Chapter 2

Complexity

In Chapter 1 we learned some nifty algorithmic techniques. However, sometimes we are dealing with a problem where none of these techniques seem to work! Should we give up? And what do we do after giving up?

2.1 P and NP

Definition 2.1 (Computational Problem)**.** *A* ***computational problem*** *is defined as a (possibly infinite) set of inputs X and a computational goal. An algorithm solving the problem takes as input $x \in X$ and produces an output satisfying the goal.*

Definition 2.2 (Decision Problem)**.** *A* ***decision problem*** *is a computational problem specified by a partition of $X = X_{yes} \cup X_{no}$ into "yes" instances X_{yes} and "no" instances X_{no}. An algorithm solves the decision problem if, when run on $x \in X_{yes}$ outputs "yes" and when run on $x \in X_{no}$ outputs "no".*

Definition 2.3 (Function Problem)**.** *A* ***function problem*** *is a computational problem specified by a function Y, mapping each input $x \in X$ to a (possibly empty) set of feasible outputs Y_x for that input. An algorithm solves the function problem if, when run on $x \in X$, produces an output $y \in Y_x$, or outputs "no" in case $Y_x = \varnothing$.*

Remarks:

- For any function problem, asking the question "is Y_x nonempty" is a decision problem, which is no harder than the function problem: if we can find $y \in Y_x$ or tell that $Y_x = \varnothing$, then we can also state whether $Y_x \neq \varnothing$.

Definition 2.4 (Optimization Problem)**.** *An* ***optimization problem*** *is a function problem enhanced with a quality measure function $q(x, y)$. An algorithm solves the optimization problem if, when run on $x \in X$, produces an output $y \in Y_x$ which maximizes/minimizes $q(x, y)$, or outputs "no" in case $Y_x = \varnothing$.*

Remarks:

- Solving the optimization problem is no easier than solving the function problem: if we can find the best solution, we can also find a solution.
- We have seen several optimization problems in the previous chapter: What is the maximum value that can be packed into a knapsack? What is the optimal value of flow in a given network?
- Any linear program (Definition 1.18) is an optimization problem: an input to the problem is of the form $x = (A, b, c) \in X$, the set of admissible outputs for x is $Y_x = \{y \mid Ay \leq b\}$. The function to be maximized is $q(x, y) = c^T y$.
- The distinction between maximization and minimization problems is insignificant, since maximizing $q(x, y)$ is the same as minimizing $-q(x, y)$.
- In general, one can reformulate any optimization problem as a decision problem. The corresponding decision problem asks a "yes"/"no" question about the quality of a solution. For example: Can we pack items of total value at least 100 into a knapsack? Is there a flow of value at least 5 in a given network?

Definition 2.5. *Let (X, Y, q) be a maximization optimization problem, then the corresponding decision problem has as inputs pairs of the form (x, k) with $x \in X$, and (x, k) is a yes instance iff there exists $y \in Y_x$ such that $q(x, y) \geq k$.*

Lemma 2.6. *By solving the optimization problem, we can find a solution for the corresponding decision problem, or report none exist.*

Proof. We show this for the maximization variant, minimization being similar. Upon solving the optimization problem, one of the following two will hold:

- Either the optimization problem returns that $Y_x = \varnothing$, in which case we know that the answer is "no" for the decision problem;
- Or the optimization problem returns $y \in Y_x$ such that $q(x, y)$ is maximized. If any solution $y' \in Y_x$ satisfies $q(x, y') \geq k$, then the returned y will satisfy this as well, since $q(x, y) \geq q(x, y')$. Therefore, in this case we can just check whether $q(x, y) \geq k$ and return "yes" if so, and "no" otherwise. □

Remarks:

- In this chapter, we assume that $q(x, y)$ is efficient to evaluate, so the step where we check whether $q(x, y) \geq k$ in the proof of Lemma 2.6 does not add too much overhead. But there are optimization problems where finding the optimal $y \in Y_x$ is feasible within reasonable time, but computing the quality $q(x, y)$ is actually difficult.
- The opposite direction of Lemma 2.6 is usually also possible: if we have an algorithm for a decision problem, we can run it for many different decision values k in order to determine the optimal value. If we can then also find a feasible solution with this optimal value of k, then we are done. This way, we can solve the corresponding optimization problem, but it is not clear how efficiently.

- In the first part of this chapter, we will focus on decision problems.
- In the previous chapter, we have discussed a measure for discussing the running time of algorithms — the time complexity (Definition 1.5). Algorithms for Knapsack seemed to have exponential time complexity, while network flow and matching were polynomial.
- When studying time complexity, we are first and foremost interested in whether a problem can be solved in polynomial time. All problems that can be solved in polynomial time are considered "easy", and grouped together in a class.

Definition 2.7 (Complexity Class P)**.** *The* ***complexity class*** P *contains all decision problems that can be solved in deterministic polynomial time, i.e. where there exists an algorithm whose running time is in $\mathcal{O}(\mathsf{poly}(n))$, where n is the size of the input.*

Remarks:

- There are many problems that belong to class P: checking if an array is sorted, checking whether a graph is bipartite, checking whether an integer exists in an array, etc.
- A common misconception is that all problems that can be solved in polynomial time belong to class P; e.g. sorting an array, computing the product of two matrices, finding the maximum in an array. The reason this is *not* true is that they are *not* decision problems, but function problems.
- In order to show that a problem is in P, we usually give an explicit algorithm and analyze the running time directly. There is also an indirect way of showing that a problem is in P: if we can show that a problem known to be in P can be used to solve our problem. Then, our problem is "at most as difficult" as a problem in P, and is therefore also in P.

Definition 2.8 (Polynomial Reduction)**.** *Let A and B be computational problems (decision, function or optimization). A polynomial-time reduction from problem A to problem B is an algorithm that solves problem A using a polynomial number of calls to a subroutine for problem B, and polynomial additional time. If such an algorithm exists, then we write $A \leq B$.*

Remarks:

- Note that the reduction is not allowed to look inside the subroutine solving B, it should work no matter how the subroutine is implemented.

Lemma 2.9. *"Sorting an Array" $\leq$ "Maximum of an Array." Moreover, "Sorting an Array" can be solved in polynomial time.*

Proof. Assume we have a subroutine implementing "Maximum of an Array." Then, we can compute the maximum value iteratively, n times, at each step removing the maximum from the array. Putting together these maxima, we get the sorted array. We can find a maximum in linear time, and we call the maximum n times, which completes the proof. □

Remarks:

- We can use polynomial reductions between problems in order to determine classes of problems that have roughly the same difficulty.

Problem 2.10 (Clique)**.** *The input is a graph $G = (V, E)$ with n nodes, and an integer $k < n$. In the* ***clique (decision) problem*** *we want to know whether there exists a subset of k nodes in G such that there is an edge between any two nodes.*

Problem 2.11 (Independent Set)**.** *The input is a graph $G = (V, E)$ with n nodes, and an integer $k < n$. In the* ***independent set problem*** *we want to know whether there exists a subset of k nodes in G such that there is* ***no*** *edge between any two nodes in the subset.*

Theorem 2.12. *Clique $\leq$ Independent Set and Independent Set $\leq$ Clique.*

Proof. Let $K = \{\{u, v\} \in V^2 \mid u \neq v\}$ denote all possible edges in a graph G. Given a graph $G = (V, E)$, remove all edges E of this graph and add all edges in $K \setminus E$. Then, all cliques in G will become independent sets in the new graph, and vice-versa. This transformation can be computed in polynomial time. As a result, if we have an algorithm solving Independent Set, we can solve Clique by calling it on the new graph, and vice-versa. □

Problem 2.13 (Vertex Cover)**.** *The input is a graph $G = (V, E)$ with n nodes, and an integer $k < n$. In the* ***vertex cover problem*** *we want to know whether there exists a subset S of k nodes in G such that every edge in E has at least one of its two endpoints in S.*

Theorem 2.14. *Vertex Cover $\leq$ Independent Set and Independent Set $\leq$ Vertex Cover.*

Proof. For any independent set S, all other nodes $V \setminus S$ will be a vertex cover. This is because the nodes in S are not connected to each other, by definition, so the nodes $V \setminus S$ must cover all edges. Likewise, if the nodes $V \setminus S$ cover all edges, then the nodes S cannot have any edges between them, so S is an independent set. If independent set S has size ℓ, then vertex cover $V \setminus S$ has size $n - \ell$, so the maximum size S corresponds to the minimum size $V \setminus S$. It is now easy to see that if we have a subroutine that solves one of the two problems in polynomial time, then we can also solve the other in polynomial time. □

Lemma 2.15. *The reduction relation $\leq$ is transitive, i.e., if $A \leq B$ and $B \leq C$, then $A \leq C$.*

Proof. If there is an algorithm that can solve A using polynomially many calls to a procedure solving B and polynomial time outside those calls and an algorithm that can solve B using polynomially many calls to a procedure solving C and polynomial time outside those calls, then, by inlining the algorithm for B inside the one for A, we get that there exists an algorithm that can solve A using polynomially many calls to a procedure solving C and polynomial time outside those calls. □

Remarks:

- Therefore, there are polynomial reductions between Clique and Vertex Cover as well. If you know how to solve one of these problems in polynomial time, you can solve the other two in polynomial time as well! Unfortunately, nobody knows whether these problems (and many others!) are in P.

