Economics and Computation

ECON 425/563 and CPSC 455/555

Professor Dirk Bergemann and Professor Joan Feigenbaum

Lecture IV

In case of any questions and/or remarks on these lecture notes, please contact Oliver Bunn at oliver.bunn(at)yale.edu.

1 Introduction to the Theory of Computation

The theory of computation pursues the following goals:

- Formulate mathematical models of ”computation”.
- Formally address issues such as:
  - What is or is not computable?
  - What is ”efficient computation”?
  - Efficient computational solutions, or ”algorithms”, for problems of interest.
  - Proofs that there are no efficient algorithms for some problems of interest (including some for which there *are* inefficient algorithms).
  - Proofs that two problems are ”equivalent” in computational difficulty or that one problem is harder than the other.

The theory of computation was initiated by Alan Turing, [Tur36a] and [Tur36b]. The fascinating and ultimately tragic life of Turing is chronicled brilliantly in the play ”Breaking the Code”, [Whi]. In 1936, the date of Turing's seminal papers, there were no actual computers, and so Turing was not concerned with efficiency. Formal treatment of efficient computation was initiated by Juris Hartmanis and Richard Stearns, [HS65].
The ”Nobel Prize of Computer Science” is the Turing Award, which Hartmanis and Stearns won for their paper [HS65].[1]

1.1 Administrative Information

Yale courses on the basics of computational theory are:

- CPSC 365,
- CPSC 468/568.

Standard introductory textbooks that deal with computational theory are:

- Cormen, Leiserson and Rivest: **Introduction to Algorithms**, [CLR01].
- Garey and Johnson: **Computers and Intractability: A Guide to the Theory of NP-Completeness**, [GJ79].

[1]Turing Award winners' names are in red throughout these notes.

2 Computational Complexity

2.1 Building Blocks

A **computational problem** is a precise statement of a **general question** to be answered, usually possessing several **parameters** or free variables. One specifies a problem by giving:

- precise descriptions of all parameters;
- a precise statement of the properties that a **solution** must satisfy.

An **instance** of the problem is obtained by specifying particular values for all parameters.

Example 1 *[Traveling Salesman Problem (TSP)]*

The parameters are:

- *a finite set* $\{c_1, ..., c_m\}$ *of cities,*
- *the distance* $d(c_i, c_j) \in \mathbb{R}_0^+$ *between city* c_i *and city* c_j *for each pair of cities* (c_i, c_j)*.*

A solution is an ordering (or ”tour”)

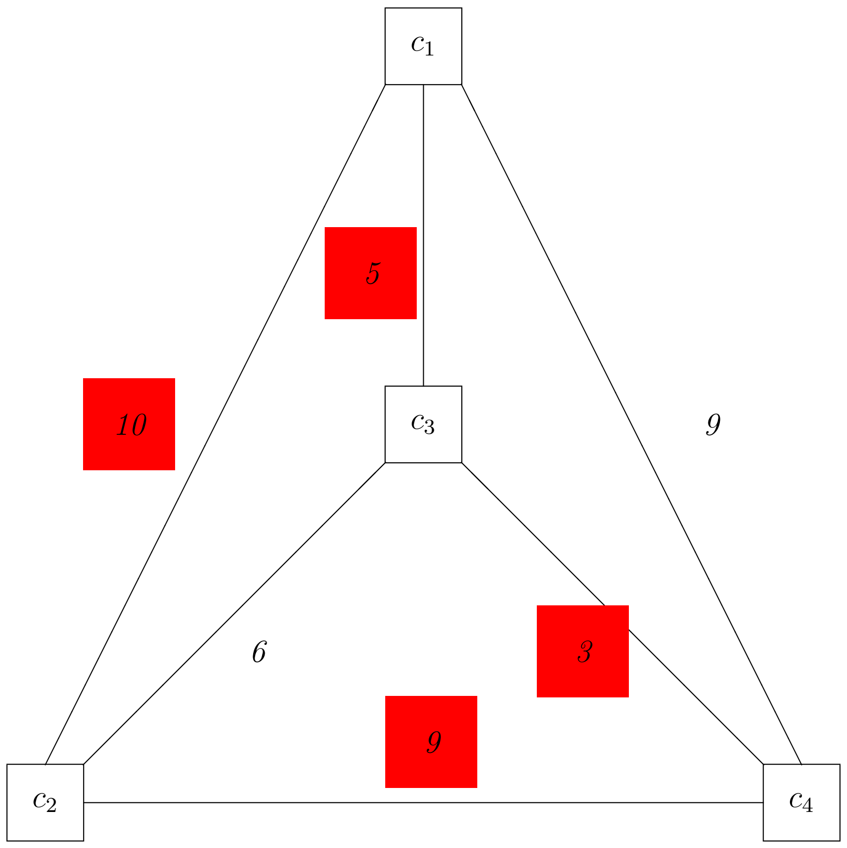
$$\langle c_{\pi(1)}, ..., c_{\pi(m)} \rangle$$

