



UNIVERSITÉ LIBRE DE BRUXELLES
MASTER EN SCIENCES INFORMATIQUE

Questions d'Examen

Computability and complexity
INFO-F408

Année académique 2017 – 2018

Contents

1	Introduction	2
1.1	(1.1.1) What is the relation between decision problems and languages ? . .	2
1.2	(1.1.2) Define finite automata and explain the difference between deterministic and nondeterministic finite automata. (cf. Sections 1.1 and 1.2)	3
2	Computability	4
2.1	(1.2.1.1) What is a Turing Machine?	4
2.2	(1.2.1.2) What is a Turing-recognizable language ? What is a Turing-decidable language ? Explain the difference between the two notions and why we use deciders to formalize the intuitive notion of algorithm.	4
2.3	(1.2.1.3) Define the notion of multitape Turing machine. Prove that every multitape Turing machine has an equivalent single tape Turing machine. .	5
2.4	(1.2.1.4) Prove that every nondeterministic Turing machine has an equivalent deterministic Turing machine.	6
2.5	(1.2.1.5) Explain what is an enumerator for a language. Prove that a language is Turing-recognizable if and only if it has an enumerator.	7
2.6	(1.2.1.6) What is the Church-Turing thesis? Give arguments in favor of it.	8
2.7	(1.2.2.1) Prove that the set of real numbers is uncountable	9
2.8	(1.2.2.2) Using a diagonalization argument, prove that some languages are not Turing-recognizable.	9
2.9	(1.2.2.3) Prove that if a language A and its complement \bar{A} are both Turing-recognisable, then A is Turing-decidable.	10
2.10	(1.2.2.4) Let A be a Turing-recognizable language consisting of descriptions of deciders. Prove that there exists a decidable language that is not decided by any decider described in A . (Exercise 4.28)	10
2.11	(1.2.2.5) Define the Halting Problem (A_{TM}) and prove it is undecidable . .	11
2.12	(1.2.2.6) Define the Post Correspondence Problem and prove it is undecidable.	13
2.13	(1.2.2.7) State Rice's Theorem and prove it. Give two problems that can be proved undecidable by applying Rice's Theorem.	15
2.14	(1.2.2.8) Give an example of a language that is not Turing-recognizable. . .	17
2.15	(1.2.2.9) Describe the $3x + 1$ problem (also known as Collatz problem/conjecture), and show that if A_{TM} were decidable, then there exists a Turing machine guaranteed to state the answer to the $3x + 1$ problem (Exercise 5.31).	17
3	Time complexity	19
3.1	(1.3.1) Prove that for $t(n) \geq n$, every $t(n)$ -time nondeterministic Turing machine has an equivalent $2^{O(t(n))}$ -time deterministic Turing machine.	19
3.2	(1.3.2) Give two definitions of the complexity class NP and prove that they are equivalent.	19
3.3	(1.3.3) Explain why all problems in NP can be decided in time $2^{O(n^k)}$ for some k	20
3.4	(1.3.4) Is it the case that $P \subseteq NP$? Is it the case that $NP \subseteq P$? Explain.	20
3.5	(1.3.5) Define the notion of <i>polynomial time (mapping) reduction</i> and show that if A is polynomial time reducible to B and $B \in P$, then $A \in P$	21

3.6	(1.3.6) Define the notion of NP-completeness.	21
3.7	(1.3.7) Prove the following two statements: 1) If B is NP-complete and $B \in P$ then $P = NP$. 2) If B is NP-complete and $B \leq_P C$ and $C \in NP$ then C is NP-complete.	21
3.8	(1.3.8) State the Cook-Levin Theorem and give a detailed outline of the proof	21
3.9	(1.3.9) Define the VERTEX-COVER problem and prove it is NP-complete.	24
3.10	(1.3.10) Define the HAMILTONIAN-PATH problem and prove it is NP-complete.	26
3.11	(1.3.11) Define the CLIQUE problem and prove it is NP-complete.	29
3.12	(1.3.12) Define the INDEPENDENT-SET problem and prove it is NP-complete. (Exercise)	30
3.13	(1.3.13) Define the 3-COLORING problem and prove it is NP-complete. (Exercise 7.27)	30
3.14	(1.3.14) Define the SUBSET-SUM problem and prove it is NP-complete. .	32
3.15	(1.3.15) Give a dynamic programming algorithm for SUBSET-SUM and explain why it does not imply $P = NP$	34
3.16	Show that if $P = NP$ you can determine a satisfying assignment to a 3-CNF formula in polynomial time, if one exists (Exercise 7.36).	34
3.17	Show that if $P = NP$ you can determine a maximum-size clique in a graph in polynomial time (Exercise 7.38).	35
4	Space complexity	36
4.1	(1.4.1) Define the space complexity of deterministic and nondeterministic Turing machines, and give an example of a problem that can be solved in $O(t(n))$ space, but for which we do not know any $O(t(n))$ -time algorithm, for some function $t(n)$	36
4.2	(1.4.2) State and prove Savitch's Theorem, and show it implies $PSPACE = NPSPACE$	36
4.3	(1.4.3) Define the notion of PSPACE-completeness	38
4.4	(1.4.4) Define the TQBF problem and prove it is in PSPACE	38
4.5	(1.4.5) Prove that TQBF is PSPACE-complete.	39
4.6	(1.4.6) Explain the interpretation of the TQBF problem as a game between an existential and a universal player.	43
4.7	(1.4.7) What do we know about the relations between the complexity classes P , NP , $PSPACE$, and $EXPTIME$? Explain.	44

Division

	C C	RTOS		C C	RTOS
Alexis	1.11 + 1.42	10 + 32	Rémy	1.221	2 + 21 + 42
Abde	/	11	Stan	1.222	3 + 22
Cédric G.	1.12 + 1.43	33	Tanguy	1.223	4 + 23
Cédric S.	1.21.1	12 + 34	Yannick	1.224	5 + 24
Denis	1.21.2	15 + 35	Yassin	1.225	6 + 28
Hakim	1.21.3	16 + 36	Youssef	1.226	7 + 29
HuSejin	1.21.4	17 + 37	Prabh	1.228	8 + 30
Jacky	1.21.5	18 + 38	Mi	1.31	9 + 31
Kais	1.21.6	19 + 39			
Nathan	1.32	1 + 20 + 40			

42 Questions sans réponses:

1.2.2.7 Mi	1.315	Tanguy	RTOS: 13	Cédric S.
1.2.2.9 Prabh	316	Yannick	14	Denis
1.3.3 Alexis	41	Yassin	25	Hakim
34 Abde	44	Youssef	26	HuSejin
35 Cédric G.	45	Abde	27	Jacky
36 Cédric S.	46	Abde	41	Kais
37 Denis	47	Abde		
38 Hakim				
39 HuSejin				
310 Jacky				
311 Kais				
312 Nathan				
313 Rémy				
314 Stan				

1 Introduction

1.1 (1.1.1) What is the relation between decision problems and languages ?

Problème de décision : l'ensemble des inputs pour lesquels la réponse est oui.

Autrement dit, une instance d'un problème de décision permet de savoir si oui ou non un input A fait partie de l'ensemble de solution. Il est important de faire la distinction entre une instance et un problème (le premier se réfère à un input spécifique alors que le second prend en référence l'ensemble des inputs possibles).

On définit le langage d'une machine M comme étant : $L(M) = A$. Si A est l'ensemble de tous les mots (strings) que la machine M accepte.

On peut dire que :

- M accepte A
- M reconnaît A

Ainsi, un langage est défini par un ensemble, et un problème de décision consiste à déterminer si un input appartient à un ensemble; on peut donc définir l'appartenance d'un mot à un langage comme un problème de décision.

1.2 (1.1.2) Define finite automata and explain the difference between deterministic and nondeterministic finite automata. (cf. Sections 1.1 and 1.2)

Un automate fini est défini comme 5-tuple $(Q, \Sigma, \delta, q_0, F)$ où :

1. Q est l'ensemble fini des états possibles
2. Σ est un alphabet, il s'agit de l'ensemble fini des symboles acceptés par l'automate.
3. δ est la fonction de transition

Dans le cas d'un automate déterministe, $\delta : Q \times \Sigma \rightarrow Q$. C'est-à-dire une fonction qui reçoit en argument un état et un symbole, et retourne l'état atteint.

