Appendix. NP-completeness and NP-complete problems under consideration

In this section we briefly introduce the notions of NP-hardness, NP-completeness, and *polynomial reductions*, which allows to prove the NP-hardness of a problem as well as relating NP-hard problems with each other. For the sake of briefness, some notions will be intuitively introduced. For a completely formal description of these concepts, the reader can check e.g. [26, 10]. We also define the NP-complete problems we have considered in our experiments.

A decision problem is a problem whose answer for each input is *yes* or *not* (e.g. is X a perfect square? is there any hamiltonian cycle in graph G? Does program P halts?), and it implicitly defines a set: the set of inputs whose answer for this problem is *yes*. A decision problem is in class P if it can be solved in a polynomial number of steps with respect to the input size (i.e. the size of X, G, and P in previous examples) by a *Deterministic Turing Machine* (DTM). A Deterministic Turing Machine is a computation formalism based on a set of states and transitions, a tape of cells, and a tape head where, at each step, depending on the current state and the symbol at the tape cell pointed by the head, we change the symbol pointed by the head by another, the head is shifted one cell to the right or to the left, and we go to another state. In addition, for each configuration (state, tape, and head location), the next execution step is uniquely determined. A DTM has the same computation capability as any modern programming language (C++, Java, Haskell, Prolog, etc) under the assumption that the amount of available memory is arbitrarily high. Solving a problem by a DTM may take different times as solving it by a Java program (C++ program, etc). For instance, the time complexity of a problem might be in $O(n^k)$ if it is solved by a DTM and in $O(n^{k'})$ with $k' < k$ if it is solved by a Java program. However, DTMs and Java programs depict the same *border* between problems requiring polynomial time and problems requiring exponential time: A problem can be solved in polynomial time by a DTM if and only if it can be so by a Java program.

A decision problem is in class NP if it can be solved by an *Non-Deterministic Turing Machine* (NDTM) in polynomial time. An NDTM is like a DTM though, at each NDTM configuration, more than one transition might be available to continue the execution. If the NDTM can take these non-deterministic choices in such a way that the execution eventually answers yes, then the answer of the NDTM is yes, else it is not. Alternatively, we can imagine a NDTM as a parallel machine where all possible non-deterministic paths are run in parallel: We can do as many `fork` operations as required and execute all resulting processes in parallel in different processors (we have as many processors as processes), though processes cannot communicate with each other (neither by messages nor by shared variables), but they can just emit an *answer*: Again, the execution answer is *yes* iff at least one of these processes says *yes*. Since we answer *yes* only if there exists an execution path where we answer *yes*, we may consider the following alternative view of class NP: A decision problem is in class NP if there exists a polynomial-time *deterministic* TM such that, for each problem input, the DTM accepts (i.e. says *yes* to) an string consisting in the concatenation of the input and *some* polynomial-size sequence of additional symbols iff the answer of the problem for that input is *yes*. This alternative definition of the NP class is linked with the former one, mentioned earlier and based on a NDTM, as follows: If the input is accepted by the NDTM then the sequence of additional symbols tells the deterministic TM *which* execution path of the NDTM reaches acceptance. Let us consider a decision problem of the form "*Is there X such that P(X)?*" (e.g. given a graph G, is there a path p in G which is a hamiltonian cycle?). If we view each execution branch of a NDTM as a branch where a specific candidate solution for the problem is checked (e.g., is path p a hamiltonian cycle in G?), then the sequence of additional symbols of the DTM represents the candidate solution under consideration in the branch (that is, p). So, the answer of the NDTM for some input is *yes* iff there exists a sequence of additional symbols such that the answer of the DTM for the concatenation of the input and the sequence is *yes*.

A decision problem A can be *polynomially reduced* into a decision problem B if there is a polynomial-time function f such that, for any instance X of problem A, the answer of problem A for this instance is *yes* iff the answer of problem B for instance $f(X)$ is *yes*. Intuitively, if A can be polynomially reduced to B then, if there existed a polynomial-time solution of problem B, there would also exist a polynomial-time solution for problem A: We could solve A by transforming X into $f(X)$ (in polynomial time) and next call B for instance $f(X)$ (in polynomial time too). We say that a problem A is NP-hard if any problem B in NP can be polynomially reduced to A; if A is also in NP then we say that A is NP-complete. Cook proved that the *Satisfiability* problem, SAT (i.e. given a propositional logic formula expressed in conjunctive normal form, is there any valuation of proposition symbols which makes the formula true?) is NP-complete [9]. On the one hand, it is easy to see that SAT is in NP: an NDTM could non-deterministically explore all possible valuations in polynomial-time by opening an execution branch for each possible valuation and check whether this valuation satisfies the formula. On the other hand, the idea behind its NP-hardness is the following. Let B be any problem in NP, that is, B is any problem which can be solved by a polynomial-time NDTM. We may construct a polynomial-size propositional logic expression denoting, at each execution time t, how the values of the configuration of such NDTM (i.e. state, contents of all cells, and the head location) depend on the corresponding values at time $t-1$. Since the NDTM solving B takes polynomial time, the number of times to be denoted in the formula is polynomial too. Since only a polynomial number of cells can be used in polynomial time, we infer that the size of that formula is polynomial. If that formula *also* requires that the NDTM ends at an accepting state, then the resulting expression will be satisfiable iff the NDTM (which solves B, i.e. *any* problem in NP) answers yes.

