in \{2, 3, \dots, n-1\}$,

$$\begin{aligned} OPT[\{c_{\pi(2)}, \dots, c_{\pi(k+1)}\}, c_{\pi(k+1)}] = OPT[\{c_{\pi(2)}, \dots, c_{\pi(k)}\}, c_{\pi(k)}] \\ + d(c_{\pi(k)}, c_{\pi(k+1)}). \end{aligned}$$

A dynamic programming algorithm computing the optimal value of the solution of a problem can typically also produce an optimal solution of the problem. This is done by adding suitable pointers such that a simple backtracing starting at an optimal value constructs an optimal solution without increasing the running time.

One of the main drawbacks of dynamic programming algorithms is that they need a lot of space. During the execution of the dynamic programming algorithm above described, for each $i \in \{2, 3, \dots, n\}$ and $j \in \{1, 2, \dots, n-1\}$, we have to keep all the values $OPT[S, c_i]$ for all sets of size j and $j+1$. Hence the space needed is $\Omega(2^n)$, which means that not only the running time but also the space used by the algorithm is exponential.

Dynamic Programming is one of the major techniques to design and analyse exact exponential time algorithms. Chapter 3 is dedicated to Dynamic Programming. The relation of exponential space and polynomial space is studied in Chap. 10.

1.3 A Branching Algorithm for Independent Set

A fundamental and powerful technique to design fast exponential time algorithms is Branch & Reduce. It actually comes with many different names: branching algorithm, search tree algorithm, backtracking algorithm, Davis-Putnam type algorithm etc. We shall introduce some of the underlying ideas of the Branch & Reduce paradigm by means of a simple example.

Maximum Independent Set. In the MAXIMUM INDEPENDENT SET problem (`MIS`), we are given an undirected graph $G = (V,E)$. The task is to find an independent set $I \subseteq V$, i.e. any pair of vertices of I is non-adjacent, of maximum cardinality. For readers unfamiliar with terms from Graph Theory, we provide the most fundamental graph notions in Appendix .

A trivial algorithm for this problem would be to try all possible vertex subsets of G, and for each subset to check (which can be easily done in polynomial time), whether this subset is an independent set. At the end this algorithm outputs the size of the maximum independent set or a maximum independent set found. Since the number of vertex subsets in a graph on n vertices is 2^n, the naive approach here requires time $\Omega(2^n)$.

Here we present a simple branching algorithm for `MIS` to introduce some of the major ideas. The algorithm is based on the following observations. If a vertex v is in an independent set I, then none of its neighbors can be in I. On the other hand, if I is a maximum (and thus maximal) independent set, and thus if v is not in I then at least one of its neighbors is in I. This is because otherwise $I \cup \{v\}$ would be an independent set, which contradicts the maximality of I. Thus for every vertex v and every maximal independent set I, there is a vertex y from the closed neighborhood $N[v]$ of v, which is the set consisting of v and vertices adjacent to v, such that y is in I, and no other vertex from $N[y]$ is in I. Therefore to solve the problem on G, we solve problems on different reduced instances, and then pick up the best of the obtained solutions. We will refer to this process as branching.

The algorithm in Fig. 1.2 exploits this idea. We pick a vertex of minimum degree and for each vertex from its closed neighborhood we consider a subproblem, where we assume that this vertex belongs to a maximum independent set.

```
Algorithm mis1(G).
Input: Graph $G = (V,E)$.
Output: The maximum cardinality of an independent set of $G$.

  if $|V| = 0$ then
    return 0
  choose a vertex $v$ of minimum degree in $G$
  return $1 + \max\{\texttt{mis1}(G \setminus N[y]) : y \in N[v]\}$
```

Fig. 1.2 Algorithm `mis1` for MAXIMUM INDEPENDENT SET

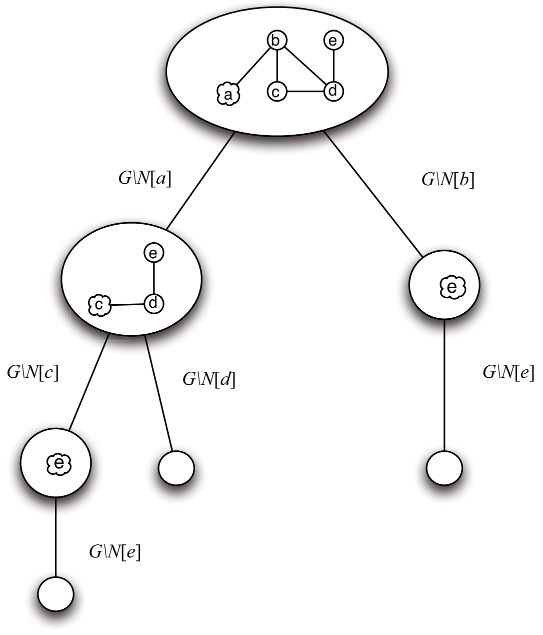

Fig. 1.3 Example of a minimum degree branching algorithm. We branch on vertex a. Then in one subproblem we branch on c, and in the other on e, etc.