Dans le cas d'un automate non-déterministe, $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$. Σ_ϵ est l'alphabet de l'automate auquel on ajoute le symbole vide ϵ . Ici, la fonction reçoit un état et un symbole en argument et retourne **un ensemble d'états atteignables**.

$\mathcal{P}(Q)$ est une notation qui signifie "l'ensemble de tous les sous-ensembles de Q ".

4. q_0 est l'état initial de l'automate,
5. F représente l'ensemble des états qui acceptent, et est un sous-ensemble de Q .

Une autre différence réside également dans la manière dont ces automates sont exécutés. Dans le cas d'un déterministe, l'exécution est déterminé. D'un état, en lisant un symbole, nous atteignons **au maximum 1** état. Tandis que dans le non-déterministe, lorsque plusieurs états sont atteignables depuis un état, l'automate va **se cloner et atteindre tous les états** (comme un `fork()`). Évidemment, en faisant cela, certains clones n'atteindront pas un état d'acceptance. Donc, dès lors qu'un automate non-déterministe ne peut plus avancer (il lit un symbole où aucun état n'est atteignable), le clone se tue. Un automate non-déterministe accepte **si au moins une exécution accepte** (c'est-à-dire qu'il y a **au moins une suite de transition qui mène à un état d'acceptance**).

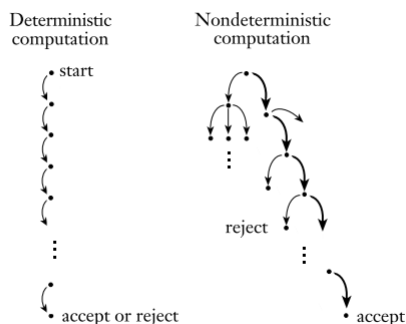


Figure 1: Deterministic and nondeterministic computations with an accepting branch

2 Computability

2.1 (1.2.1.1) What is a Turing Machine?

A *Turing machine (TM)* is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the blank symbol \sqcup ,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q' \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function, where Q' is Q without q_{accept} and q_{reject} ,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

The following list summarizes the differences between finite automata and Turing machines.

1. A Turing machine can both write on the tape and read from it.
2. The read-write head can move both to the left and to the right.
3. The tape is infinite.
4. **The special states for rejecting and accepting take effect immediately.**

2.2 (1.2.1.2) What is a Turing-recognizable language ? What is a Turing-decidable language ? Explain the difference between the two notions and why we use deciders to formalize the intuitive notion of algorithm.

A language L is *turing-recognizable* if there is a *TM* such that L is the set of accepted inputs.

I.e. \exists a *TM* such that :

- if $x \in L$: it halts in q_{accept}
- if $x \notin L$: it halts in q_{reject} or it loops for ever

A language L is *turing-decidable* if there is a *TM* that halts on all inputs such that L is the set of accepted inputs.

I.e. \exists a *TM* such that :

- if $x \in L$: it halts in q_{accept}
- if $x \notin L$: it halts in q_{reject}

2.3 (1.2.1.3) Define the notion of multitape Turing machine. Prove that every multitape Turing machine has an equivalent single tape Turing machine.

Une machine de Turing multi-ruban est comme une machine de Turing ordinaire mais avec plusieurs ruban. Chacun de ces rubans possède sa propre tête de lecture/écriture. Initialement, l'input apparaît sur le premier ruban et les autres sont vides. La fonction de transition est modifiée afin de permettre la lecture, l'écriture et le mouvement des têtes de lecture/écritures sur plusieurs ou tous les rubans simultanément. Formellement, la fonction est défini comme suit :

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k \quad (1)$$

où k est le nombre de ruban.

Exemple L'expression

$$\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, L, R, \dots, L) \quad (2)$$

signifie que si nous sommes dans l'état q_i et que les têtes de lecture/écriture des rubans 1 à k lisent les symboles a_1 à a_k , alors la machine atteint l'état q_j , écrit les symboles b_1 à b_k sur les rubans correspondant et bouge les têtes de lecture/écriture soit à gauche (L), soit à droite (R), soit reste sur place (S) sur les rubans correspondants.

TM équivalentes signigie que les T.M. **reconnaissent** le même langage.

Preuve que les multi-rubans ont un équivalent simple ruban Soit une machine de Turing multi-ruban M et une autre mono-ruban S . Nous devons convertir M en un équivalent mono-ruban. L'idée principale est de montrer comment simuler M avec S .

Disons que M possède k rubans. Alors S simule l'effet de k rubans en sauvegardant leur information sur un seul ruban. Pour ce faire, la machine utilise le nouveau symbole $\#$ comme délimiteur pour séparer les contenus de chaque ruban. En plus du contenu de ces rubans, S doit garder une trace de la localisation des têtes. Elle le fait via l'alphabet du ruban (on rajoute des symboles). Un symbole avec un point au-dessus de celui-ci marque l'emplacement de la tête sur ce ruban. L'image suivante montre comment un mono-ruban peut représenter trois rubans.

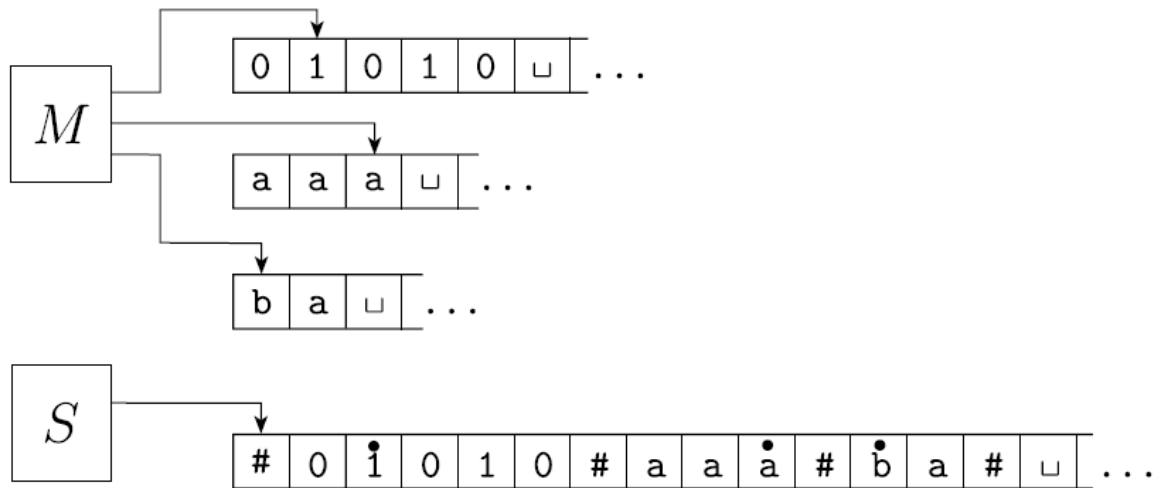


Figure 2: Representing three tapes with one

2.4 (1.2.1.4) Prove that every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Proof idea. We can simulate any nondeterministic Turing machine (TM) N with a deterministic TM D . The idea behind the simulation is to have D try all possible branches of N 's nondeterministic computation. If D ever finds the accept state on one of these branches, D accepts. Otherwise, D 's simulation will not terminate. To avoid going forever down on an infinite branch, we explore the tree in a breadth-first manner.

Proof. Let the simulating deterministic TM D have three tapes. We know that it is equivalent to have a single tape. We will design the TM to use the tapes as follows: The tape 1 (input tape) always contains the input string. Tape 2 (simulation tape) maintains a copy of N 's tape on some branch of its nondeterministic computation. Tape 3 (address tape) keeps track of D 's location in N 's nondeterministic computation tree.