that minimizes

$$\left[\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right] + d(c_{\pi(m)}, c_{\pi(1)}),$$

where $\pi : \{1, ..., m\} \to \{1, ..., m\}$ *is a bijective (one-to-one and onto) mapping.*

As an example, consider the following instance that will be called $*$ *for further reference:*



In instance $*$ *of the TSP, the tour* $\langle c_1, c_2, c_4, c_3\rangle$*, edges of which are labeled in red, has length* 27 *and is a solution (i.e., an optimal or minimum-length tour).*

Note that there are TSP instances on m cities *for any* $m \geqslant 2$. The fact that *a problem is defined on an infinite family of instances* is crucial to computational theory. An **algorithm** that solves a problem is a **step-by-step procedure** that, for *any* instance, is **guaranteed to produce a solution**.

In this course, it will suffice to think of an algorithm as a program in a standard programming language such as Java or C++ and of "steps" as native commands in that language. In fact, we will usually describe algorithms in even higher level notation than that and count as "steps" instructions that correspond intuitively to atomic operations on a computer, e.g. reading or writing a memory location, performing an arithmetic or logical operation, sending a packet across a network link, etc. These components of a *computational model* can be made completely formal with the notion of a *Turing-machine model.*[2]

Instances are encoded as finite strings of symbols from a finite alphabet. For example,

[2]See Appendix *A* for a definition of the *Turing-machine model.*

one can encode TSP instances as strings over the alphabet

$$\{c, [,], /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

The encoding instance of instance $*$ would then be

`c[1]c[2]c[3]c[4]//10/5/9//6/9//3.`

2.2 Computational Complexity

The **length** or **size** of an instance is the number n of symbols needed to encode it. Intuitively, we expect an algorithm to *spend more computational resources on larger instances of a problem than on smaller instances.* The following two natural measures capture the computational complexity of a problem:

- "Time Complexity" $T(n)$
  $T(n)$ denotes the maximum number of steps needed to solve an instance of length n.

- "Space Complexity" $S(n)$
  $S(n)$ denotes the maximum number of memory cells needed to solve an instance of length n.

One may also consider some less traditional measures, e.g. bandwidth consumed during a distributed computation over a network.

Definition 1 *A problem is called **polynomial-time solvable** iff it can be solved in time*

$$T(n) = \mathcal{O}(n^c),$$

for some $c \in \mathbb{N}$.

In the following, conjunctive-normal-form formulas (exact definition below) will be presented as a structure to examine the concept of time-complexity in more detail.

2.2.1 Conjunctive-Normal-Form Formulas (CNF)

Let $\{x_1, ..., x_m\}$ be a set of Boolean variables, i.e. variables that can take the values T or F (true or false). A *literal* is either a variable x_i or its negation $\bar{x_i}$. A *clause* C_j is a disjunction of literals, e.g. $x_1 \vee \bar{x_{17}} \vee \bar{x_{20}}$. A *Conjunctive-normal-form (CNF) formula* is a conjunction of clauses, e.g. $C_1 \wedge C_2$, where $C_1 = (\bar{x_2} \vee x_5)$ and $C_2 = (x_1 \vee x_3 \vee \bar{x_4})$. An *assignment* $\epsilon_1, ..., \epsilon_m$ of truth values to the variables $x_1, ..., x_m$ in a CNF formula $\mathcal{C}$ is a *satisfying assignment* if $\mathcal{C}(\epsilon_1, ..., \epsilon_m)$ evaluates to T. For example, (T, F, T, T, T) is a satisfying assignment of $C_1 \wedge C_2$ above, but (T, T, F, T, F) is not.

Example 2 (Verification of Satisfiability)

An instance is a pair $(\mathcal{C}, \vec{\epsilon})$, where $\mathcal{C}$ is a CNF formula on m variables with k clauses, and $\vec{\epsilon} = (\epsilon_1, ..., \epsilon_m)$ is an assignment to the variables of $\mathcal{C}$. The solution is "yes" if $\vec{\epsilon}$ is a satisfying assignment of $\mathcal{C}$ and "no" otherwise.