The correctness of branching algorithms is usually easy to verify. The algorithm consists of a single branching rule and its correctness follows from the discussions above.

As an example, let us consider the performance of algorithm `mis1` on the graph G of Fig. 1.3. At the beginning the minimum vertex degree is 1, so we select one of the vertices of degree 1, say a. We branch with 2 subproblems, the left branch corresponding to $G \setminus N[a]$ and the right branch to $G \setminus N[b]$. For the right branch there is a unique choice and after branching on e we obtain an empty graph and do not branch anymore. The value the algorithm outputs for this branch is 2 and this corresponds to the maximal independent set $\{b,e\}$. For the left branch we pick a vertex of minimum degree (again 1), say c, and branch again with 2 subproblems. The maximal independent sets found in the left branch are $\{e,c,a\}$ and $\{d,a\}$ and the algorithm reports that the size of a maximum independent set is 3. Let us observe the interesting fact that every maximal independent set can be constructed by following a path from some leaf to the root of the search tree.

Analysing the worst case running time of a branching algorithm can be nontrivial. The main idea is that such an algorithm is recursive and that each execution of it can be seen as a search tree T, where a subproblem, here $G' = G \setminus V'$, is assigned to a node of T. Furthermore when branching from a subproblem assigned to a node of T then any subproblem obtained is assigned to a child of this node. Thus a solution in a node can be obtained from its descendant branches, and this is why we use the term branching for this type of algorithms and call the general approach *Branch & Reduce*. The running time spent by the algorithm on computations corresponding to each node is polynomial—we construct a new graph by removing some vertices, and up to a polynomial multiplicative factor the running time of the algorithm is upper bounded by the number of nodes in the search tree T. Thus to determine the worst case running time of the algorithm, we have to determine the largest number $T(n)$ of nodes in a search tree obtained by any execution of the algorithm on an input graph G having n vertices. To compute $T(n)$ of a branching algorithm one usually relies on the help of linear recurrences. We will discuss in more details how to analyze the running time of such algorithms in Chap. 2.

Let us consider the branching algorithm `mis1` for `MIS` of Fig. 1.2. Let G be the input graph of a subproblem. Suppose the algorithm branches on a vertex v of degree $d(v)$ in G. Let $v_1, v_2, \dots, v_{d(v)}$ be the neighbors of v in G. Thus for solving the subproblem G the algorithm recursively solves the subproblems $G \setminus N[v]$, $G \setminus N[v_1], \dots, G \setminus N[v_{d(v)}]$ and we obtain the recurrence

$$T(n) \le 1 + T(n - d(v) - 1) + \sum_{i=1}^{d(v)} T(n - d(v_i) - 1).$$

Since in step 3 the algorithm chooses a vertex v of minimum degree, we have that for all $i \in \{1, 2, \dots, d(v)\}$,

$$\begin{aligned} d(v) &\le d(v_i), \\ n - d(v_i) - 1 &\le n - d(v) - 1 \end{aligned}$$

and, by the monotonicity of $T(n)$,

$$T(n - d(v_i) - 1) \le T(n - d(v) - 1).$$

We also assume that $T(0) = 1$. Consequently,

$$\begin{aligned} T(n) &\le 1 + T(n - d(v) - 1) + \sum_{i=1}^{d(v)} T(n - d(v) - 1) \\ &\le 1 + (d(v) + 1) \cdot T(n - d(v) - 1). \end{aligned}$$

By putting $s = d(v) + 1$, we obtain

$$\begin{aligned} T(n) &\le 1 + s \cdot T(n - s) \le 1 + s + s^2 + \cdots + s^{n/s} \\ &= \frac{1 - s^{n/s+1}}{1 - s} = \mathcal{O}^*(s^{n/s}). \end{aligned}$$

For $s > 0$, the function $f(s) = s^{1/s}$ has its maximum value for $s = e$ and for integer s the maximum value of $f(s) = s^{1/s}$ is when $s = 3$.

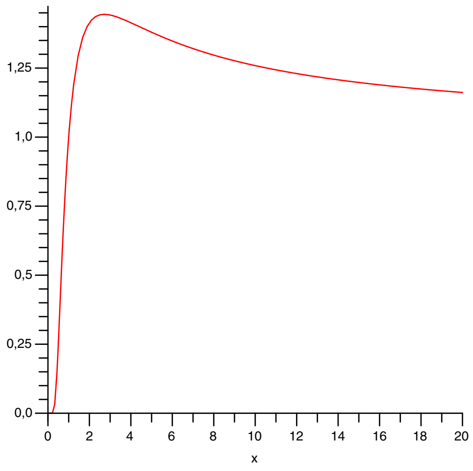

Fig. 1.4 $f(s) = s^{1/s}$

Thus we obtain

$$T(n) = \mathcal{O}^*(3^{n/3}),$$

and hence the running time of the branching algorithm is $\mathcal{O}^*(3^{n/3})$.