Let's first consider the tape 3: Every node in the tree has at most b children, with b the size of the largest set of possible choices given by N 's transition function. To every node we assign an address that is a string over $\Sigma_b = \{1, 2, \dots, b\}$. We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child. (Some address may be invalid, that is they does not correspond to any node.) This way we can store on the tape 3 the branch of N 's computation.

So D can be described as follow:

1. Initially, tape 1 contains the input w , and tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2 and initialize the string on tape 3 to be ϵ .
3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N , consult the next symbol on tape 3 to determine which choice to make among those allowed by N 's transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by

going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, accept the input.

4. Replace the string on tape 3 with the next string in the string ordering. Simulate the next branch of N 's computation by going to stage 2.

2.5 (1.2.1.5) Explain what is an enumerator for a language. Prove that a language is Turing-recognizable if and only if it has an enumerator.

Enumerator Un énumérateur E peut être vu comme une Turing machine (TM) avec une imprimante. Cette imprimante est en fait le ruban de sortie (output-tape) de la définition formelle suivante:

An *Enumerator* is almost like a 2-tapes TM, *i.e.* a 8-tuple, $(Q, \Sigma, \Gamma_i, \Gamma_o, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the blank symbol \sqcup ,
3. Γ_i is the first, or input-, tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. Γ_o is the second, or output-, tape alphabet, where $\{\sqcup, \#\} \subseteq \Gamma$ and $\Sigma \subseteq \Gamma$,
5. $\delta : Q' \times \Gamma \rightarrow Q \times \Gamma^2 \times \{L, R\}^2$ is the transition function, where Q' is Q without q_{accept} and q_{reject} ,
6. $q_0 \in Q$ is the start state,
7. $q_{accept} \in Q$ is the accept state, and
8. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

For a 2-tape TM, the transition function is in the form $\delta : Q' \times \Gamma^2 \rightarrow Q \times \Gamma^2 \times \{L, R\}^2$, but an enumerator does not read on its second tape.

Enumerator for a language An enumerator E for a language L is an enumerator which starts with its two tapes empty and prints on its second tape (that I called the *output-tape*) the (possibly infinite) set of strings belonging to L . Strings are separated by the $\#$ separator character.

E peut générer les chaînes de caractères du langage dans n'importe quel ordre, même avec répétitions.

Théorème Un langage est TM-reconnaissable si et seulement si un énumérateur l'énumère (le liste, le print...).

Preuve Tout d'abord, montrons que si nous avons un énumérateur E qui énumère un langage A , alors une machine de Turing M reconnaît A . La machine de Turing W travaille comme suit.

$M =$ "Sur l'input w :

1. Exécuter E . À chaque fois que E imprime une chaîne de caractère, la comparer avec w .
2. Si w apparaît dans la sortie de E , alors on *accepte*."

Clairement, M accepte également les chaînes de caractères qui apparaissent dans la liste de E .

Maintenant, faisons la procédure dans un autre sens. Si une machine de Turing M reconnaît un langage A , nous pouvons construire l'énumérateur E suivant pour A . Mettons que $s_1, s_2, s_3 \dots$ soit une liste de toutes les chaînes de caractères dans Σ^* .

$E =$ "Ignore l'input.

1. Répéter ce qui suit pour $i = 1, 2, 3 \dots$:
2. Exécuter M pour i étapes sur chaque input $s_1, s_2 \dots s_i$.
3. Si une des exécutions accepte, imprimer la chaîne de caractère s_j correspondante."

2.6 (1.2.1.6) What is the Church-Turing thesis? Give arguments in favor of it.

The thesis is : Algorithmically computable = computable by a TM. So basically, if anything is said computable, then it is computable by a TM, and vice-versa. If a problem is undecidable, there is not any TM that can decide it.

In the 30's, we wanted to define clearly the meaning of being "computable". At that time, there were also several variations on the TMS and we were not sure about their levels of power. For example:

- One tape or many
- Infinite tapes on both sides
- Tiny alphabet $\{0,1\}$ or not
- Accepts the nondeterminism or not

The conclusion is that all these possible variations are equivalent in computing capability.

Two arguments going in that way are that :

- Every multitape TM has an equivalent single-tape TM : we can simulate k tapes on one tape, by augmenting the tape alphabet (Γ) with an element to separate the contents of every tape and having a specific symbol for every $x \in \Gamma$ for the heads.
- Every NTM has an equivalent DTM: We could represent all the execution branches of a NTM as a tree. A NTM can be simulated on a 3 tape DTM, which goes through the whole NTM tree in breadth first. The first tape maintains the input, the second tape simulates the tape of the NTM, and the third tape keeps track of the possible computations.

2.7 (1.2.2.1) Prove that the set of real numbers is uncountable

Preuve par contradiction: supposons que \mathbb{R} soit dénombrable. On a donc une liste qui fait correspondre tous les nombres naturels (\mathbb{N}) à un nombre présent dans \mathbb{R} . On a donc une liste dénombrable où chaque élément i (appartenant à \mathbb{N} donc) fait correspondre un nombre j (appartenant lui à \mathbb{R}).

Un exemple concrèt est représenté dans le tableau 1.

Construisons un $x \in [0, 1)$ tel que la i^{eme} décimale de x soit égal à la i^{eme} décimale

1	0, 3 1415926535
2	1,0 0 0000000000
3	22,12 3 12312312
4	323,010 1 0101010
5	4,1502 6 535010
6	...

Table 1: Exemple du point 1.2.2.1

incrémenté de 1 du i^{eme} nombre présent dans cette ensemble dénombrable (dans l'exemple présenté au tableau 1, on aurait donc $x = 0, 41427\dots$).

Par construction, ce nombre x ne peut pas appartenir à notre ensemble dériavable, pourtant il appartient bien à \mathbb{R} .

2.8 (1.2.2.2) Using a diagonalization argument, prove that some languages are not Turing-recognizable.

(From reference, 2d edition, corollary 4.18)

To show that the set of all Turing machines is countable we first observe that the set of all strings Σ^* is countable, for any alphabet Σ . With only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, length 1, and so on.

The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$. If we simply omit those strings that are not legal encodings of Turing machines, we can obtain a list of all Turing machines.

To show that the set of all languages is uncountable, we first observe that the set of all infinite binary sequences is uncountable. An *infinite binary sequence* is an unending sequence of 0s and 1s. Let B be the set of all infinite binary sequences. We can show that B is uncountable by using a proof by *diagonalization* (see previous question).

Let Λ be the set of all languages over alphabet Σ . We show that Λ is uncountable by giving a correspondence with B , thus showing that the two sets are the same size. Let $\Sigma^* = \{s_1, s_2, s_3, \dots\}$. Each language $A \in \Lambda$ has a unique sequence in B . The i^{th} bit of that sequence is a 1 if $s_i \in A$ and is a 0 otherwise, which is called the *characteristic sequence* of A . For example, if A were the language of all strings starting with a 0 over the alphabet $\{0, 1\}$, its characteristic sequence Ξ_A would be :

The function $f : L \rightarrow B$, where $f(A)$ equals the characteristic sequence of A , is a correspondence (one-to-one). Therefore, as B is uncountable, L is uncountable as well.

$$\begin{array}{lcl}
\Sigma^* & = & \{ \varepsilon, 0, 1, 00, 01, 10, 11, 000 \dots \} \\
A & = & \{ 0, 00, 01, 000 \dots \} \\
\Xi_A & = & \{ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \dots \}
\end{array}$$

Thus we have shown that the set of all languages cannot be put into a correspondence with the set of all Turing machines. We conclude that some languages are not recognised by any Turing machine.

2.9 (1.2.2.3) Prove that if a language A and its complement \bar{A} are both Turing-recognisable, then A is Turing-decidable.

Théorème Un langage A est décidable si et seulement si A et son complément sont Turing-reconnaissables. Autrement dit, un langage est décidable seulement lorsque celui-ci et son complément sont tous les deux Turing-reconnaissables.