- Using polynomial reductions, we can define a hierarchy of different problems.

- Recall the decision variant of Knapsack: Can we pack items of total value at least k into the knapsack? Even though we could not find an efficient (polynomial) algorithm for Knapsack, we can try to see whether a simpler decision problem can be solved efficiently: Assume a friend has spent a considerable amount of time trying to pack items of total value at least k into the knapsack, and they claim that the answer is "yes". Can they prove this to you without redoing all the work they went through? Of course! They just need to show you which items to pack, and you can check for yourself that this is indeed a valid solution.

Definition 2.16 (NP)**.** *The class of non-deterministic polynomial problems* (NP) *contains decision problems $X = X_{yes} \cup X_{no}$ such that there is a polynomial algorithm A, called a polynomial verifier, which takes as input two arguments (x, y) with $x \in X$, and has the following properties:*

1. *For all $x \in X_{yes}$, there is a solution y of length polynomial in the length of x such that A outputs "yes" on (x, y).*

2. *For all $x \in X_{no}$, A outputs "no", irrespective of y.*

Remarks:

- The solution y in Definition 2.16 is also called a certificate or witness.

- Note how decision problems do *not* necessarily have a "native" notion of "solutions", in contrast to function/optimization problems; e.g. think what would be a feasible solution to "is an array sorted?"

- However, decision problems coming from function/optimization problems will oftentimes hint at an implicit notion of solutions; e.g. for Knapsack solutions are sets of items to pack that do not exceed the weight constraint.

- The definition above essentially states that "a decision problem is in NP if there is a way to define solutions for it, whose validity can be checked in polynomial time".

- The term NP ("non-deterministic polynomial") comes from an equivalent definition of NP, which is outside our scope.

- The class NP contains a large set of computational problems. But, we should also remember that there are also problems that are not in NP. For example, given any chess configuration, we not know how to determine in polynomial time whether there exists a chess move that guarantees that we will win the game (assuming both players are playing optimally).

Lemma 2.17. *Knapsack is in* NP*.*

Proof. For Knapsack, a solution is a set of items. In order for Knapsack to be in NP, we would be given a set of items and a total value, for example 100. Verifying whether a given set S of items has a total value of ≥ 100 is just a sum. We also need to make sure that S is admissible; i.e. it does not exceed the maximum knapsack weight constraint. This can be also done in linear time. Therefore, Knapsack is in NP. □

Lemma 2.18. *Clique is in* NP*.*

Proof. Solutions are subsets of nodes which form a clique. Given a graph and a proposed set of k nodes for a clique, we need to check whether the k proposed nodes are connected. This can be done by going through the list of all edges and counting the edges between the k nodes. If we counted $k(k-1)/2$ edges, the proposed node set forms a clique. □

Lemma 2.19. $\mathsf{P} \subseteq \mathsf{NP}$.

Proof. For any problem in P, we actually do not need the solutions to tell yes instances apart from no instances: we can just use $y = 0$ value (or any other value) as an artificial universal witness, and then checking whether $x \in X$ is a yes instance is done solely by solving the instance x in polynomial time and outputting accordingly (without relying on the witness y). □

Remarks:

- The definitions of P and NP only concern deterministic algorithms. A problem for which there exists a randomized algorithm running in polynomial time is not necessarily in P.

- Observe that all three problems (Clique, Independent Set, Vertex Cover) are in the class NP, and they seem to be more difficult than the problems in P. Are there even more difficult problems? Is Knapsack more difficult? What is the most difficult problem in NP?!

2.2 NP-hard

Definition 2.20 (NP-hard)**.** *A decision problem H is called* NP*-hard if there exists a polynomial reduction from every problem in* NP *to H.*

Remarks:

- In other words, an NP-hard problem is at least as difficult as *all* the problems in NP.
- NP-hard problems do *not* necessarily have to be in NP.
- Are there also NP-hard problems in NP?

Lemma 2.21. *Let B be an* NP*-hard problem and C be a problem for which there exists a polynomial reduction $B \leq C$. Then, C is* NP*-hard.*

Proof. Since B is NP-hard, there exists a polynomial reduction from every problem in NP to B. By using Lemma 2.15, we get that there also exists a polynomial reduction from every problem in NP to C. Therefore, C is NP-hard. □

Definition 2.22 (NP-complete)**.** *A decision problem is called* NP*-complete if it is* NP*-hard and it is contained in* NP*.*

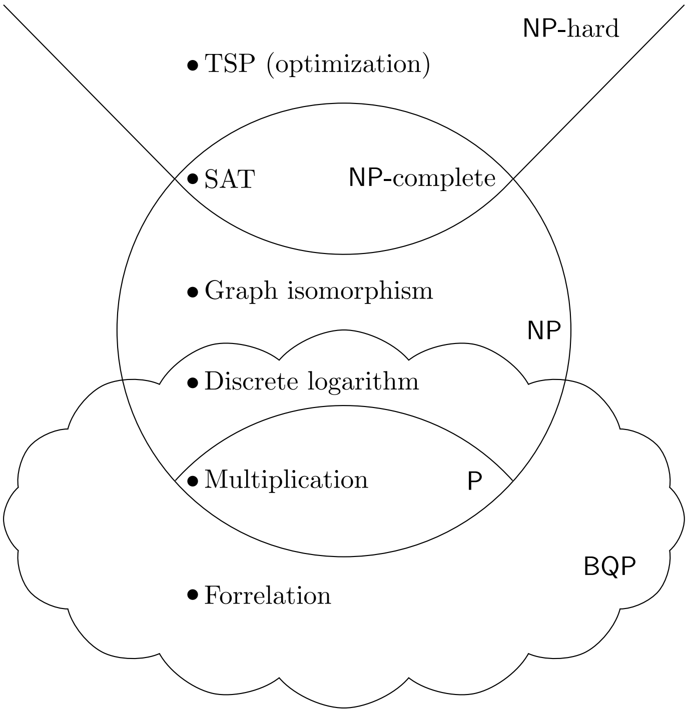


Figure 2.23: A complexity "pet" zoo: Diagram with different complexity classes and sample problems inside each class. The boolean satisfiability problem (SAT) will be introduced in the next section; the optimization version of the traveling salesperson problem (TSP) will be introduced in Section 2.5; the discrete logarithm and the graph isomorphism problems will be discussed in Chapter 3.

Remarks:

- Can we identify an NP-complete problem? If yes, we could use polynomial reductions to show that there are many more NP-complete problems. We will postpone this question to Section 2.3.
- Figure 2.23 visualizes some of the complexity classes and how they are related to each other. The figure just represents our current knowledge of how these classes relate to each other. There are still many open questions in complexity theory that could change the landscape once an answer is found. Also, there are many more classes not discussed.
- It is generally believed that there are some problems which are in NP but not in P. Or simply:

Conjecture 2.24. $\mathsf{P} \neq \mathsf{NP}$.

Remarks:

- The P versus NP question is one of the most important open scientific problems. There were many proof attempts, however, so far, without success.
- Another open question is whether quantum computers will allow to solve all difficult problems efficiently.

Definition 2.25 (BQP)**.** *A decision problem is in the class of* ***bounded-error quantum polynomial time*** (BQP) *problems, if it can be solved on a quantum computer in polynomial time and if for any input the probability to compute the wrong answer is at most* $1/3$.

Remarks:

- It is known that some problems from NP are also in BQP.
- The relation between the class of NP-complete problems and problems from BQP is unknown.
- It has been shown that there is a problem in BQP that is not in NP — the forrelation problem. In this problem, the question is whether a boolean function correlates to a Fourier transform of another boolean function. This result implies that only a quantum computer could solve (or even just verify) Forrelation efficiently.