Observe that one is given a formula and an assignment. So, example 2 is solely about determining a true/false-statement.

To see why this problem is solvable in polynomial time, observe that, under a reasonable encoding, the length of an instance is at least proportional to $m + k$ and at most proportional to mk (the former applies if all clauses are short and the latter if they are long). On the other hand, the formula can be evaluated at $\vec{\epsilon}$ by a program that performs $\mathcal{O}(mk)$ logical operations.

Definition 2 (The Class P)

The class of "yes/no" (or ***decision****) problems that are solvable in polynomial time is denoted by P.*

Instead of the verification from Example 2, one might also be interested in finding/determining a satisfying assignment if one is given a CNF. This is captured by the following example:

Example 3 (Solving for Assignment in SAT - Search Version of SAT) *An instance of this problem is a CNF formula $\mathcal{C}$ on Boolean variables $x_1, ..., x_m$. A solution is a satisfying assignment $\epsilon_1, ..., \epsilon_m$ of $\mathcal{C}$ if one exists.*

Observe that in this case, an assignment is no longer part of the input, but part of the solution. Instead of simply deciding whether a particular assignment is true or false, a satisfying assignment needs to be determined. If a solution to Example 3 is found, then one can use the logic as outlined in the sequel of Example 2 to quickly verify a solution. This leads to the following definition:

Definition 3 ***Nondeterministic-polynomial-time*** *problems are those for which correct solutions, if they exist, can be verified in polynomial time.*

Hence, nondeterministic-polynomial-time problems are solely characterized by the efficiency of the verification of their solution.

Note that the search version of the satisfiability problem is a *partial relation* on the set of CNF formulas, in that it is not defined on unsatisfiable formulas (i.e. those formulas $\mathcal{C}(x_1, ..., x_m)$ on which *all* assignments evaluate to F). Hence, it is natural to consider the following example that considers the "decision version" of the satisfiability problem:

Example 4 (Decision Version of SAT) *Here, an instance is once again a CNF formula $\mathcal{C}$, but a solution is simply "yes", if $\mathcal{C}$ is satisfiable (i.e. if $\exists\vec{\epsilon} : \mathcal{C}(\vec{\epsilon}) = T$) and "no" if $\mathcal{C}$ is unsatisfiable.*

An important technique in the study of computational complexity is *reducing search to decision*. In the context of the satisfiability problem, this means that, if there were a polynomial-time algorithm for the problem in Example 4, there would be one for the problem in Example 3.

To see this, note that, if $\mathcal{C}(x_1, ..., x_i, ..., x_m)$ is a CNF formula on m Boolean variables, then both

$$\mathcal{C}_{i,T}(x_1, ..., T, ..., x_m)$$

and

$$\mathcal{C}_{i,F}(x_1, ..., F, ..., x_m)$$

are CNF formulae on $m-1$ variables. Both can be computed quickly by "plugging in"the appropriate truth value for the Boolean variable x_i and "simplifying" the result.

For example, if

$$\mathcal{C}(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_1} \vee \bar{x_2} \vee x_3),$$

then

$$\mathcal{C}_{1,T} = (x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee x_3)$$

and

$$\mathcal{C}_{1,F} = (x_2 \vee \bar{x_3}).$$

Let "decide-sat()" be an algorithm that takes as input a CNF formula, outputs "yes" if the input is satisfiable, and outputs "no" otherwise. Observe that the following algorithm, which uses "decide-sat()" as a subroutine, solves the search version of the satisfiability problem:

```
$find - sat(\mathcal{C})$
{
    $\Psi \leftarrow \mathcal{C}$;
    for $i \leftarrow 1$ to $m$
    {
        if (decide-sat($\Psi_{i,T}$) == yes)
        {
            $\epsilon_i \leftarrow T$;
            $\Psi \leftarrow \Psi_{i,T}$;
        }
        else if (decide-sat($\Psi_{i,F}$) == yes)
        {
            $\epsilon_i \leftarrow F$;
            $\Psi \leftarrow \Psi_{i,F}$;
        }
        else OUTPUT("$\mathcal{C}$ is unsatisfiable");
    }
    OUTPUT($\epsilon_1, ..., \epsilon_m$)
}
```

To see that this **reduction** of "find-sat()" to "decide-sat()" is correct, note that, if $\mathcal{C}$ is satisfiable, one or both of the following statements must be true:

- "$\mathcal{C}$ has a satisfying assignment in which $x_1 = T$, and $\mathcal{C}_{1,T}$ is also satisfiable."
- "$\mathcal{C}$ has a satisfying assignment in which $x_1 = F$, and $\mathcal{C}_{1,F}$ is also satisfiable."