Preuve Nous avons deux choses à prouver. Premièrement, si A est décidable, nous pouvons aisément voir qu' A et son complément sont Turing-reconnaissables. N'importe quel langage décidable est Turing-reconnaissable, et le complément d'un décidable est aussi décidable. Deuxièmement, si A et \bar{A} sont Turing-reconnaissables, nous créons M_1 , un reconnaiseur pour A , et M_2 , un reconnaiseur pour \bar{A} . La machine de Turing M suivante est un décideur pour A :

$M =$ "sur l'input ω :

1. Exécuter M_1 et M_2 en parallèle sur l'input ω
2. Si M_1 accepte, on accepte. Si M_2 accepte, on rejette"

Exécuter deux machines en parrallèle signifie que M possède deux rubans: l'un pour simuler M_1 et l'autre pour simuler M_2 . Dans ce cas, M tourne en simulant une étape de chaque machine, et cela se poursuit jusqu'à ce que l'un des deux accepte.

Maintenant, nous montrons que M décide A . Chaque chaine de caractères ω est soit dans A , soit dans \bar{A} . Par conséquent, M_1 ou M_2 doit accepter ω . Etant donné que M s'arrête à chaque fois que A ou \bar{A} accepte, M s'arrête toujours et est donc un décideur. De plus, M accepte toutes les chaines de caractères de A et rejette toutes celles qui ne sont pas dans A . Donc, M est un décideur pour A , et A est décidable.

2.10 (1.2.2.4) Let A be a Turing-recognizable language consisting of descriptions of deciders. Prove that there exists a decidable language that is not decided by any decider described in A . (Exercise 4.28)

1. Create a new language, that, by contruction, is different of all the others. This can easily be done by diagonalization.

	w1	w2	w3		wi	wi+1	...
La	1	0	1	1	0	1	
Lb	0	0	0	0	0	0	
Lc	1	0	1	0	1	0	
...							
Li	0	1	0	...			

Figure 3: Diagonalization of language of deciders

By construction, $L_i \notin A$.

2.11 (1.2.2.5) Define the Halting Problem (A_{TM}) and prove it is undecidable

Le problème de l'arrêt ou (A_{TM}) est un problème qui consiste à déterminer, à partir d'une description d'un programme et de son input, si ledit programme finira par s'arrêter ou pas. Alan Turing a prouvé qu'un algorithme permettant de résoudre le problème de l'arrêt pour toutes les paires $\langle \text{Programme} - \text{son Input} \rangle$ **ne peut pas** exister. Il s'agit d'un problème indécidable pour les machines de Turing.

Il est à noter que A_{TM} est *Turing-reconnaissable* (*Turing recognizable*). Donc, le théorème ci-dessus nous montre que les *reconnaisseur* (*recognizers*) sont plus puissants que les décideurs. Exiger qu'une machine de Turing ait la capacité de s'arrêter sur toutes les input va restreindre les différentes sortes de langage que la machine peut reconnaître. Mettons qu'on possède une machine de Turing U qui reconnaisse A_{TM} . Cette machine devra donc s'exécuter comme suit:

$U =$ "Sur l'input $\langle M, w \rangle$, où M est une machine de Turing et w est une chaîne de caractère:

1. Simuler M sur l'entrée w .
2. Si jamais M entre dans son état d'acceptance, U accepte; si jamais M entre dans son état de rejet, U rejette."

À noter également que U va boucler éternellement sur l'input $\langle M, w \rangle$ si jamais M boucle sur son input w . C'est donc pourquoi U ne décide pas A_{TM} . Cette machine est appelée "machine de Turing universelle" (initialement proposée par Alan Turing). Elle est appelée "universelle" parce qu'elle peut simuler n'importe quelle autre machine de Turing à partir de la description de celle-ci. La machine de Turing universelle a joué un rôle important dans les débuts du développement des *Ordinateur à programme enregistré*¹.

¹Un ordinateur à programme enregistré (ou calculateur à programme enregistré; en anglais stored-program computer) est un ordinateur qui enregistre les instructions des programmes qu'il exécute dans

Mais tout ceci n'est qu'anecdotique et ne répond pas à la deuxième partie de la question qui est de prouver l'indécidabilité d' A_{TM} .

Théorème : A_{TM} est indécidable

Preuve que A_{TM} est un problème indécidable Définissons d'abord formellement A_{TM} :

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ est une machine de Turing et } M \text{ accepte } w \} \quad (3)$$

signifie que A_{TM} correspond à l'ensemble des machines de Turing qui acceptent leur entrée.

Nous supposons que A_{TM} soit décidable et obtenons une contradiction. Donc, on crée une machine de Turing H qui est un décideur pour A_{TM} . Sur l'entrée $\langle M, w \rangle$, où M est une machine de Turing et w est une chaîne de caractère, H s'arrêtera et acceptera si jamais M accepte w . De plus, H s'arrêtera également et rejettera si jamais M rejette w . En d'autres termes, nous supposons que H est une machine de Turing où :

$$H(\langle M, w \rangle) = \begin{cases} \text{accepte si } M \text{ accepte } w \\ \text{refuse si } M \text{ refuse } w \end{cases} \quad (4)$$

Maintenant, construisons encore une nouvelle machine de Turing D avec H comme sous-routine. Cette nouvelle machine de Turing appellera H pour déterminer ce que M fait lorsque l'entrée de M est sa propre description $\langle M \rangle$. Une fois que D a déterminé cette information, elle fait l'opposé. C'est-à-dire que D rejettera si M accepte et acceptera si M rejette.

Voici la description de D :

$D =$ "Sur l'entrée $\langle M \rangle$, où M est une machine de Turing :

1. Exécuter H sur l'entrée $\langle M, \langle M \rangle \rangle$.
2. Faire l'opposé de ce que H fait. C'est-à-dire, si H accepte, *rejeter*; et si H rejette, *accepter*."

Il ne faut pas être confus par la notion d'exécuter une machine avec sa propre description comme entrée. C'est similaire à exécuter un programme avec lui-même comme entrée, chose qui arrive parfois en pratique. Par exemple, un compilateur est un programme qui traduit d'autres programmes. Un compilateur pour le langage *Python* peut lui-même être écrit en *Python*, donc exécuter un programme sur lui-même peut avoir du sens.

sa mémoire vive. Un ordinateur avec une architecture de von Neumann enregistre ses données et ses instructions dans la même mémoire; un ordinateur avec une architecture Harvard enregistre ses données et ses instructions dans des mémoires séparées

En résumé, $D(< M >)$ accepte si M rejette M , et rejette si M accepte $< M >$. Que se passe-t-il lorsque nous exécutons D avec sa propre description $< D >$ en input ? Dans ce cas, nous avons que $D(< D >)$ *accepte* si D rejette $< D >$ et *rejette* si D accepte $< D >$. Peu importe ce que fait D , la machine est obligé de retourner l'opposé, ce qui est contradictoire. Du coup, ni H , ni D ne peuvent exister. CQMD (Ce que Mourad démontre lol.)

Résumons les étapes de cette preuve. Premièrement, nous supposons une machine de Turing H qui décide A_{TM} . Nous utilisons H pour construire une autre machine de Turing D qui prend une description de machine $< M >$ en input. Finalement, nous exécutons D avec sa propre description $< D >$ en input. La machine fera les actions suivantes, avec la dernière ligne qui est une contradiction :

- H accepte $< M, w >$ exactement lorsque M accepte w .
- D rejette $< M >$ exactement lorsque M accepte $< M >$.
- D rejette $< D >$ exactement lorsque D accepte $< D >$.

2.12 (1.2.2.6) Define the Post Correspondence Problem and prove it is undecidable.

Let us define first a *domino* like an ordered pair of string. The first string is called the *top string* and the second the *bottom string*. We denote the domino with a as top string and ab as bottom string by $\left[\begin{smallmatrix} a \\ ab \end{smallmatrix} \right]$.

A list of dominos is called a *match* if the concatenation of the top-strings of the dominos of this list is the same as the concatenation of the bottom-strings.

The Post correspondence problem (Problème de correspondance de post) is to determine whether a collection of dominos has a match.