2.3 Boolean Formulas

Definition 2.26 (Boolean Formula)**.** *Let* $\boldsymbol{x} = (x_1, \ldots, x_n)$ *be a vector of boolean variables; i.e. each variable can take one of the values True/False, or, equivalently, 1/0. A* ***boolean formula*** f *is an expression consisting of boolean variables and the logical connectives/operations AND, OR and NOT, denoted by* $\wedge$, $\vee$ *and* $\neg$, *respectively. Given an assignment of True/False values to each of the variables in* $\boldsymbol{x}$, *the formula* f *can be evaluated to yield a truth value True/False (or, equivalently,* $1/0$*). We say that a boolean formula is* ***satisfiable*** *if there* *exists an assignment of variables $\boldsymbol{x}$, such that $f(\boldsymbol{x}) = 1$. We will call x_i and its negation, $\neg x_i$, **literals**. Literals that are all connected by a logical OR (AND) operation we will call an **OR-clause** (AND-clause).*

Example 2.27. *Consider variables x_1, x_2, x_3, then an example of a boolean formula is $f = x_1 \wedge (x_2 \vee \neg x_3)$. Formula f evaluates to True if and only if x_1 is True and either x_2 is True or x_3 is False. In particular, this means that $\boldsymbol{x} = (1, 1, 1)$ is a satisfying assignment, while $\boldsymbol{x} = (0, 1, 0)$ and $\boldsymbol{x} = (1, 0, 1)$ are not. The subformula $(x_2 \vee \neg x_3)$ is an OR-clause.*

Definition 2.28 (Conjunctive Normal Form (CNF))**.** *A boolean formula f is in **conjunctive normal form (CNF)**, if it consists of OR-clauses that are connected by AND operations. A CNF formula is satisfiable if there is an assignment of variables such that all OR-clauses of the formula are satisfied.*

Remarks:

- Clause usually means OR-clause, unless stated otherwise.
- Any boolean formula can be rewritten as a CNF.

Problem 2.29 (SAT)**.** *In the boolean satisfiability problem (SAT), the task is to determine whether a given CNF formula is satisfiable.*

Problem 2.30 (3-SAT)**.** *3-SAT is a special case of SAT, where the given formula is a CNF that has exactly three literals in each OR-clause.*

Theorem 2.31. *SAT is in* NP*.*

Proof. We can use assignment vectors as solutions. Assume we are given an arbitrary boolean formula and an assignment of its variables that acts as a solution, we then need to verify in polynomial time whether the given assignment satisfies the formula. This is straight-forward to do in polynomial time. □

Remarks:

- Intuitively, SAT seems difficult to solve. With n variables, there are 2^n possible assignments to check. If only one assignment satisfies the boolean formula, we are trying to find a needle in a haystack.
- The satisfiability problem with DNF (disjunctive normal form) formulas, on the other hand, is in P. In a DNF formula, the operations AND and OR are swapped. Therefore, only one AND-clause has to be satisfied in order to satisfy the whole formula. To do so, it is sufficient to verify that there is an AND-clause that contains no variable in its negated and non-negated form simultaneously.
- Note that CNF $\leq$ DNF is not true. While every CNF can be expressed as DNF, the DNF may be exponentially bigger than the CNF, so a reduction like in Definition 2.8 would not be polynomial.

Theorem 2.32 (The Cook-Levin Theorem)**.** *SAT is* NP*-complete.*

Proof. The proof of this theorem is beyond the scope of this script and hence omitted here. The theorem is usually proven by using the alternative definition of the class NP via non-deterministic Turing machines. Using this definition one can show that it is possible to encode a Turing Machine as a SAT formula in polynomial time. □

Lemma 2.33. *3-SAT is* NP*-complete.*

Proof. Since 3-SAT is a special case of SAT, Theorem 2.31 also shows that 3-SAT is in NP. To show that 3-SAT is NP-hard, we reduce from SAT (i.e. Lemma 2.21). Assume we have a SAT formula where some clauses do not have 3 literals. We want to construct a 3-SAT formula that is satisfiable if and only if the SAT formula was satisfiable. Afterwards, we can use a subroutine solving 3-SAT to complete the proof. OR-clauses that contain more than three literals need to be split into smaller clauses, while clauses with less literals need to be filled up. Here, we will only consider some small examples. A clause with two literals $(x_1 \vee x_2)$ can simply be replaced by $(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y)$, where y is a new variable. A clause of four literals $(x_1 \vee x_2 \vee x_3 \vee x_4)$ we can rewrite as $(x_1 \vee x_2 \vee z) \wedge (x_3 \vee x_4 \vee \neg z)$, where z is a new variable. Successively applying such transformations achieves our goal. □

Remarks:

- Not all instances of SAT are hard. In 2-SAT, every clause contains exactly two literals. In contrast to 3-SAT, instances of 2-SAT can be solved in polynomial time.

- In order to show that some of the previously introduced problems (e.g. Independent Set, Clique or Vertex Cover) are also NP-complete, we will search for polynomial-time reductions from SAT (or 3-SAT) to these problems.

Theorem 2.34. *Independent Set is* NP*-hard.*

Proof. To show hardness, we show the reduction 3-SAT $\leq$ Independent Set. Therefore, we will model boolean formulas as graphs. Given a 3-CNF formula f, we construct a graph G as follows:

1. For each clause consisting of 3 literals in f (e.g. $\neg x_1 \vee x_2 \vee x_3$) we add a cluster of 3 nodes to the graph. Each literal from the clause corresponds to one node in the cluster. We label the three nodes with their literals (e.g. $\neg x_1$, x_2 and x_3). Note that, after processing multiple clauses, it might be that multiple nodes get the same label: such nodes are considered distinct.

2. For the edges, we connect the following:

   (a) All nodes inside the same cluster.

   (b) All pairs of nodes that represent the same variable in negated and non-negated form (e.g. x and $\neg x$).

Note that G can be constructed in polynomial time.

Next, we show that f is satisfiable if and only if the corresponding graph G has an independent set of size m, where m is the number of clauses in f.

" $\Longrightarrow$ " Assume that f is satisfiable; i.e. there is an assignment $\boldsymbol{x}$ such that in each clause there is a literal that evaluates to True. Select one such literal from each of the m clauses. Note that, in graph G, no two of these literals will be connected. This is because the corresponding nodes are all in different clusters and because nodes with labels x and $\neg x$ cannot both be True in the boolean assignment. Therefore, these m nodes form an independent set in G.

" $\Longleftarrow$ " Assume there exists an independent set of size m in G. This independent set cannot contain nodes from the same cluster, as such nodes would be connected. By definition of G, the independent set will also not contain nodes with labels x and $\neg x$. Therefore, the independent set of size m will have exactly one node from each cluster, and no variable will be present in both its negated and non-negated form. By setting the values of the node labels in the independent set to True, we get an assignment of variables that satisfies the boolean formula. Note that this might not set a value to all variables, but in that case those variables can just be set arbitrarily. □

Remarks:

- Note that the opposite direction, Independent Set $\leq$ 3-SAT, does not directly follow from the above proof. Not all graphs G can be reached with the construction. In order to show Independent Set $\leq$ SAT, one would have to show that every graph G can be transformed into a boolean formula in polynomial time.
- In Lemma 2.18, we showed that Independent Set is in NP. Therefore, Independent Set is also NP-complete.
- Clique and Vertex Cover are also NP-complete, since there are reductions in both directions between Independent Set and these problems, which we have shown in Theorem 2.12 and Theorem 2.14.
- In other words, all these problems are the most difficult problems in NP. If you could solve one of them in polynomial time, you could solve all problems in NP in polynomial time!
- What about Knapsack? We need yet another problem.

Problem 2.35 (Subset Sum)**.** *Given a number $s \in \mathbb{N}$ and a set of n natural numbers $x_1, \ldots, x_n$, does there exist a subset of these numbers that has a sum of s?*

Theorem 2.36. *Subset Sum is NP-complete.*

Proof. It is easy to verify whether a given subset of elements has the desired sum s. Therefore, Subset Sum is in NP. In order to show that Subset Sum is NP-hard, a reduction 3-SAT $\leq$ Subset Sum is shown. This reduction is non-trivial and will be omitted in this script. □

Theorem 2.37. *The Knapsack (decision) problem is NP-complete.*

a gate node is determined recursively by applying the logic operation that this gate represents to the values received through incoming edges. The output of the boolean circuit are the values of the output nodes.

Remarks:

- Boolean circuits provide a nice tool for understanding the complexity of computation. They are limited in computation power, yet may help to answer the $\mathsf{P} \neq \mathsf{NP}$ conjecture.

- In the literature, it is often assumed that AND and OR gates have a bounded number of input values, or that the circuit has a bounded depth.

- In the following, we will consider boolean circuits that have only one output value (True or False).

Problem 2.40 (Circuit-SAT)**.** *Let $C : \{0,1\}^n \to \{0,1\}$ be a boolean circuit with only one output value. In* ***Circuit-SAT****, the task is to determine whether for a given circuit C there exists an input vector $\boldsymbol{z} \in \{0,1\}^n$, such that $C(\boldsymbol{z}) = 1$.*

Theorem 2.41. *Circuit-SAT is in NP.*

Proof. We can use the vector $\boldsymbol{z}$ as a certificate. To check the it, we can just evaluate $C(\boldsymbol{z})$ in polynomial time. □

Theorem 2.42. *Circuit-SAT $\leq$ SAT.*

Proof. We need to construct a boolean formula that represents the gates of the given circuit. First, we introduce a new variable g_i representing gate i for each gate of the circuit. Next, for each gate i, we differentiate between the three possible gate types:

- If i is a NOT gate of gate j, then we add the clauses $(g_i \vee g_j) \wedge (\neg\, g_i \vee \neg g_j)$ to the formula.