Branch & Reduce is one of the fundamental paradigms in the design and analysis of exact exponential time algorithms. We provide a more detailed study of this approach in Chaps 2 and 6.

Notes

As a mathematical problem, `TSP` was first formulated in 1930 but the history of the problem dates back in the 1800s when Hamilton studied related problems. See [115] for the history of the problem. The dynamic programming algorithm for `TSP` is due to Bellman [16, 17] and to Held and Karp [111]. Surprisingly, for almost 50 years of developments in Algorithms, the running time $\mathcal{O}^*(2^n)$ of an exact algorithm for `TSP` has not been improved. Another interesting question is on the space requirements of the algorithm. If the maximum distance between two cities is W, then by making use of inclusion-exclusion (we discuss this technique in Chap. 4), it is possible to solve the problem in time $\mathcal{O}^*(W2^n)$ and space $\mathcal{O}^*(W)$ [127]. Recently, Lokshtanov and Nederlof used the discrete Fourier transform to solve `TSP` in time $\mathcal{O}^*(W2^n)$ and polynomial, i.e. $n^{\mathcal{O}(1)} \cdot (\log W)^{\mathcal{O}(1)}$ space [154]. See also Chap. 10 for a $\mathcal{O}^*(4^n n^{\mathcal{O}(\log n)})$ time and polynomial space algorithm.

For discussions on computational models we refer to the book of Cormen et al. [52]; see also [61]. The classical algorithm of Schönhage and Strassen from 1971 multiplies two b-bit integers in time $\mathcal{O}(b \log b \log\log b)$ [198]. Recently Fürer improved the running time to $b \log b\, 2^{\mathcal{O}(\log^* b)}$, where $\log^* b$ is the iterated logarithm of b, i.e. the number of times the logarithm function must be iteratively applied before the result is at most 1 [98].

`MIS` is one of the benchmark problems in exact algorithms. From an exact point of view `MIS` is equivalent to the problems MAXIMUM CLIQUE and MINIMUM VERTEX COVER. It is easy to modify the branching algorithm `mis1` so that it not only finds one maximum independent set but outputs all maximal independent sets of the input graph in time $\mathcal{O}^*(3^{n/3})$. The idea of algorithm `mis1` (in a different form) goes back to the works of Miller and Muller [155] from 1960 and to Moon and Moser [161] from 1965 who independently obtained the following combinatorial bound on the maximum number of maximal independent sets.

Theorem 1.1. *The number of maximal independent sets in a graph on n vertices is at most*

$$\begin{cases} 3^{n/3} & \text{if } n \equiv 0 \pmod 3, \\ 4 \cdot 3^{(n-4)/3} & \text{if } n \equiv 1 \pmod 3, \\ 2 \cdot 3^{(n-2)/3} & \text{if } n \equiv 2 \pmod 3. \end{cases}$$

Moreover, all bounds of Theorem 1.1 are tight and are achievable on graphs consisting of $n/3$ disjoint copies of K_3s; one K_4 or two K_2s and $(n-4)/3$ K_3s; one K_2 and $(n-2)/3$ copies of K_3s. A generalization of this theorem for induced regular subgraphs is discussed in [107].

While the bound $3^{n/3}$ on the number of maximal independent sets is tight, the running time of an algorithm computing a maximum independent set can be strongly improved. The first improvement over $\mathcal{O}^*(3^{n/3})$ was published in 1977 by Tarjan and Trojanowski [213]. It is a Branch & Reduce algorithm of running time $\mathcal{O}^*(2^{n/3}) = \mathcal{O}(1.26^n)$ [213]. In 1986 Jian published an improved algorithm with running time $\mathcal{O}(1.2346^n)$ [125]. In the same year Robson provided an algorithm of running time $\mathcal{O}(1.2278^n)$ [185]. All these three algorithms are Branch & Reduce algorithms, and use polynomial space. In [185] Robson also showed how to speed up Branch & Reduce algorithms using a technique that is now called Memorization (and studied in detail in Chap. 10), and he established an $\mathcal{O}(1.2109^n)$ time algorithm that needs exponential space. Fomin, Grandoni, and Kratsch [85] showed how to solve the problem `MIS` in time $\mathcal{O}(1.2202^n)$ and polynomial space. Kneis, Langer, and Rossmanith in [133] provided a branching algorithm with a computer-aided case analysis to establish a running time of $\mathcal{O}(1.2132^n)$. Very recently Bourgeois, Escoffier, Paschos and van Rooij in [38] improved the best running time of a polynomial space algorithm to compute a maximum independent set to $\mathcal{O}(1.2114^n)$. A significant amount of research has also been devoted to solving the maximum independent set problem on sparse graphs [13, 37, 39, 47, 48, 97, 179].