Formally, an instance of the Post correspondence problem (PCP) is a collection of dominos:

$$P = \left\{ \left[\begin{smallmatrix} t_1 \\ b_1 \end{smallmatrix} \right], \left[\begin{smallmatrix} t_2 \\ b_2 \end{smallmatrix} \right], \dots, \left[\begin{smallmatrix} t_k \\ b_k \end{smallmatrix} \right] \right\}$$

and a match is a sequence i_1, i_2, \dots, i_l , where $t_{i_1}t_{i_2} \dots t_{i_l} = b_{i_1}b_{i_2} \dots b_{i_l}$. The problem is to determine whether P has a match. One can express the problem has a language as follows:

$$PCP = \{ \langle P \rangle \mid P \text{ is an instance of the Post correspondence problem with a match} \}$$

We define also the modified Post correspondence problem (MPCP) as the Post correspondence problem where we require that a match begins with the first domino.

$$MPCP = \{ \langle P \rangle \mid P \text{ is an instance of the Post correspondence problem with a match that starts with the first domino} \}$$

Proof idea. We will perform a reduction from A_{TM} , the problem of whether a Turing machine accepts a given input, via accepting computation histories for M on w . We show that from any TM M and input w we can construct an instance P where a match is an accepting computation history for M on w .

Proof. Let R be a TM that decide the PCP. We construct S deciding A_{TM} .

Given a TM M and a string w , S first constructs an instance P of the PCP s.t. P has a match if and only if M accepts w . Then it runs R to determines if P has a match and so decides if M accepts w .

To describe the construction of P we first realize the construction of an instance P' of the MPCP then derive P from P' .

1. $\left[\frac{\#}{\#q_0w_1w_2\cdots w_n\#} \right]$ is the first domino of P' (i.e. $t_1 = \#$ and $b_1 = \#q_0w_1w_2\cdots w_n\#$).
2. For every $a, b \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$ and $\delta(q, a) = (r, b, R)$, put $\left[\frac{qa}{br} \right]$ into P' to be able to simulate a motion of the head to the right.
3. For every $a, b, c \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$ and $\delta(q, a) = (r, b, L)$, put $\left[\frac{cqa}{rcb} \right]$ into P' to be able to simulate a motion of the head to the left.
4. For every $a \in \Gamma$ put $\left[\frac{a}{a} \right]$ into P' to handle the tape cells not adjacent to the head.
5. Put $\left[\frac{\#}{\#} \right]$ into P' to allows to copy the $\#$ symbol that marks the separation of configurations.
6. Put $\left[\frac{\#}{\#} \right]$ into P' to be able to add a blank symbol at the end of the configuration to simulate the infinitely many blanks to the right that are suppressed when we write the configuration.
7. For every $a \in \Gamma$ put $\left[\frac{aq_{\text{accept}}}{q_{\text{accept}}} \right]$ and $\left[\frac{q_{\text{accept}}a}{q_{\text{accept}}} \right]$ into P' . These are to allow the top-string to “catch up” the bottom string when the Turing machine stops on its accepting state.
8. Finally $\left[\frac{q_{\text{accept}}\#\#}{\#} \right]$ allows to complete the match.

See the Sipser’s book to have an example of correspondence between P' and the A_{TM} instance with M and w .

Let $u = u_1u_2u_3\cdots u_n$ be any string of length n . We define the three following strings:

$$\begin{aligned} \star u &= \star u_1 \star u_2 \cdots \star u_n \\ u \star &= u_1 \star u_2 \star \cdots \star u_n \star \\ \star u \star &= \star u_1 \star u_2 \star \cdots \star u_n \star \end{aligned}$$

To convert P' to an instance P of PCP, with

$$P' = \left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \dots, \left[\frac{t_k}{b_k} \right] \right\}$$

we let P be the collection

$$P = \left\{ \left[\frac{\star t_1}{\star b_1 \star} \right], \left[\frac{\star t_1}{b_1 \star} \right], \left[\frac{\star t_2}{b_2 \star} \right], \dots, \left[\frac{\star t_k}{b_k \star} \right], \left[\frac{\star \diamond}{\diamond} \right] \right\}$$

This way, the only domino that could possibly starts a match is the first one. Otherwise the presence of \star s does not affect possible matches. The last domino allows the top to add an extra \star at the end of the match.

2.13 (1.2.2.7) State Rice's Theorem and prove it. Give two problems that can be proved undecidable by applying Rice's Theorem.

Informellement, le théorème de Rice dit que le problème consistant à déterminer si une machine de Turing possède telle propriété non-triviale est un problème indécidable.

Théorème Formellement, soit P le langage consistant en la description de plusieurs machines de Turing où P remplit deux conditions :

1. P est une propriété qui n'est pas triviale. P contient certaines mais pas toutes les descriptions d'une machine de Turing.
2. P fait partie des propriétés du langage de la machine de Turing (lorsque $L(M_1) = L(M_2)$, nous avons que la description de la machine de Turing $\langle M_1 \rangle \in P$ ssi $\langle M_2 \rangle \in P$).

Prouver que P est indécidable.

Preuve Supposons que P est un langage décidable satisfaisant une propriété et supposons par contradiction que R_P est une machine de Turing qui décide P .

Nous montrons comment décider A_{TM} en utilisant R_P en construisant une machine de Turing S . Tout d'abord, posons T_\emptyset , une TM qui rejette toujours. Donc, $L(T_\emptyset) = \emptyset$ (*Aucun langage n'est accepté par T_\emptyset*). Il est permis de supposer que $L(T_\emptyset) \notin P$ parce qu'il est possible de procéder avec \overline{P} au lieu de P si $L(T_\emptyset) \in P$.

Comme P n'est pas trivial, il existe une machine de Turing T qui satisfait la propriété p (par exemple: $p = \text{"est une fonction qui élève l'input au carré"}$). Donc $L(T) \in P$.

On conçoit M_w de sorte que S appelle d'abord M sur w (une TM) construite de toutes pièces (et qui n'est pas utilisée), puis qui appelle T sur l'input donné et renvoie la même chose:

Pour $\langle M, w \rangle$ donnés, on construit la machine de Turing M_w suivante:

$M_w =$ " sur l'entrée x :

- Simuler M sur w . Si M s'arrête et rejette, *on rejette*. Sinon, étape II.
- Simuler T sur x . Si T accepte, *on accepte*."

On construit ensuite S :

"Sur l'entrée $\langle M, w \rangle$:

Construction de M_w .

Utiliser R_P pour déterminer si $L(M_w) \in P$. Si oui, *on accepte*; Sinon, *on rejette*."

Epilogue Etant donné que R_P est un décideur, R_P va s'arrêter quoiqu'il arrive. Or, quand R_P décide si M_w a la propriété ou non, il va en même temps simuler $M(w)$. Or, M_w n'a la propriété que si $M(w)$ s'arrête, car T ne peut s'exécuter que si $M(w)$ s'arrête.

Par conséquent, si M_w a la propriété, cela signifie que $M(w)$ s'arrête. R_P est capable de déterminer si une TM M accepte sur un input w donné, ce qui est la définition même de A_TM . Donc, R_P **ne peut pas** être un décideur.

Formellement La machine de Turing M_w simule T si M accepte w . Ainsi $L(M_w) = L(T)$ si M accepte w ; sinon $L(M_w) = \emptyset$. Conclusion, $L(M_w) \in P$ ssi M accepte w .

Exemple 1 Soit $X = \{ \langle M \rangle \mid M \text{ est une machine de Turing qui accepte } w^R \text{ lorsqu'elle accepte } w \}$, où $w = w_1w_2w_3...w_n$ et $w^R = w_nw_{n-1}...w_1$. (L'inverse de w en somme)

X est indécidable : (Par le théorème de Rice) Soit P une propriété non-triviale tel que $P = \{ A \mid w \in A \Leftrightarrow w^R \in A \}$ (P est défini comme l'ensemble des langages où chaque mot de A possède son inverse dans A également.). Nous devons montrer que P est une propriété non triviale.

Par exemple, $\{01\} \notin P$, donc $P \neq \mathcal{P}(\Sigma^*)$; et $\{1\} \in P$ donc $P \neq \emptyset$.