- If k is an OR gate with inputs from two gates, i and j, we add the following clauses to the formula: $(\neg g_i \vee g_k) \wedge (\neg g_j \vee g_k) \wedge (g_i \vee g_j \vee \neg g_k)$.

- If k is an AND gate with inputs from two gates, i and j, we add the following clauses to the formula: $(\neg g_k \vee g_i) \wedge (\neg g_k \vee g_j) \wedge (\neg g_i \vee \neg g_j \vee g_k)$.

Observe that an AND or an OR gate with k inputs can be replaced by $k-1$ gates with two inputs by increasing the depth of the circuit, so our construction also applies to circuits where some gates have more than 2 inputs.

The resulting boolean formula is satisfiable if and only if the circuit evaluates to 1 for some vector $\boldsymbol{z}$. The construction was polynomial in the input size, and the resulting boolean formula can be transformed into a CNF with minimal overhead. □

Theorem 2.43. *Circuit-SAT is NP-complete.*

Proof. Previously, we have shown that Circuit-SAT is in NP. In order to show that Circuit-SAT is NP-hard, we will reduce SAT to Circuit-SAT; i.e. SAT $\leq$ Circuit-SAT. Let each variable of the boolean formula be an input to the circuit. For each negated variable, we can add a NOT gate. Then, for each clause in the formula, we add an OR gate that has all variables from the clause as input to the gate. And, finally, we add one AND gate that has values from all OR gates as its input. This gives a polynomial construction of a boolean formula as a circuit, thus showing that Circuit-SAT is NP-complete. □

Problem 2.44 (Minimum Circuit Size Problem (MCSP)). *Given a boolean function $f : \{0,1\}^n \to \{0,1\}$, represented as a truth table, the task of the* ***Minimum Circuit Size Problem (MCSP)*** *is to determine whether f can be represented by a boolean circuit of size at most s; i.e. a circuit with at most s gates.*

Remarks:

- There are several approaches of how boolean formulas can be simplified. Often the laws from boolean algebra are applied (Distributive law, Idempotent law, Identity law, Complement law, DeMorgan's law, Karnaugh maps). These minimization rules however do not guarantee that the resulting boolean formula uses the minimum number of gates.

- How difficult is MCSP?

Lemma 2.45. *MCSP is in* NP.

Proof. In essence, we can use as witness the circuit of size at most s that implements the function f. □

Remarks:

- Is MCSP also NP-hard? As of now, there is no answer to this question. In fact, there is evidence that this result might be difficult to establish. However, we know that the problem is NP-hard if the circuit is allowed to have multiple outputs.

- Some complexity classes are defined with respect to boolean circuits.

Definition 2.46 (Class AC^0). *The class AC^0 contains all decision problems that can be decided by a family of circuits $\{C_n\}$, where C_n has a constant depth and the size of the circuit is polynomial in n.*

Remarks:

- Note that we need to consider a circuit family, because the input size of each circuit is fixed, while the input size to a problem can vary.

- Note that we do not allow the boolean circuits to save output bits or use recursive operations.

Note that G can be constructed in polynomial time.

Next, we show that f is satisfiable if and only if the corresponding graph G has an independent set of size m, where m is the number of clauses in f.

" $\Longrightarrow$ " Assume that f is satisfiable; i.e. there is an assignment $\boldsymbol{x}$ such that in each clause there is a literal that evaluates to True. Select one such literal from each of the m clauses. Note that, in graph G, no two of these literals will be connected. This is because the corresponding nodes are all in different clusters and because nodes with labels x and $\neg x$ cannot both be True in the boolean assignment. Therefore, these m nodes form an independent set in G.

" $\Longleftarrow$ " Assume there exists an independent set of size m in G. This independent set cannot contain nodes from the same cluster, as such nodes would be connected. By definition of G, the independent set will also not contain nodes with labels x and $\neg x$. Therefore, the independent set of size m will have exactly one node from each cluster, and no variable will be present in both its negated and non-negated form. By setting the values of the node labels in the independent set to True, we get an assignment of variables that satisfies the boolean formula. Note that this might not set a value to all variables, but in that case those variables can just be set arbitrarily. □

Remarks:

- Note that the opposite direction, Independent Set $\leq$ 3-SAT, does not directly follow from the above proof. Not all graphs G can be reached with the construction. In order to show Independent Set $\leq$ SAT, one would have to show that every graph G can be transformed into a boolean formula in polynomial time.
- In Lemma 2.18, we showed that Independent Set is in NP. Therefore, Independent Set is also NP-complete.
- Clique and Vertex Cover are also NP-complete, since there are reductions in both directions between Independent Set and these problems, which we have shown in Theorem 2.12 and Theorem 2.14.
- In other words, all these problems are the most difficult problems in NP. If you could solve one of them in polynomial time, you could solve all problems in NP in polynomial time!
- What about Knapsack? We need yet another problem.

Problem 2.35 (Subset Sum)**.** *Given a number $s \in \mathbb{N}$ and a set of n natural numbers $x_1, \ldots, x_n$, does there exist a subset of these numbers that has a sum of s?*

Theorem 2.36. *Subset Sum is NP-complete.*

Proof. It is easy to verify whether a given subset of elements has the desired sum s. Therefore, Subset Sum is in NP. In order to show that Subset Sum is NP-hard, a reduction 3-SAT $\leq$ Subset Sum is shown. This reduction is non-trivial and will be omitted in this script. □

Theorem 2.37. *The Knapsack (decision) problem is NP-complete.*

Proof. Subset Sum is a special case of Knapsack: let the natural numbers be items with values $x_1, \ldots, x_n$ and weights $x_1, \ldots, x_n$, and let the knapsack have capacity s. If items of value s can be packed into the knapsack, then the corresponding items form a subset of sum s. Therefore, Subset Sum $\leq$ Knapsack. Since Subset Sum is NP-complete, Knapsack has to be NP-hard. In Lemma 2.17, we already showed that Knapsack is in NP. □

Remarks:

- In fact, thousands of interesting problems in many different areas are NP-complete: Achromatic Number, Battleship, Cut, Dominating Set, Equivalence Deletion, ..., Super Mario Bros, Tetris, ..., Zoe.
- Figure 2.38 below shows the reductions of NP-complete problems that we discuss in this chapter.

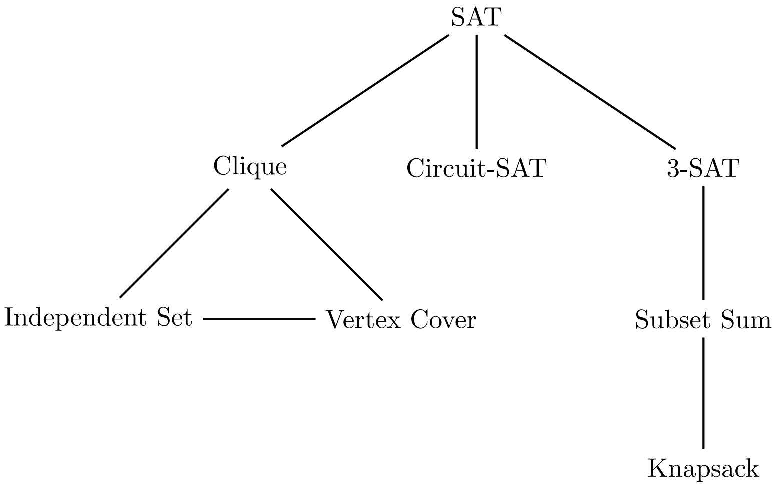


Figure 2.38: Reductions between NP-complete problems.

2.4 Boolean Circuits

Definition 2.39 (Boolean Circuit)**.** *Let $n, m \in \mathbb{N}$. A **boolean circuit** is a directed acyclic graph (DAG) with n boolean **input nodes** and m **output nodes**. The input nodes have no incoming edges, while the output nodes have no outgoing edges. The nodes of the graph that are not input nodes represent logical operations (AND, OR, NOT) and are called **gate nodes**. We will label the corresponding nodes $\wedge, \vee$ and $\neg$, respectively. Each NOT gate has in-degree 1. The in- and out- degrees of the other gates do not have to be bounded. We further define the **size** of the circuit to be its total number of gates, and the **depth** of the circuit to be the length of the longest (directed) path from an input to an output node.*

A boolean circuit C represents a mapping $C : \{0,1\}^n \to \{0,1\}^m$. The value of an input node is the value of the corresponding input bit of C. The value of *a gate node is determined recursively by applying the logic operation that this gate represents to the values received through incoming edges. The output of the boolean circuit are the values of the output nodes.*

Remarks:

- Boolean circuits provide a nice tool for understanding the complexity of computation. They are limited in computation power, yet may help to answer the $\mathsf{P} \neq \mathsf{NP}$ conjecture.

- In the literature, it is often assumed that AND and OR gates have a bounded number of input values, or that the circuit has a bounded depth.

- In the following, we will consider boolean circuits that have only one output value (True or False).