In the first execution of the for-loop in "find-sat()", one discovers whether *both* of these statements are false; if they are, the algorithm terminates and declares $\mathcal{C}$ to be unsatisfiable. If at least one of these statements is true, the algorithm records the first bit of a satisfying assignment and proceeds on a formula on $m-1$ variables. Iterations 2 through m of the for-loop perform the analogous computation for variables x_2 through x_m.

To see that this is a **polynomial-time reduction**, note that the size n of input $\mathcal{C}$ is greater than m. "find-sat()" makes at most $2m < 2n$ calls to "decision-sat()". Thus, if "decision-sat()" ran in time[3] $\mathcal{O}(n^c)$, "find-sat()" would run in time $\mathcal{O}(n^{c+1})$.

[3]Note that this hypothesis is believed to be false, i.e., it is not expected that the problem of Example 4 is in P.

2.3 The Class NP

The notion of nondeterministic-polynomial time (see Definition 3) can now be fleshed out to yield a formal definition of the class NP. Recall that a computational problem is characterized by a set of X of instances and a set Y of solutions. We generalize our earlier notion of "computational problem" by allowing for the possibility that some instances have no solutions. Let $|x|$ and $|y|$ denote the sizes of strings $x \in X$ and $y \in Y$.

Definition 4 (The Class NP)
*The class **NP** of nondeterministic polynomial-time decision problems are those in which the sets X and Y have the following properties:*

- *There is a polynomial p such that*

$$|y| \leqslant p(|x|),$$

  whenever y is a solution to the instance x.

- *The decision problem "is y a solution to the instance x" is in P.*

The set of "yes-instances", i.e. the set of all x for which there exists at least one solution y, is called an **NP set** or an **NP language**. The corresponding *NP search problem* is "given an instance x, find a solution y if one exists".

The language SAT of satisfiable CNF formulas is a canonical NP set. Example 1 gives an *optimization version* of the TSP. The TSP can be formulated as a nondeterministic polynomial-time decision problem. An instance of this decision problem is a triple (C, d, K), where C is a set of cities and d a distance function (as before), and K is a positive, real number. (C, d, K) is a yes-instance if there exists a tour of (C, d) that has total length at most K. The corresponding search problem is "given an instance (C, d, K), find a tour of (C, d) that has total length at most K if one exists."

Central to the study of computational complexity is the notion of an **NP-complete set**.

Definition 5 (NP-completeness) *The set S is **NP-complete** iff it is in NP and is "a hardest set in NP" in the following sense:*
For any set $T \in NP$, there is a polynomial-time computable function f such that

$$x \in T \Leftrightarrow f(x) \in S.$$

Note that the existence of f means that, if S were in P, then T would be in P. Equivalently, if S is NP-complete, and $S \in P$, then $P = NP$. The search and optimization problems that correspond to an NP-complete set S are called **NP-hard**; if they can be solved in polynomial time,[4] then $P = NP$.

SAT is an NP-complete set. So is the set of yes-instances of TSP described above. Literally, tens of thousands of problems that arise naturally in mathematics, science, engineering, economics, and other fields have been proven to be hard NP-hard. One of them is the "Combinatorial Auction Problem" that will be studied later in this course.

NP-completeness was first formulated by Steven Cook, [Coo71], who also showed (in the same paper) that SAT is NP-complete. The fact that natural NP-complete problems abound was demonstrated shortly thereafter by Richard Karp, [Kar72].

Extraordinary effort has gone into the quest for polynomial-time algorithms that solve NP-hard problems of practical importance, and yet no such algorithm has been found. Furthermore, it seems intuitively clear that verifying the correctness of a solution, should, in general, be easier than finding a solution, in other words, it seems clear that $P \subsetneq NP$. Surprisingly, no proof of this claim has been found.