Given a problem C such that we do not know whether it is NP-hard or not, we can prove its NP-hardness as follows: We take any other problem B such that its NP-hardness has already been proved, and we prove that B can be polynomially reduced to C. Since all problem A in NP can be polynomially reduced to B (as B is NP-hard), by the transitivity of polynomial reductions we have that A (which can be *any* problem in NP) can be polynomially reduced to C, so C is NP-hard too. Thousands of NP-hard problems have been identified by reducing previously-known NP-hard problems to them and so on, in turn composing a tree of consecutive polynomial reductions which actually departed from SAT.

Many computational problems are not decision problems, i.e. they do not consist in answering *yes*/*no* but in returning some non-boolean result. *Optimization* problems are problems where we seek for the value that maximizes/minimizes some function. Let us illustrate this difference by formally introducing both problems under consideration in this paper. First we consider their *decision* versions. Knapsack (KN) is defined as follows: Given pairs of naturals $(g_1, w_1), \ldots, (g_n, w_n)$ (where each pair (g_i, w_i) denotes the i-th available *item*, whose *value* is g_i and whose *weight* is w_i), and two naturals W (maximum weight capacity) and L (required value), can we select some pairs (items) such that the addition of their weights is not higher than W and the addition of their values is not lower than L? Besides, Vertex Cover (VC) is defined as follows: Given a (non-directed) graph G and a natural K, can we select K or *less* vertexes of G in such a way that all edges are connected to a node included in the selection? The *optimization* versions of both problems are defined as expected: In Knapsack, given

$(g_1, w_1), \ldots, (g_n, w_n)$ and W, we have to select items in such a way that their addition of values is maximized but the addition of their weights is at most W. In Vertex Cover, given the graph, we have to select a set of vertexes covering all edges in such a way that the number of selected vertexes is minimized.

Typically, the NP-hardness of an optimization problem is studied in terms of its corresponding decision problem. Note that, in general, solving the optimization version of a problem implies trivially solving its decision version. Let us suppose that we wish to solve the decision version of Knapsack for a given set of items, a given minimum value L, and a given maximum weight W. If we observe that the *optimal* set of items which weights at most W has value V with $V \geq L$, then the answer of the decision problem is *yes*, else it is *not*.

However, NP-hard *optimization* problems can, in turn, be classified into some interesting subclasses. For the sake of notation simplicity, let us suppose that optimization problems are *minimization* problems (all the following notions are trivially adapted to the maximization case). We say that an optimization problem is in APX if there exists a polynomial-time algorithm and $\epsilon \in \mathbb{R}^+$ such that the algorithm finds solutions whose cost in the worst case is $1 + \epsilon$ times greater than the cost of optimal solutions. An optimization problem is APX-hard if any optimization problem in APX can be *PTAS reduced* to it, where a PTAS reduction is similar to the reduction between decision problems mentioned before, though the possibility to reach solutions with some approximation ratio in a problem must imply the possibility of reaching solutions of some approximation ratio, depending on the former ratio, in the other problem. An APX-hard problem is also APX-complete if it also belongs to APX. An optimization problem is in PTAS if, for *each* ϵ, there exists a polynomial-time algorithm whose approximation ratio is $1 + \epsilon$ in the worst case. Note that there must exist a polynomial-time algorithm *for each* ϵ, so the asymptotic complexity of each algorithm could be different and unbounded with ϵ. Also note that no APX-hard problem is in PTAS unless P=NP. Finally, an optimization problem is in FPTAS if, for each ϵ, there exists an algorithm running in polynomial-time with the input size *as well as with* $\frac{1}{\epsilon}$ whose approximation ratio is $1 + \epsilon$ in the worst case. Clearly, $FPTAS \subseteq PTAS \subseteq APX$. Moreover, if $P \neq NP$, then both inclusions are proper.