Problem 2.40 (Circuit-SAT)**.** *Let $C : \{0,1\}^n \rightarrow \{0,1\}$ be a boolean circuit with only one output value. In* ***Circuit-SAT****, the task is to determine whether for a given circuit C there exists an input vector $\boldsymbol{z} \in \{0,1\}^n$, such that $C(\boldsymbol{z}) = 1$.*

Theorem 2.41. *Circuit-SAT is in NP.*

Proof. We can use the vector $\boldsymbol{z}$ as a certificate. To check the it, we can just evaluate $C(\boldsymbol{z})$ in polynomial time. □

Theorem 2.42. *Circuit-SAT $\leq$ SAT.*

Proof. We need to construct a boolean formula that represents the gates of the given circuit. First, we introduce a new variable g_i representing gate i for each gate of the circuit. Next, for each gate i, we differentiate between the three possible gate types:

- If i is a NOT gate of gate j, then we add the clauses $(g_i \vee g_j) \wedge (\neg\, g_i \vee \neg g_j)$ to the formula.

- If k is an OR gate with inputs from two gates, i and j, we add the following clauses to the formula: $(\neg g_i \vee g_k) \wedge (\neg g_j \vee g_k) \wedge (g_i \vee g_j \vee \neg g_k)$.

- If k is an AND gate with inputs from two gates, i and j, we add the following clauses to the formula: $(\neg g_k \vee g_i) \wedge (\neg g_k \vee g_j) \wedge (\neg g_i \vee \neg g_j \vee g_k)$.

Observe that an AND or an OR gate with k inputs can be replaced by $k-1$ gates with two inputs by increasing the depth of the circuit, so our construction also applies to circuits where some gates have more than 2 inputs.

The resulting boolean formula is satisfiable if and only if the circuit evaluates to 1 for some vector $\boldsymbol{z}$. The construction was polynomial in the input size, and the resulting boolean formula can be transformed into a CNF with minimal overhead. □

Theorem 2.43. *Circuit-SAT is* NP*-complete.*

Proof. Previously, we have shown that Circuit-SAT is in NP. In order to show that Circuit-SAT is NP-hard, we will reduce SAT to Circuit-SAT; i.e. SAT $\leq$ Circuit-SAT. Let each variable of the boolean formula be an input to the circuit. For each negated variable, we can add a NOT gate. Then, for each clause in the formula, we add an OR gate that has all variables from the clause as input to the gate. And, finally, we add one AND gate that has values from all OR gates as its input. This gives a polynomial construction of a boolean formula as a circuit, thus showing that Circuit-SAT is NP-complete. □

Problem 2.44 (Minimum Circuit Size Problem (MCSP))**.** *Given a boolean function $f : \{0,1\}^n \to \{0,1\}$, represented as a truth table, the task of the* ***Minimum Circuit Size Problem (MCSP)*** *is to determine whether f can be represented by a boolean circuit of size at most s; i.e. a circuit with at most s gates.*

Remarks:

- There are several approaches of how boolean formulas can be simplified. Often the laws from boolean algebra are applied (Distributive law, Idempotent law, Identity law, Complement law, DeMorgan's law, Karnaugh maps). These minimization rules however do not guarantee that the resulting boolean formula uses the minimum number of gates.

- How difficult is MCSP?

Lemma 2.45. *MCSP is in* NP*.*

Proof. In essence, we can use as witness the circuit of size at most s that implements the function f. □

Remarks:

- Is MCSP also NP-hard? As of now, there is no answer to this question. In fact, there is evidence that this result might be difficult to establish. However, we know that the problem is NP-hard if the circuit is allowed to have multiple outputs.

- Some complexity classes are defined with respect to boolean circuits.

Definition 2.46 (Class AC^0)**.** *The class AC^0 contains all decision problems that can be decided by a family of circuits $\{C_n\}$, where C_n has a constant depth and the size of the circuit is polynomial in n.*

Remarks:

- Note that we need to consider a circuit family, because the input size of each circuit is fixed, while the input size to a problem can vary.

- Note that we do not allow the boolean circuits to save output bits or use recursive operations.

- There exist many other classes based on the boolean circuits: The class NC^0 is defined in the same way as AC^0, but the number of incoming edges to the AND and the OR gate is restricted to only two edges. Note that "two" can really be replaced by any constant, since, e.g., a three-input AND can be simulated with two binary ANDs. Both classes can be extended to versions that allow a non-constant depth of the circuit: AC^i and NC^i have depth $\mathcal{O}(\log^i n)$.

- Since the NAND gate (negated AND) can simulate all three of NOT, OR and AND, we could have defined AC and NC equivalently using only NAND gates. Note that this would fail to be true for the XOR gate (exclusive OR), since XOR can simulate neither AND nor OR.

2.5 Solving Hard Problems

We have seen that many important computational problems are known to be NP-hard. So what do we do if we encounter such a hard problem?

Remarks:

- We will now consider optimization problems instead of decision problems. This often makes sense in practice, where you usually want to find the best solution to some problem, instead of just deciding if the problem has a solution with a specific cost.

- So how can we approach a hard problem in practice? So far, we have mostly considered exact algorithms.

Definition 2.47 (Exact Algorithm)**.** *An **exact algorithm** always returns the optimal solution to the problem.*

Remarks:

- We have seen that for NP-complete problems we do not know if there is an exact algorithm with polynomial running time.

- Even when the worst-case running time of an algorithm is exponential, one can often use tricks to considerably reduce this running time in practice, e.g. by using look-ahead or pruning techniques as in Definition 1.9.

- What if we also accept weaker-than-optimal solutions while requiring that our algorithm always has a reasonable running time?

Definition 2.48 (Heuristic)**.** *A **heuristic** is an algorithm that is guaranteed to have a polynomial running time, at the cost of returning a suboptimal solution.*

Remarks:

- One example for a heuristic is our greedy algorithm for Knapsack, Algorithm 1.8.
- Good heuristics are usually based on insights into the structure or the behavior of the problem. Some heuristics are known to usually provide good solutions in practice.
- However, while heuristics are often good on some inputs, they may return weak solutions on other inputs.
- In general, it is better to have algorithms which provide guarantees that they always return a solution which is reasonably close to the optimum. We want to understand algorithms with such guarantees.

2.6 Vertex Cover Approximation

Let us first revisit the optimization version of Vertex Cover: we want to find a vertex cover $S \subseteq V$ in an input graph $G = (V, E)$, with $|S|$ as small as possible. Can we come up with an algorithm where we can prove that the size of the returned vertex cover is always "reasonably close" to the optimum?

Remarks:

- A natural approach is the greedy method of Algorithm 2.49.

```
def VertexCover_Greedy_Naive(G):
    $S = \emptyset$
    while $E \neq \emptyset$:
        select an arbitrary edge $(u, v) \in E$
        $S = S \cup \{u\}$
        remove all edges from $E$ that are adjacent to $u$
    return $S$
```

Algorithm 2.49: Naive greedy algorithm for Vertex Cover.

Lemma 2.50. *Algorithm 2.49 returns a vertex cover.*

Proof. An edge from E is only removed when one of its incident nodes is included in the set S. □

Remarks:

- However, the size of this vertex cover can be far from the optimal size.

Theorem 2.51. *The size $|S|$ of the vertex cover returned by Algorithm 2.49 can be an $n - 1$ factor larger than the optimum.*

Proof. Consider a "star" graph with n nodes and $n-1$ edges, where $n-1$ distinct leaf nodes u_1, u_2, $\ldots$, u_{n-1} are connected by an edge to the same center node v. Whenever Algorithm 2.49 selects an edge (u_i, v) in this graph, it can happen that it always chooses the leaf node u_i. In this case, the algorithm only removes a single edge (u_i, v) in each step, and the algorithm lasts for $n-1$ iterations of the loop. In the end, the algorithm returns the vertex cover $S = \{u_1, \ldots, u_{n-1}\}$, which consists of $n-1$ nodes.

On the other hand, the set $\{v\}$ is an optimal vertex cover of cost $c^* = 1$ in this graph, so we have $\frac{|S|}{c^*} = n - 1$. □

Remarks:

- It is a natural idea to try to improve Algorithm 2.49 with a clever tie-breaking rule: for example, to always select the higher-degree adjacent node of the chosen edge, or the highest-degree node altogether among the available nodes. This solves the star. However, there are more complicated counterexamples which show that even in this case, the solution we obtain can be a $\log n$ factor worse than the optimum.

- However, there is a slightly different greedy approach that provides much better guarantees.

```
def VertexCover_Greedy(G):
    $S = \emptyset$
    while $E \neq \emptyset$:
        select an arbitrary edge $(u, v) \in E$
        $S = S \cup \{u, v\}$
        remove all edges from $E$ that are adjacent to $u$ or $v$
    return $S$
```

Algorithm 2.52: Greedy algorithm for Vertex Cover.

Theorem 2.53. *Algorithm 2.52 always returns a vertex cover with* $|S| \leq 2 \cdot c^*$*.*

Proof. Algorithm 2.52 returns a correct vertex cover: an edge is only removed from E if at least one of its incident nodes is inserted into S.