Conjecture 1 (Fundamental Conjecture of Computer Science)

$$P \neq NP.$$

2.4 The Class $PSPACE$

After restricting attention to computational complexity as measured by $T(n)$, the last part of this lecture will focus on the concept of space-complexity as captured by $S(n)$. Analogous to the definition of polynomial-time solvability, the following definition captures the notion of **polynomial-space solvability**:

Definition 6 ***Polynomial-space solvable problems*** *are those solvable by algorithms whose space complexity satisfies*

$$S(n) = \mathcal{O}(n^c),$$

for $c \in \mathbb{N}$.

An enormously important class of examples of polynomial-space solvable problems is given by **quantified Boolean formulas**:

[4]This is a question of ongoing interest, but computer scientists do not believe that NP-hard problems can be solved in polynomial time.

Example 5 (Quantified Boolean Formula (QBF))
An instance is a quantified Boolean formula Ψ, i.e. an expression of the form

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 ... \exists x_{m-1} \forall x_m \qquad \mathcal{C}(x_1, ..., x_m),$$

where $\mathcal{C}$ is a quantifier-free CNF formula. A solution is simply "yes" if Ψ is a true quantified Boolean formula and "no" if Ψ is false. The language of yes-instances of this language is denoted by TQBF.

Definition 7 (The Class $PSPACE$)
If a decision-problem is solvable in polynomial time, and L is the set of yes-instances of that problem, then we say that

$$L \in LSPACE.$$

The QBF problem crystallizes two important properties of the complexity class $PSPACE$ and of space-bounded computation in general:

1. Space is a more powerful computational resource than time in the sense that an algorithm can reuse space but cannot reuse time.
2. There is an intimate relationship between the class $PSPACE$ and games. Each QBF instance can be interpreted as a perfect-information game between the "exists"-player, whose goal is to satisfy the formula $\mathcal{C}$, and the "forall"-player, whose goal is to falsify $\mathcal{C}$. The yes instances of QBF are the games in which the "exists"-player has a winning strategy.

These two properties will be outlined separately in the following two parts.

2.4.1 Space as a Powerful Computational Resource

To illustrate the fact that "space is a more powerful computational resource than time", we give a polynomial-space algorithm to decide QBF:
We represent the truth values F/T as bits $0/1$ and use the fact that $0 < 1$. Recall the notion of the *lexicographic order* of the Cartesian product $A \times B$ of two totally ordered sets A and B:

$$(a, b) < (a', b') \Leftrightarrow a < a' \text{ or } (a = a' \text{ and } b < b').$$

Repeated application of this construction imposes a total lexicographic order on $\{0,1\}^k$ for any k. For example, the lexicographic enumeration of $\{0,1\}^3$ is given by

$$000, 001, 010, 011, 100, 101, 110, 111.$$

If $\vec{\epsilon}$ is a bit string in $\{0,1\}^k - \{1\}^k$, let $next(\vec{\epsilon})$ be the string that follows $\vec{\epsilon}$ in the lexicographic enumeration; $next(\vec{\epsilon})$ is undefined and denoted by some symbol $\perp \notin \{0,1\}^k$.

```
$decide - QBF(\Psi)$
{
     OddBits $\equiv (\epsilon_1, \epsilon_3, ..., \epsilon_{m-1}) \leftarrow 0^{m/2}$
    $L_1$ : EvenBits $\equiv (\epsilon_2, \epsilon_4, ..., \epsilon_m) \leftarrow 0^{m/2}$
    $L_2$ : If $\mathcal{C}(\epsilon_1, \epsilon_2, ..., \epsilon_{m-1}, \epsilon_m) == 1$
    {
        If EvenBits== $1^{m/2}$, OUTPUT(YES);
        Else
        {
            EvenBits $\leftarrow$ next(EvenBits);
            Goto $L_2$;
        }
    }
    Else
    {
        If OddBits== $1^{m/2}$, OUTPUT(NO);
        Else
        {
            OddBits $\leftarrow$ next(OddBits);
            Goto $L_1$;
        }
    }
}
```

In the outer loop of decide-QBF, we consider in turn each assignment $\epsilon_1, \epsilon_3, ..., \epsilon_{m-1}$ to the existentially quantified Boolean variables $x_1, x_3, ..., x_{m-1}$. In the inner loop, we check whether for all possible assignments $\epsilon_2, \epsilon_4, ..., \epsilon_m$ to the universally quantified variables $x_2, x_4, ..., x_m$ the formula $\mathcal{C}$ is true. If the inner loop completes successfully for any assignment $\epsilon_1, \epsilon_3, ..., \epsilon_{m-1}$, we have proven that Ψ is true, and hence we halt and output YES. Otherwise, we have proven that no assignment has this property, and we output NO.