Exemple 2 (Sipser 5.30 p 244) Soit $INFINITE_{TM} = \{ \langle M \rangle \mid M \text{ est une machine de Turing et } L(M) \text{ est un langage infini} \}$, $INFINITE_{TM}$ est un langage de descriptions de machines de Turing. Ce langage satisfait les deux conditions du théorème de Rice. D'abord, c'est non trivial car certaines machines de Turing possèdent des langages infinis et d'autres non. Ensuite, cela dépend uniquement du langage. En effet, si deux machines de Turing reconnaissent le même langage, soit elles ont toutes les deux des descriptions dans $INFINITE_{TM}$, soit aucune des deux n'en a. Par conséquent, le théorème de Rice implique que $INFINITE_{TM}$ est indécidable.

Exemple 3 Soit $ALL_{TM} = \{ \langle M \rangle \mid M \text{ est une machine de Turing et } L(M) = \Sigma^* \}$, ALL_{TM} est indécidable selon le théorème de Rice. En effet, la propriété P dans notre cas est $L(M) = \Sigma^*$ et celle-ci est non triviale car il existe au moins une TM qui appartient à ALL_{TM} (*Une machine qui accepte tout*) et il existe également au moins une TM qui n'appartient pas à ALL_{TM} (*Une machine qui rejette tout*). De plus, P est effectivement une propriété du langage des TM car pour toute paire de machines M_1 et M_2 , telles que $L(M_1) = L(M_2)$,

$$\begin{aligned} \langle M_1 \rangle \in ALL_{TM} &\Leftrightarrow L(M_1) = \Sigma^* \\ &\Leftrightarrow L(M_2) = \Sigma^* \\ &\Leftrightarrow \langle M_2 \rangle \in ALL_{TM} \end{aligned}$$

2.14 (1.2.2.8) Give an example of a language that is not Turing-recognizable.

Le langage $\overline{A_{TM}}$ est le complément du langage A_{TM} , défini par :

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ est une machine de Turing et } M \text{ accepte } w \} \quad (5)$$

ce qui signifie que A_{TM} correspond à l'ensemble des représentations des machines de Turing et de leurs entrées qu'elles acceptent. Nous allons montrer que $\overline{A_{TM}}$ n'est pas Turing-reconnaissable.

Preuve Nous savons que A_{TM} est Turing-reconnaissable, il suffit de simuler l'exécution de M sur w pour vérifier que $\langle M, w \rangle$ répond aux critères. Nous avons également prouvé que si un langage A et son complément sont tous les deux Turing-reconnaissables, alors A est décidable. Finalement nous avons prouvé que A_{TM} n'est pas décidable. Donc si son complément $\overline{A_{TM}}$ était également Turing-reconnaissable, alors A_{TM} serait décidable, ce qui serait une contradiction.

Ainsi, $\overline{A_{TM}}$ n'est pas Turing-reconnaissable.

2.15 (1.2.2.9) Describe the $3x + 1$ problem (also known as Collatz problem/conjecture), and show that if A_{TM} were decidable, then there exists a Turing machine guaranteed to state the answer to the $3x + 1$ problem (Exercise 5.31).

Conjecture de Collatz: Soit

$$f : \mathbb{N}_1 \rightarrow \mathbb{N}_1 : f(x) = \begin{cases} 3x + 1 & \text{si } x \text{ impair} \\ x/2 & \text{si } x \text{ pair} \end{cases}$$

pour tout nombre entier x . Avec \mathbb{N}_1 l'ensemble de tout les nombres naturels sauf 0. Pour tout $x \in \mathbb{N}_1$ l'ensemble $\{x, f(x), f(f(x)), \dots\}$ contient 1.

Si on commence avec un entier x et itérons f , nous obtenons une séquence $(x, f(x), f(f(x)), \dots)$. On arrête si on tombe sur 1. Par exemple, avec $x=17$, nous avons la séquence 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Plusieurs tests on montré que des entiers entre 1 et un grand nombre positif donnent une séquence se terminant par 1. Mais la question de savoir si c'est le cas pour tout les entiers positif plus grand que 1 restent non-résolue; c'est ce qu'on appelle le problème du $3x+1$.

Nous voulons montrer que s'il existait un décideur H pour A_{TM} , nous aurions pu créer une TM qui prouverait la conjecture. Soit une TM C qui reconnaît le sous-ensemble de \mathbb{N}_1 pour lequel la conjecture est respectée: $\{x \in \mathbb{N}_1 : 1 \in \{x, f(x), f(f(x)), \dots\}\}$

$C = \text{en input } \langle x \rangle :$

1. While $x \neq 1$: $x \leftarrow f(x)$

2. 1 $\{1 \in \{x, f(x), f(f(x)), \dots\}\}$ alors accepter.

Afin de combiner cette TM avec les autres, définissons une TM C_{Σ^*} qui maintient la conjecture ssi $L(C_{\Sigma^*}) = \Sigma^*$. Ici $\Sigma = \{0, 1\}$.

C_{Σ^*} = en input $\langle w \rangle$:

1. If $w \notin \{1, 10, 11, 100, 101, 110, 111, \dots\}$, alors w n'est pas la représentation binaire $\langle x \rangle$ du nombre $x \in \mathbb{N}_1$, accepter.
2. While $x \neq 1 : x \leftarrow f(x)$
3. 1 $\{1 \in \{x, f(x), f(f(x)), \dots\}\}$ alors accepter.

En utilisant H, nous créons $R_{\neq \Sigma^*}$, un reconnaisseur pour le langage $L(R_{\neq \Sigma^*}) = \langle M \rangle : L(M) \neq \Sigma^*$.

$R_{\neq \Sigma^*}$ = en input $\langle M \rangle$:

1. Pour tout $w \in \Sigma^*$ en ordre lexicographique:
 - 1.1 Run H sur $\langle M, w \rangle$.
 - 1.2 If H rejette, accepter.

Finalement, on run H sur $\langle R_{\neq \Sigma^*}, \langle C_{\Sigma^*} \rangle \rangle$ pour vérifier que la conjecture est vraie. Si H n'accepte pas alors $L(C_{\Sigma^*}) = \Sigma^*$ et la conjecture est maintenue, sinon elle ne l'est pas.

3 Time complexity

3.1 (1.3.1) Prove that for $t(n) \geq n$, every $t(n)$ -time nondeterministic Turing machine has an equivalent $2^{O(t(n))}$ -time deterministic Turing machine.

Théorème Soit $t(n)$ une fonction où $t(n) \geq n$. Alors, toutes les machines de Turing non-déterministes mono-ruban s'exécutant en un temps de l'ordre de $t(n)$ possèdent un équivalent déterministe s'exécutant en $2^{O(t(n))}$.

Preuve Soit N une machine de Turing non-déterministe s'exécutant en un temps $t(n)$. Nous construisons une machine de Turing déterministe D qui simule N comme montré dans un théorème plus haut en cherchant dans l'arbre d'exécution non-déterministe de N . Maintenant, analysons cette simulation. Sur une entrée de taille n , chaque branche de l'exécution non-déterministe de N possède une longueur d'au plus $t(n)$. Chaque noeud dans l'arbre peut avoir b enfants, où b est le nombre maximum de choix possibles en fonction de la fonction de transition de N .

Du coup, le nombre total de feuilles dans l'arbre est d'au plus $b^{t(n)}$. La simulation procède en explorant l'arbre par niveau de profondeur. Donc, elle visite tous les noeuds d'une profondeur d avant de passer à la profondeur $d + 1$.

Le nombre total de noeuds dans l'arbre est inférieur au double du nombre de feuilles, donc nous pouvons borner cette valeur par $O(b^{t(n)})$. Le temps pris pour aller de la racine vers un noeud est de $O(t(n))$. Donc, le temps d'exécution de D est $O(t(n) * b^{t(n)}) = 2^{O(t(n))}$.