Assume that the algorithm runs for c iterations of the main loop, i.e. it selects c different edges (u_i, v_i), and inserts both endpoints u_i and v_i of these edges into S for $i \in \{1, \ldots, c\}$. Note that none of these edges (u_i, v_i) share a node, because all edges adjacent to either u_i or v_i are removed when (u_i, v_i) is selected (in graph theory, such a set of edges is called a *matching*). This means that in the optimal vertex cover S^*, each vertex can only be incident to at most one of the edges (u_i, v_i); thus, in order to cover all the edges (u_i, v_i), we already need at least c nodes. As a result, we get that $c^* = |S^*| \geq c$.

On the other hand, our algorithm returns a vertex cover of size $|S| = 2 \cdot c$, so we have $|S| = 2 \cdot c \leq 2 \cdot c^*$. □

Remarks:

- In a graph of $\frac{n}{2}$ independent edges, the solution returned by Algorithm 2.52 is indeed 2 times worse than the optimum, so our analysis is tight.
- This algorithm was somewhat counter-intuitive; one might expect Algorithm 2.49 to be more efficient. However, as we have seen, Algorithm 2.52 is always at most a factor 2 away from the optimum c^*, while Algorithm 2.49 can be an $(n-1)$-factor away.
- Algorithms that are proven to always be within an α factor from the optimum are called *approximation algorithms*. Theorem 2.53 shows that Algorithm 2.52 is a 2-approximation algorithm for Vertex Cover.
- For our formal definition of this notion, we assume a minimization problem.

Definition 2.54 (Approximation Algorithm)**.** *We say that an algorithm $\mathcal{A}$ is an α-**approximation algorithm** if for every possible input of the problem, it returns a solution with a cost of at most $\alpha \cdot c^*$, where c^* is the cost of the optimal solution (the one with minimal cost).*

Remarks:

- The definition is similar for maximization problems. In this case, we denote the value of the optimal solution (the one with highest value) by v^*. For an approximation algorithm, we require that the returned solution always has a value of at least v^*/α.
- This section only considers approximations algorithms that run in polynomial time. That is exactly the point of these algorithms: to find a reasonably good solution without having an unreasonably high running time.

Definition 2.55. *We say that a problem is α-approximable if there exists a polynomial-time algorithm $\mathcal{A}$ that is an α-approximation algorithm for the problem.*

Remarks:

- In general, α can be any constant value with $\alpha > 1$, or it can also be a function of n (the size of the input), e.g. $\alpha = \log n$ or $\alpha = n^{3/4}$.
- In many cases, we can already get some approximability results from the most trivial algorithms. Independent Set naturally has $v^* \leq n$. Also, we can find a solution of value 1 by simply outputting an arbitrary single node. This shows that Independent Set is (at least) n-approximable.
- This allows us to classify hard problems, based on how well their optimum solution can be approximated. E.g. the complexity class of problems that are α-approximable for some constant α is called APX. Since Theorem 2.53 has $\alpha = 2$, Vertex Cover is in APX.

- For Vertex Cover, there is currently no $2 - \varepsilon$ approximation algorithm known for any $\varepsilon > 0$, so the algorithm above is indeed the best we have. Also, it is proven that no approximation better than 1.3606 is possible at all unless $\mathsf{P} = \mathsf{NP}$. This means that unless $\mathsf{P} = \mathsf{NP}$, we are unable to approximate the best solution of this problem arbitrarily well (in polynomial time).

2.7 Bin Packing Approximation

Let us now look at some other problems that are α-approximable to some constant α. We continue with the bin packing problem, which is similar to Knapsack.

Problem 2.56 (Bin Packing)**.** *We have a set of n items of sizes $x_1, \ldots, x_n$, and an unlimited number of available bins, each having a capacity of B. A set of items fits into a bin if their total size is at most B. Our goal is to put all of the items into bins, using the smallest possible number of bins.* → notebook

Remarks:

- You can easily imagine immediate applications of this, e.g. packing files on disks.
- For this problem, c^* denotes the number of bins that are used in the optimal solution.
- In Bin Packing, a simple greedy heuristic already allows us to obtain a 2-approximation.

```
def FirstFit(items, B):
    for each item in items:
      place item in the first bin where it still fits
      if item does not fit into any bin:
        open a new bin, and insert item into the new bin
```

Algorithm 2.57: First Fit algorithm for Bin Packing.

Lemma 2.58. *In Algorithm 2.57, at most 1 bin is filled at most half.*

Proof. Assume that there are at least 2 bins b_1 and b_2 that are filled at most half. This means that b_1 still has free space of at least $\frac{B}{2}$, and b_2 only contains items of size $x_i \leq \frac{B}{2}$. However, in this case, the items sorted into b_2 would also fit into b_1. This contradicts the First Fit algorithm, which only opens a new bin when the next item does not fit into any of the previous bins. □

Theorem 2.59. *Algorithm 2.57 is a 2-approximation.*

Proof. Assume that the First Fit algorithm uses m bins. Due to Lemma 2.58, at least $m-1$ of these bins are filled to a capacity of more than $\frac{B}{2}$. This implies

$$\sum_{i=1}^{n} x_i > (m-1) \cdot \frac{B}{2}.$$

On the other hand, we know that

$$c^* \geq \frac{\sum_{i=1}^{n} x_i}{B},$$

since all the items have to be sorted into a bin, and every bin can only take at most B. The two inequalities imply

$$B \cdot c^* > (m-1) \cdot \frac{B}{2},$$

and thus

$$2 \cdot c^* > m-1.$$

Since c^* and m are integers, this implies $2 \cdot c^* \geq m$, proving our claim. □

Remarks:

- One can even improve on this algorithm by first sorting the elements in decreasing order, and then applying the same First Fit rule. With only a slightly more detailed analysis, one can show that this improved algorithm achieves an approximation ratio of 1.5.
- Can we get a better approximation ratio than 1.5? Unfortunately not, unless we have $\mathsf{P} = \mathsf{NP}$. One can show that getting a better than 1.5 approximation is already an NP-hard problem.
- To prove this, we present a reduction to Partition.

Problem 2.60 (Partition)**.** *In the Partition Problem, given a set of n items of positive size $x_1, \ldots, x_n$, we want to decide if we can partition them into two groups such that the total sum is equal in the two groups.*

Remarks:

- Partition is a special case of Subset Sum where $s = \frac{1}{2} \cdot \sum_{i=1}^{n} x_i$.
- It is known that this special case is still NP-complete.

Theorem 2.61. *Subset Sum $\leq$ Partition.*

Proof. The reduction is not too complex, but a bit boring, so we do not discuss it here. □

Theorem 2.62. *It is NP-hard to solve Bin Packing with an α-approximation ratio for any constant $\alpha < \frac{3}{2}$.*

Proof. Given an input of Partition with integers $x_1, \ldots, x_n$, we convert it into a Bin Packing problem: we consider items of size $x_1, \ldots, x_n$, and we define $B = \frac{1}{2} \cdot \sum_{i=1}^{n} x_i$.

If the original Partition problem is solvable, then this Bin Packing problem can also be solved with 2 bins. In this case, an α-approximation algorithm with $\alpha < \frac{3}{2}$ must always return a solution with $m \leq c^* \cdot \alpha < 2 \cdot \frac{3}{2} = 3$ bins; since m is an integer, this means a solution with $m = 2$ bins.

On the other hand, if the items cannot be partitioned into two sets of size B, then the algorithm can only return a solution with $m \geq 3$ bins.

Hence an α-approximation algorithm for Bin Packing would also solve Partition in polynomial time: we can just solve the converted Bin Packing problem, and if $m = 2$, then output 'Yes', whereas if $m \geq 3$, then output 'No'. □

Remarks:

- Thus, Bin Packing is a problem that can be approximated to some constant $\alpha = 1.5$, but not arbitrarily well, i.e. not for any $\alpha > 1$.
- Let us consider a classical graph question.

2.8 Metric TSP Approximation

Problem 2.63 (Traveling Salesperson or TSP)**.** *The input is a clique graph $G = (V, E)$ (there is an edge between any two nodes of G), with positive edge weights, i.e. a function $d : E \to \mathbb{R}^+$. The goal of **TSP** is to find a Hamiltonian cycle in G where the total weight of the edges contained in the cycle is minimal.*

→ notebook

Remarks:

- In general, solving the traveling salesperson problem is NP-hard.
- Even approximating TSP to a constant factor is already NP-hard. We will prove this in the next section.
- On the other hand, there is a special case of the problem that does allow a constant-factor approximation.