The crucial point about the space complexity of this procedure is that there is no need to allocate fresh space when the function "next()" is called. The same bits that were used to store $\epsilon_1, \epsilon_3, ..., \epsilon_{m-1}$ ($\epsilon_2, \epsilon_4, ...\epsilon_m$, respectively) can be re-used to store $next(\epsilon_1, \epsilon_3, \epsilon_{m-1})$ ($next(\epsilon_2, \epsilon_4, ...\epsilon_m)$, respectively). Time cannot be used the same fashion. Indeed, "decide-

QBF()" will run for time $\Omega(2^m)$ in the worst case.

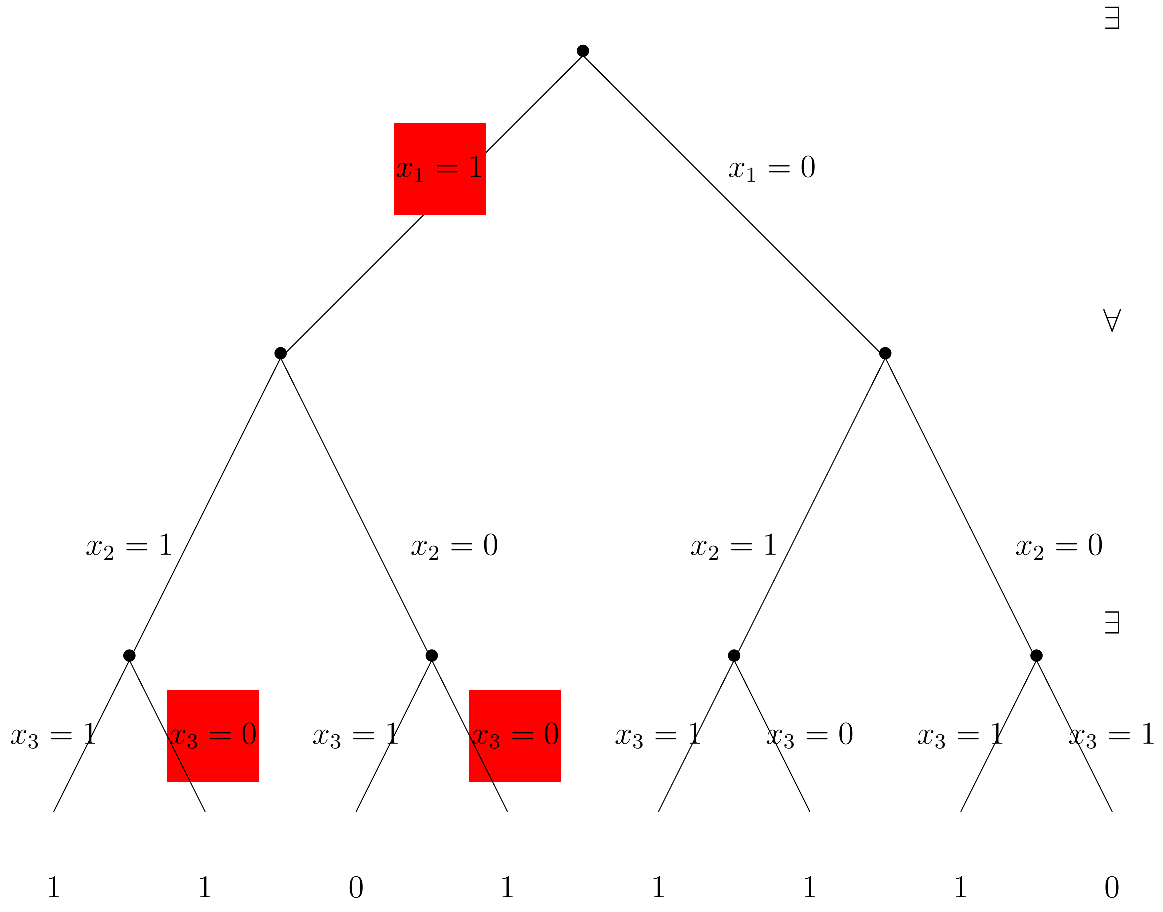
2.4.2 $PSPACE$ and Games

The second point crystallized by the QBF problem is the relationship between PSPACE and games. Each QBF instance can be interpreted as a perfect-information game between the "exists"-player, whose goal is to satisfy the formula $\mathcal{C}$, and the "forall"-player, whose goal is to falsify $\mathcal{C}$. Here, one builds upon the notion of extensive-form games, which have not been covered in the introductory lectures on game theory. In short, these games have an additional time-component. A player can wait and observe the other player's action before making a move by herself. Clearly, this is a generalization of the concept of simultaneous-move games in which all players act at the same time. The yes instances of QBF are the games in which the "exists"-player has a winning strategy. [5]