Aussi, comme décrit dans le théorème plus haut, la machine de Turing D possède trois rubans. Convertir ça en un mono-ruban prend un temps quadratique par rapport au temps d'exécution normal (par le théorème 7.8 du livre de référence). Ainsi, le temps d'exécution mono-ruban est $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$. CQFD.

3.2 (1.3.2) Give two definitions of the complexity class NP and prove that they are equivalent.

NP is the class of languages that have polynomial time verifiers.

NP is the class of languages that have nondeterministic polynomial time deciders.

Proof idea. We show how to convert a polynomial time verifier to an equivalent polynomial time Nondeterministic Turing machine (NTM) and vice versa. The NTM simulate the verifier by guessing the certificate. The verifier simulates the NTM by using the accepting branch as the certificate.

Proof.

Let A be a language and V a verifier for it. Assume that V is a TM that runs in time n^k and construct the NTM N to be decider for A as follows:

$N =$ "On input w of length n :

1. Nondeterministically select the string c of length at most n^k .

2. Run V on input $\langle w, c \rangle$.
3. if V accepts, *accepts*; otherwise, *reject*."

Let A be a language and N a polynomial time NTM that decides A . Construct a polynomial time deterministic verifier as follows:

$V =$ "On input $\langle w, c \rangle$, where c and w are strings,

1. Simulate N on input w , treating each symbol on c as a description of the nondeterministic choice to make at each step (as when using a 3-tapes deterministic TM to simulate a NTM).
2. If this branch of N 's computation accepts, *accepts*; otherwise, *reject*."

3.3 (1.3.3) Explain why all problems in NP can be decided in time $2^{O(n^k)}$ for some k .

Recall that a nondeterministic decider have all its computation branches that halt on all inputs.

A problem in **NP** can be described as a language L that have a nondeterministic decider N running in time $O(n^k)$ for some k . Assume that this decider is a NTM (that is polynomially equivalent to all the reasonable nondeterministic computational models). We can simulate N on a deterministic TM M by exploring all the computation branches of N . Thus M decide L .

With b the maximum number of children of a single node in the computation tree of N , N has at most $b^{O(n^k)+1} - 1$ nodes in its computation tree. So M will have to simulate N for at most $b^{O(n^k)+1} - 1 = 2^{O(n^k)}$ steps.

(Maybe add subtlety if M decides in time $2^{O(n^k)}$ but have longer branch for some input (but guesses to choose another branch that give the good answer in the desired time) ?)

3.4 (1.3.4) Is it the case that $P \subseteq NP$? Is it the case that $NP \subseteq P$? Explain.

Let's recall the definition of P and NP problems:

- P : Problem that can be solved and verified in $O(n^k)$ (polynomial) time.
- NP : Problem that can be verified in $O(n^k)$ (polynomial) time.

In the case that $P \neq NP$, $P \subseteq NP$ is true because every instance of a P problem can be verified in polynomial time. In the other hand $NP \subseteq P$ is false, because there exist problems in NP non solvable in polynomial time (like SAT). In the case that $P = NP$, $NP \subseteq P$ would be true (because it'd be the same set of problems).

3.5 (1.3.5) Define the notion of *polynomial time (mapping) reduction* and show that if A is polynomial time reducible to B and $B \in P$, then $A \in P$

Definition: a language A is **poly-time reducible** to a language B, written $A \leq_p B$, if \exists a poly-time computable function f such that:

$$\forall w, w \in A \iff f(w) \in B$$

If $B \in P$, and $A \leq_p B$, then $A \in P$.

Indeed, if we have a poly-time algorithm to decide B, we can do the following for a given $a \in A$:

1. $f(a) = b$
2. use the algorithm for B on $f(a)$

With this construction, we can decide A in poly-time, and we can conclude that $A \in P$.

3.6 (1.3.6) Define the notion of NP -completeness.

Definition: An NP-complete problem A is a problem such that:

- $A \in NP$
- Any problem in NP reduces in polynomial time to A.

Consequently, showing that one NP-complete problem is decidable in polynomial time would prove that $P=NP$. Similarly, proving that \nexists a poly-time algorithm for an NP-complete problem would prove that $P \neq NP$.

3.7 (1.3.7) Prove the following two statements: 1) If B is NP -complete and $B \in P$ then $P = NP$. 2) If B is NP-complete and $B \leq_p C$ and $C \in NP$ then C is NP-complete.

1. If B is NP-Complete and $B \in P$, then $P = NP$:
Using the definition of *poly-time reducibility*(see proof above) : yes. Let us set $A \in NP$. Because B is NP-Complete, A is reducible in B in poly-time ($A \leq_p B$). Hence, $A \in P$ as $B \in P$.
2. If B is NP-Complete and $B \leq_p C$ where C is NP, then C is NP-Complete :
Because B is NP-Complete, every A in NP is reducible in poly-time to B. In addition, B is poly-time reducible to C. Thus, every A in NP is poly-time reducible to C (This matches the definition of NP-Completeness).

3.8 (1.3.8) State the Cook-Levin Theorem and give a detailed outline of the proof

(Answer from Sipser textbook)

The Cook-Levin Theorem: **SAT is NP-complete**

To prove that we have to:

1. $\text{SAT} \in \text{NP}$
2. any language in NP is \leq_p to SAT

(1) A nondeterministic polynomial time machine can guess an assignment to a given formula ϕ and accept if the assignment satisfies ϕ .

(2) Let A a language in NP and N be the NTM (Nondeterministic Turing Machine) that decides A . The machine N decides A in n^k time for some constant k .

We will construct a tableau for N on w of size $n^k \times n^k$. The tableau is constructed following these properties:

- Each row represents the configurations of a branch of computation of N on w
- Each row starts and ends with the symbol $\#$
- The first row is the starting configuration and each row follows the previous according to N 's transition function
- A tableau is **accepting** if any row of the tableau is an accepting configuration

The reduction build tableau based on the computation tree, and every accepting tableau corresponds to an accepting computation branch of N on w .

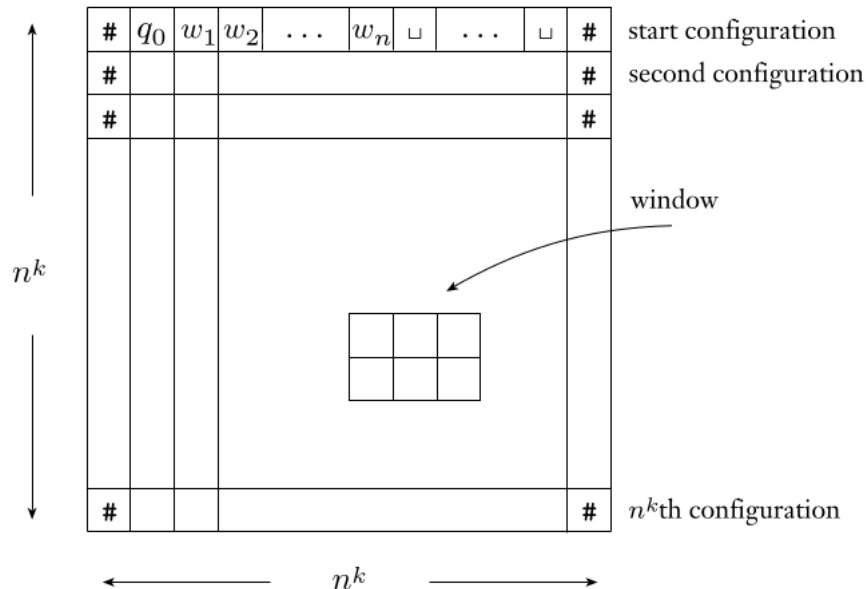


Figure 4: A tableau is an $n^k \times n^k$ table of configurations

Reduction

On input w the reduction produces a formula ϕ .

$$\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$$

This formula is composed of a set of literals $x_{i,j,s}$ where $1 \leq i, j \leq n^k$ and $s \in C$ with $C = Q \cup \Gamma \cup \{\#\}$ (i.e. a symbol that can be in the tableau).