Problem 2.64 (Metric TSP)**.** *In Metric TSP, the distances between any three nodes v_1, v_2, v_3 must satisfy the triangle-inequality: $d(v_1, v_2) \leq d(v_1, v_3) + d(v_3, v_2)$.*

Remarks:

- Note that the triangle inequality is a realistic assumption in real-world applications.
- This is another possible approach to solving hard problems in practice: maybe we can show that our actual problems are restricted to only a special case of the original problem, and that this special case is computationally more tractable.
- A Metric TSP already has a 2-approximation algorithm.

```
def Tree_Based_TSP(G)
    find a Minimum Spanning Tree $T$ in $G$
    form a sequence of nodes $P_0$ by traversing all nodes of $T$
    simplify $P_0$ to $P$ by only keeping the first occurrence of each node
    return $P$
```

Algorithm 2.65: Tree-based approximation for TSP.

Remarks:

- The Minimum Spanning Tree (MST) is, intuitively speaking, the subset of edges with smallest total weight which already connects the whole graph, i.e. there is a path between any two nodes through these edges. An MST in a graph can easily be found in polynomial time with some simple algorithms.

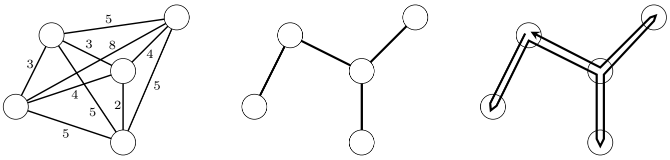
Theorem 2.66. *Algorithm 2.65 is a 2-approximation.*

Proof. See also Figure 2.67. Let t^* denote the total cost of the MST. Note that if we delete a single edge from any Hamiltonian cycle, we get a spanning tree, so t^* must be smaller than the cost of any Hamiltonian cycle. This implies $t^* < c^*$.

Now consider the cost of the Hamiltonian cycle returned by Algorithm 2.65. Since the traversal visits each edge of the spanning tree exactly twice, the total cost of P_0 is $2 \cdot t^*$. By only keeping the first occurrence of each node in P_0, we create "shortcuts": whenever we delete a node from P_0, we only shorten our tour due to the triangle inequality. As such, for the total costs we have

$$cost(P) \leq cost(P_0) = 2 \cdot t^* < 2 \cdot c^*.$$

□



(a) (b) (c) (d)

Figure 2.67: A weighted complete graph as an input to TSP (a), an MST in this graph (b), the tour P_0 obtained from this MST, starting from an arbitrary node (c), and the simplified tour P after removing repeated occurrences (d).

Remarks:

- Being more clever even allows a 1.5-approximation.
- For TSP, it is often reasonable to assume the even more special case of Euclidean TSP, where the nodes of the graph corresponds to specific points in a plane, for example. That is, each node v is associated with two coordinates v_x and v_y, and the weight of the edge between u and v is the Euclidean distance of u and v, i.e. $\sqrt{(u_x - v_x)^2 + (u_y - v_y)^2}$.

2.9 Non-Approximability: TSP

We now return to the general version of TSP. By a reduction to the following problem, we prove that approximating TSP up to a constant factor is already NP-hard.

Problem 2.68 (Hamiltonian Cycle)**.** *The input is a graph $G = (V, E)$. In the **Hamiltonian cycle problem** we want to know whether there exists a cycle in G which contains each node of the graph exactly once.*

Remarks:

- Hamiltonian cycle was among the first problems that were shown to be NP-complete.

Theorem 2.69. *In general graphs, TSP is NP-hard to approximate to any constant factor α.*

Proof. We provide a reduction to Hamiltonian Cycle, which is NP-complete.

Assume that we have an α-approximation algorithm for TSP. Given an input graph $G = (V, E)$ for Hamiltonian Cycle, we turn this into an instance of TSP on V. For each pair of nodes u and v, we define the weight of edge (u, v) in the TSP in the following way:

- if $(u, v) \in E$, then we assign $d(u, v) = 1$ in the TSP,
- if $(u, v) \notin E$, then we assign $d(u, v) = \alpha \cdot n + 1$ in the TSP.

If the initial graph had a Hamiltonian cycle, then the resulting TSP has a cycle of total weight n as the optimal solution. In this case, our approximation algorithm is guaranteed to return a solution of cost at most $\alpha \cdot n$. On the other hand, if there was no Hamiltonian cycle in the original graph, then the optimal TSP cycle must contain at least one edge of weight $\alpha \cdot n + 1$, and thus $c^* \geq \alpha \cdot n + 1$.

Hence an α-approximation returns a solution of cost at most $\alpha \cdot n$ if and only if the original graph had a Hamiltonian cycle; thus running such an approximation algorithm allows us to solve the Hamiltonian cycle problem. This shows that finding a polynomial-time α-approximation of TSP for *any* constant α is NP-hard. □

Remarks:

- The most difficult problems are those that cannot be approximated to any constant α, but only to a factor $\alpha = f(n)$ depending on n, e.g. $\alpha = \log n$ or $\alpha = \sqrt{n}$. This means that as the size of the problem becomes larger, the difference between the optimum and our solution also grows larger.
- For TSP, not even this is possible in polynomial time. The proof above works for any function $f(n)$ that can be computed in polynomial time.
- Remember the Independent Set problem? It turns out that this is also one of the most difficult problems from this perspective: approximating Independent Set to any factor that is slightly smaller than n would already imply that $\mathsf{P} = \mathsf{NP}$. (The proof of this fact is more involved than our previous claims, so we do not discuss it here.)

- This means that for any approximation algorithm $\mathcal{A}$, there are some graphs where $\mathcal{A}$ returns weak solutions.
- This $n^{1-\delta}$ factor is indeed huge. For example, it might be that the largest independent set contains $n^{0.99}$ nodes, but $\mathcal{A}$ returns an independent set of only 1 or 2 nodes. As such, an approximation algorithm like this is not useful in practice.
- The same result also holds for the Clique problem.
- Recall from Theorem 2.14 that as a decision problem, Independent Set $\leq$ Vertex Cover; however, we have seen that Vertex Cover is 2-approximable. This shows that even if the decision version of two problems are equivalent, the approximability of the optimization version may differ significantly.
- So while all NP-complete decision problems have the same difficulty in theory, their optimization variants are substantially different in practice.

2.10 FPTAS: Knapsack

We have now seen many problems that are α-approximable for a specific constant α. Sometimes we can even get arbitrarily close to the optimum in polynomial time: there exists a $(1+\varepsilon)$-approximation for any $\varepsilon > 0$.

Definition 2.70 (FPTAS). *We say an algorithm $\mathcal{A}$ is a fully polynomial-time approximation scheme (FPTAS) if for any $\varepsilon > 0$*

- *$\mathcal{A}$ is a $(1+\varepsilon)$-approximation algorithm, and*
- *the running time of $\mathcal{A}$ is polynomial in both n and $\frac{1}{\varepsilon}$.*

Remarks:

- While an FPTAS is not as good as a polynomial-time exact algorithm, it is almost as good: for any desired error ε, we can efficiently find a solution that is only ε away from the optimum.
- There is also a slightly weaker notion of PTAS (polynomial-time approximation scheme), where we only require the running time to be polynomial in n, but not in $\frac{1}{\varepsilon}$. That is, the running time is still polynomial in n for any fixed $\varepsilon > 0$, but it might increase quickly in $\frac{1}{\varepsilon}$; for example, it includes a factor of $n^{2^{1/\varepsilon}}$. These algorithms are not as useful in practice: if we want to get really close to the optimum by selecting a small ε, the running time still becomes unreasonably large.
- As an example for a nicely approximable problem, we revisit the Knapsack problem, and present an FPTAS for it.
- Since we now study the problem from more of a mathematical than a programming-based perspective, we introduce a shorter notation for the inputs of the problem.

Definition 2.71 (Knapsack notation)**.** *We will use C to denote the capacity of our Knapsack, and we denote the value and weight of the i^{th} item as v_i and w_i, respectively. We denote the number of items by n as before. Finally, let us use $v_{\max}$ to denote the value of the highest value item, i.e. $v_{\max} := \max v_i$.*

Remarks:

- Let us use v^* to denote the maximal value we can fit into the knapsack, i.e. the optimal solution.
- Note that if any item has $w_i > C$, then it can never fit into the knapsack, so we might as well remove such items. Hence, we will assume that even the largest items still fits into the knapsack, i.e. $v_{\max} \leq C$.
- Recall that we have discussed a dynamic programming solution for Knapsack, where each cell $V[i][c]$ of our DP table stored the maximum value that can be achieved with capacity c using only the first i items. The starting point of our FPTAS algorithm will be a slightly different variant of this DP method: in our new table, $W[i][v]$ will store the weight of the lowest-weight subset of items $1, \ldots, i$ that has a total value of v.
- One can show that this table W can also be computed with a similar dynamic programming method to Algorithm 1.12. As before, the table has n rows. The number of columns is now at most $n \cdot v_{\max}$, which is an upper bound on the value of any subset; we have n items, and each of them has value at most $v_{\max}$. From this alternative DP table, we can find the optimum v^* by taking the maximal v value in the table where $W[n][v] \leq C$. → notebook
- The main difference from the original DP algorithm is that now each column expresses the *value* of a specific subset of items, instead of the *weight* of a specific subset of items as before.
- We can now use this alternative DP algorithm to develop an FPTAS for Knapsack. The main idea is to slightly round up the values of the items, which results in some inaccuracy for the algorithm, but it also reduces the number of columns in our DP table.

```
def Knapsack_FPTAS(items, C):
    $k = \varepsilon \cdot v_{\max} \,/\, n$
    consider the modified Knapsack problem where item $i$ has weight $w_i$ and value $\hat{v}_i = \lceil \frac{v_i}{k} \rceil \cdot k$
    run the alternative DP algorithm on this problem, with columns only corresponding to multiples of $k$
```

Algorithm 2.72: FPTAS algorithm for Knapsack.