For example, consider the following yes instance Ψ of QBF:

$$\exists x_1 \forall x_2 \exists x_3 \qquad (x_1 \vee x_2 \vee x_3) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}).$$

[5]For the clarity of the exposition, the term "strategy" is used in an informal manner in this context. According to the strict economic definition, a strategy in an extensive-form like the one displayed below assigns an action to every node in the game-tree, no matter whether this node is reached or not.



∃ moves at the top level, ∀ at the second level, and finally ∃ at the third. ∃ can create a winning outcome for herself by playing $x_1 \leftarrow 1$ combined with $x_3 \leftarrow 0$ with the latter play being irrespective of player ∀'s choice.[6] This strategy is displayed in red in the above game tree.
Note that the game-tree has size exponential in m and, hence, cannot be built explicitly during the execution of a polynomial-space algorithm. The algorithm "decide-QBF()" shows that we need not build the whole tree: We can explore it one branch at a time, keeping track of whether we have found a winning strategy for ∃.

2.4.3 PSPACE-Completeness of QBF

QBF is in fact a PSPACE-complete set:

For any set $T \in PSPACE$, there is a polynomial-time computable function f such that, for all instances x

$$x \in T \Leftrightarrow f(x) \in QBF.$$

[6]There are other choices for player ∃ that create a desired outcome for her.

Many other perfect-information, polynomial-depth games correspond to PSPACE-complete sets as well, including polynomial-depth $n \times n$ GO and polynomial-depth $n \times n$-CHECKERS. Indeed, in a sense that is beyond the scope of this course, all PSPACE-complete sets represent games.

2.5 Relationship among Complexity Classes

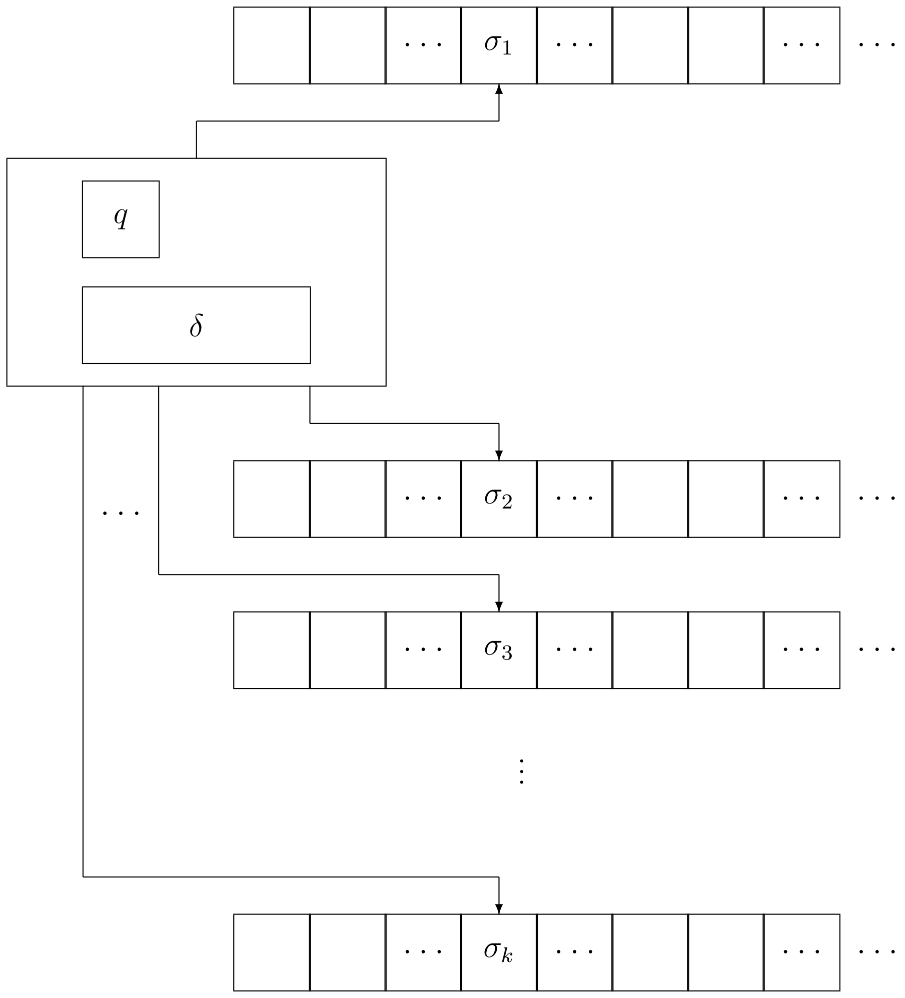
We have the following relationships among complexity classes:

$$P \subseteq NP \subseteq PSPACE.$$

Amazingly, it is not known whether either of these inclusions is proper; the seemingly bizarre possibility that $P = PSPACE$ has not been ruled out (but is, of course, believed to be false).

A Turing-Machine model of Computation

Deterministic k-tape Turing machine M.



There is one read-only **input tape** (on top) and $k-1$ read-write **work/output tapes**. M is a triple Γ, Q, δ that is defined as follows:

- Γ is the **tape alphabet**, a finite set of symbols. Assume $\square$ ("blank" symbol), $\triangleright$ ("start" symbol), 0 and 1 are four distinct elements of Γ.
- Q is the **state set**, a finite set of states that M's control register can be in. Assume q_{start} and q_{halt} are two distinct states in Q.
- δ is the **transition function**, a finite table that describes the rules (or program) by which M operates:

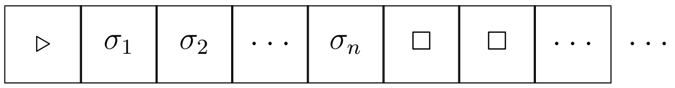
$$\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times (L, S, R)^k.$$

$\delta(q,(\sigma_1,...,\sigma_k)) = (q',(\sigma'_2,...,\sigma'_k),(z_1,...,z_k))$ means that, if M is in state q, and the read (or read/write) tape heads are pointing at the cells containing $\sigma_1,...,\sigma_k$, then the following "step" of the computation is performed:

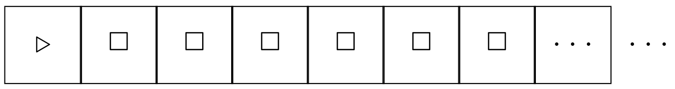
- the read/write tape symbols $\sigma_2,...,\sigma_k$ are replaced by $\sigma'_2,...,\sigma'_k$;
- tape head i moves left, stays in place or moves right, depending on whether z_i is in L,S or R;
- the control-register state is changed to q'.

When M starts its execution on input $x = \sigma_1,...,\sigma_n$, we have

- $q = q_{\text{start}}$
- input tape



- all other tapes



Meaning of q_{halt}:

$$\delta(q_{\text{halt}},(\sigma_1,...,\sigma_k)) = (q_{\text{halt}},(\sigma_2,...,\sigma_k),S^k) \qquad \forall(\sigma_1,...,\sigma_k).$$

Designate one of the read/write tapes as "the output tape".

Turing machine M "computes the function f", if for all $x \in \Gamma^*$ the execution of M on input x eventually reaches the state q_{halt}, and when it does, the contents of M's output tape is $f(x)$.

M "runs in time T" if for all n and all $x \in \Gamma^n$ M halts after at most $T(n)$ steps.

References

[CLR01] T.H. Cormen, C.E. Leiseron, and R.L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw Hill, 2nd edition, 2001. 1st Edition published in 1990.

[Coo71] S.A. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third Symposium on the Theory of Computing, Association of Computing Machinery*, pages 151–158, 1971.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[HS65] J. Hartmanis and R. Stearns. On the computational complexity of algorithms. *Transactions of the MAerican Mathematical Society*, 117:285–306, 1965.

[Kar72] R.M. Karp. *Reducibility among Combinatorial Problems, in Miller and Thatcher (eds.), Complexity of Computer Computations.* Plenum, 1972. pp. 85-103.

[Tur36a] A. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1936. article I/II.

[Tur36b] A. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 43:544–546, 1936. article II/II.

[Whi] H. Whitmore. Breaking the code. First performed in London's West End in 1986 and New York City's Broadway in 1987, both times with Derek Jacobi playing the part of Alan Turing.