Lemma 2.73. *Algorithm 2.72 has a running time of $\mathcal{O}\big(n^3 \cdot \frac{1}{\varepsilon}\big)$.*

Proof. Given the parameter ε, the algorithm introduces a scaling parameter $k = \varepsilon \cdot v_{\max} / n$, and defines the *rounded-up value* of item i as $\hat{v}_i = \lceil \frac{v_i}{k} \rceil \cdot k$. This means that whichever subset of nodes we select, the total value of the subset is a multiple of k.

On the other hand, any subset of the items has a total value of $n \cdot v_{\max}$ at most. If each column represents a multiple of k, then the table has at most $n \cdot v_{\max} / k = n^2 \cdot \frac{1}{\varepsilon}$ columns. The table still has n rows, so the total running time of the DP algorithm with these rounded values is $O(n^3 \cdot \frac{1}{\varepsilon})$. □

Theorem 2.74. *Algorithm 2.72 is an FPTAS.*

Proof. The polynomial running time has already been established in Lemma 2.73. Let us use S to denote the set of items chosen by Algorithm 2.72. Since we round the values to multiples of k, the value of each item has been overestimated by k at most, so $\hat{v}_i - k \leq v_i$. Hence for the total value of items in S we have

$$\sum_{i \in S} v_i \geq \sum_{i \in S} (\hat{v}_i - k) = \sum_{i \in S} \hat{v}_i - |S| \cdot k = \sum_{i \in S} \hat{v}_i - |S| \cdot \frac{\varepsilon \cdot v_{\max}}{n} \geq \sum_{i \in S} \hat{v}_i - \varepsilon \cdot v_{\max}.$$

To simplify this last expression even more, we make two more observations. First, note that $\sum_{i \in S} \hat{v}_i$ is the optimal solution of the rounded problem found by our algorithm. Since we only rounded values upwards, this is larger or equal to the solution of the original problem v^*. Second, choosing the item with maximal value $v_{\max}$ is always a valid solution, so we also have $v_{\max} \leq v^*$. Hence for the total value in S we have

$$\sum_{i \in S} \hat{v}_i - \varepsilon \cdot v_{\max} \geq v^* - \varepsilon \cdot v^* = (1 - \varepsilon) \cdot v^*.$$

Note that this shows $v^* / v(S) \leq \frac{1}{1-\varepsilon}$, while our original FPTAS definition requires a slightly different relation: that $v^* / v(S) \leq (1 + \varepsilon)$. However, since we now have an $\alpha = \frac{1}{1-\hat{\varepsilon}}$-approximation algorithm for any $\hat{\varepsilon} > 0$, we can obtain $\alpha = (1 + \varepsilon)$ by choosing $\hat{\varepsilon} = \frac{\varepsilon}{1+\varepsilon}$. □

Chapter Notes

The first formal definition of computation was provided by Alan Turing in 1936 [20]. In this paper, Turing presents an automatic machine (now known as Turing machine) that can compute certain classes of numbers. We will discuss the Turing machine in more detail in the last chapter. The birth of computational complexity as a field is attributed to Hartmanis and Stearns for their paper "On the computational complexity of algorithms" [14]. They proposed to measure time with respect to the input size and showed that some problems can only be solved if more time is given.

The first problem that was shown to be in P was the maximum matching problem, see Edmonds [9]. In another paper, Edmonds introduced a notion that is equivalent to the class NP [10]. In 1971, Cook [6] showed that SAT was NP-complete. Two years later, independently of Cook's result, Levin [17] showed that six problems were NP-complete, the so-called universal search problems. Both authors have formulated the famous P versus NP problem in computer science. The name NP was however given to the class a year later by Karp [16], who proved that 21 (combinatorial) problems were NP-complete. He thereby first used the technique of polynomial reductions in these proofs. Since then, thousands of problems in different areas of science have been shown to be NP-hard. Also many new complexity classes have been proposed in the literature. Different "zookeepers" around the world keep track of the newly introduced complexity classes in their complexity zoos [1]. The question $\mathsf{P} \neq \mathsf{NP}$ still remains open. In 2000, the P versus NP problem was announced as one of the seven Millennium Prize Problems by the Clay Mathematics Institute [4]. Solving P versus NP (or BQP versus NP) will be a major milestone in science, in case of equality ($\mathsf{BQP} = \mathsf{NP}$) there will be amazing practical consequences.

Circuit theory is almost 200 years old, being studied by electrical engineers and physicists. In the 1940s, the invention of semiconductor devices and later the transistor further boosted the quest for understanding circuits. The computational model of boolean circuits was introduced by Claude Shannon [19] in 1949, who showed that most boolean functions require circuits of exponential size. It was a natural step to restrict the circuits in size and depth and consider problems that are still computable on restricted circuits. Furst, Saxe and Sipser [11] for example showed that Parity is not in AC^0, but in NC^1. The first connection between circuits and Turing machines has been made by Savage [18]. It is generally believed that proving bounds using boolean circuits is easier than by using Turing machines.

While all known exact algorithms for NP-hard problems have a superpolynomial runtime in the worst case, there are numerous possible tricks and optimizations that can make these algorithms viable on special cases of the problem, even for large inputs. For example, there are yearly SAT-competitions where SAT-solver algorithms try to decide the satisfiability of SAT formulas from real-life applications, with many of these formulas containing millions of variables and tens of millions of clauses [2].

Heuristic solutions to hard problems have also been extensively studied. In particular, there is a wide range of so-called metaheuristics, which are general approaches and techniques for developing a heuristic solution to a problem. This includes some simpler approaches like local search or gradient descent, and also some more sophisticated ones like simulated annealing, tabu search or genetic algorithms [13]. Note that many techniques in machine learning are also developed for this purpose: to provide heuristic solutions to hard problems.

The 2-approximation algorithm for Vertex Cover has long been known, discovered by both Gavril and Yannakakis independently [12]. The FPTAS algorithm for Knapsack has also been around for a long time, and it is one of the most popular FPTAS examples [21].

Bin Packing has also been studied for multiple decades [12]. A recent analysis of the First Fit heuristic is available by Dósa and Sgall, proving the even stronger result that First Fit is a 1.7-approximation [8]. The variant which first sorts the items in decreasing order is known to even provide a $\frac{11}{9}$-approximation up to an additive constant, and this bound is known to be tight [7].

For Traveling Salesperson, a slightly improved version of our approximation algorithm is due to Christofides, which improves the approximation ratio to only 1.5 [5]. In the special case of an Euclidean TSP, there even exists a (rather complicated) PTAS algorithm, which received a Gödel prize [3].

The inapproximability result for Independent Set was the final result of a line of lower bounds in the early 2000s; John Hastad also received a Gödel prize for this result [15, 22].

This chapter was written in collaboration with Darya Melnyk, Pál András Papp, and Andrei Constantinescu.

Bibliography

[1] Complexity zoo. `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`.

[2] The international SAT competition. `http://www.satcompetition.org`.

[3] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *J. ACM*, 45(5):753–782, September 1998.

[4] James A Carlson, Arthur Jaffe, and Andrew Wiles. *The millennium prize problems*. American Mathematical Society Providence, RI, 2006.

[5] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[6] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[7] György Dósa. The tight bound of first fit decreasing bin-packing algorithm is ffd(i) ¡ 11/9 opt(i)+6/9. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11, Berlin, Heidelberg, 2007.

[8] György Dósa and Jiri Sgall. First Fit bin packing: A tight analysis. In *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *LIPIcs*, pages 538–549, Dagstuhl, Germany, 2013.

[9] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of research of the National Bureau of Standards B*, 69(125-130):55–56, 1965.

[10] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

[11] Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical systems theory*, 17(1):13–27, 1984.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[13] Fred W Glover and Gary A Kochenberger. *Handbook of metaheuristics*, volume 57. Springer Science & Business Media, 2006.

[14] Juris Hartmanis and Richard E Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[15] Johan Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 627–636. IEEE, 1996.

[16] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[17] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

[18] John E Savage. Computational work and time on finite machines. *Journal of the ACM (JACM)*, 19(4):660–674, 1972.

[19] Claude E Shannon. Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715, 1949.

[20] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[21] Vijay V Vazirani. *Approximation algorithms.* Springer Science & Business Media, 2013.

[22] David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '06, page 681–690, New York, NY, USA, 2006.