ϕ_{cell} Ensure that each cell contains one and only one character:

$$\phi_{cell} = \bigwedge_{1 \leq i, j \leq n^k} \left[\bigvee_{s \in C} x_{i,j,s} \wedge \left(\bigwedge_{s, t \in C, s \neq t} \neg x_{i,j,s} \vee \neg x_{i,j,t} \right) \right]$$

The idea is the following: for each cell, at least one symbol must be true (first clause) and two symbols cannot be true together (second clause) i.e. one symbol can be true for a cell.

ϕ_{start} Ensure that the first row is the start configuration of N on w :

$$\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \dots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,_} \wedge \dots \wedge x_{1,n^k-1,_} \wedge x_{1,n^k,\#}$$

ϕ_{accept} Guarantees that an accepting configurations occurs in the tableau:

$$\bigvee_{1 \leq i, j \leq n^k} x_{i,j,q_{accept}}$$

ϕ_{move} Guarantees that each row of the tableau corresponds to a configuration that legally follows the configuration of the preceding row according to N 's rules. It does so by ensuring that each 2×3 window of the tableau is **legal**.

For example, a window is legal when it respects the following transitions: $\delta(q_1, a) = \{(q_1, b, R)\}$ and $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$. (Those transitions are example, and they are used in the figures to show legal/illegal windows). With a_1, \dots, a_6 a legal window, we have the following formula:

$$\phi_{move} = \bigwedge_{1 \leq i < n^k, 1 < j < n^k} (i, j)\text{-window is legal}$$

$$\bigvee_{a_1, \dots, a_6 \text{ is a legal window}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

Figure 5: Example of legal windows

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_2	a	a

(c)

b	q_1	b
q_2	b	q_2

Figure 6: Example f illegal windows

Next, we analyze the complexity of the reduction by looking at the size of ϕ . The tableau contains n^{2k} cells. Each cells has l variables associated with l being the size of C . Because l depends only on the TM N and not on the input size n , the total number of variables in $O(n^{2k})$. ϕ_{cell} , ϕ_{accept} and ϕ_{move} contains a fixed size formula for each cell *i.e.* their size is $O(n^{2k})$. ϕ_{start} has a fragment for each cell in the top row, so its size is $O(n^k)$. Total size of ϕ is $O(n^{2k})$. This is sufficient and shows that the size of ϕ is polynomial in n , thus the reduction can be done in polynomial time.

3.9 (1.3.9) Define the VERTEX-COVER problem and prove it is NP-complete.

VERTEX-COVER = { $\langle G, K \rangle$ | G est un graphe non dirigé, et peut être couvert par k -sommets }

1) Vertex cover est dans NP ?

Nous pouvons donner comme certificat l'ensemble des k -sommets et vérifier en temps polynomial si cette ensemble couvre bien le graphe.

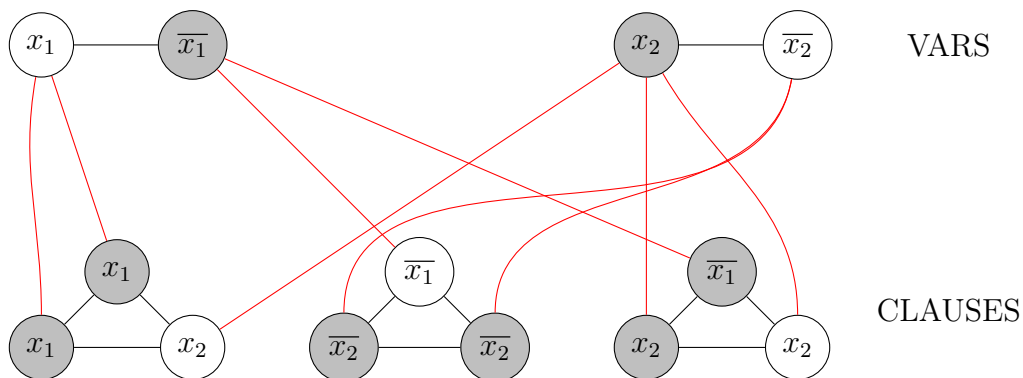
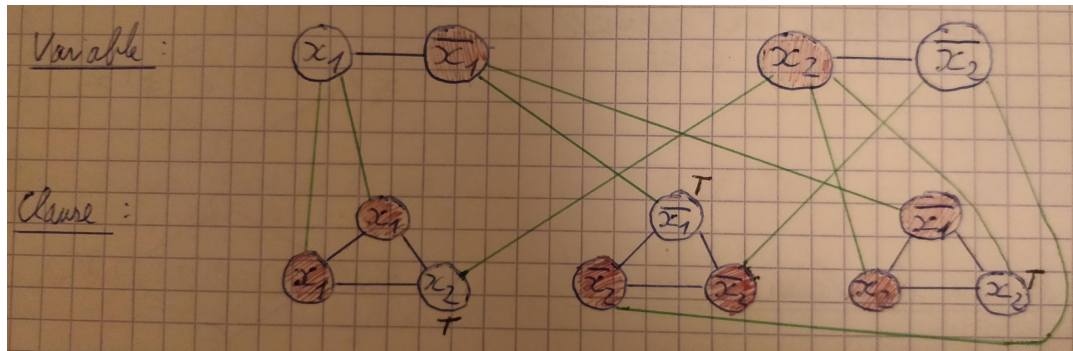
2) $3SAT \leq_p VERTEX-COVER$?

Nous allons construire un graphe à partir de la formule 3SAT.

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2).$$

Pour la construction nous utiliserons des gadgets :

- Variable : les paires de littéraux sous forme de noeud avec une arête.
 - Clause : Trois noeuds par clause qui correspondent aux littéraux dans la formule.
- Ajouter une arête entre les noeuds des variables et les clauses qui ont les mêmes littéraux.



Maintenant que la construction du graphe est fini nous pouvons nous occuper de la preuve dans les deux sens :

ϕ est satisfait SSI le graphe G a un vertex-cover avec k-sommets

Les assignations $x_1 = \text{false}$ et $x_2 = \text{true}$, satisfait la formule.

Avec ces assignements, dans la partie variable du graphe, je vais mettre dans mon ensemble des vertex-cover les noeuds x_2 et $\overline{x_1}$.

Dans la partie clause du graphe, sélectionner un noeud qui correspond à un littéral qui satisfait (donc x_2 ou $\overline{x_1}$) et ajouter les DEUX AUTRES, dans l'ensemble des vertex-cover.

→ En faisant ainsi, nous avons un ensemble de k-sommets qui couvre toutes les arêtes du graphe. Les arêtes des variables, les arêtes des clauses et les arêtes entre les clauses et variables sont couverts.

Si le graphe G a un vertex-cover avec k-sommets alors ϕ est satisfait

Il suffit de prendre les noeuds faisant partie de l'ensemble des vertex-cover dans la partie variable du graphe, et les assigner à true pour satisfaire la formule.

Cette reduction est faite en temps polynomial

3.10 (1.3.10) Définir le HAMILTONIAN-PATH problem and prove it is NP-complete.

(J'essaie de résumer ça du mieux que je peux mais y a quand même 3+ pages dans le livre sur ça)

Un problème de chemin Hamiltonien est un problème dans lequel on aimerait savoir s'il existe un chemin dans le graphe G qui va de s à t en passant seulement une fois par tous les noeuds.

Idée de preuve: Comme on sait qu'un chemin Hamiltonien est dans NP, on peut montrer que chaque problème NP est réductible dans un temps polynomial vers un chemin Hamiltonien. On essaie de convertir une formule 3-CNF en graphes dans lesquels les chemins Hamiltoniens correspondent bien aux assignements de la formule. Les graphes contiennent des gadgets qui vont imiter les variables et les clauses. Le gadget de la variable est une structure en diamant qui peut être traversée par deux chemins, ce qui correspond bien aux deux valeurs de vérité (vrai/faux). Le gadget de la clause est un simple noeud, qui assure qu'on va bien dans chaque gadget de clause (= on simule bien que chaque clause est satisfaite de la bonne manière)

Preuve: On doit montrer que $3SAT \leq_p HAMPATH$. Pour chaque formule 3-CNF ϕ , on va montrer comment construire un graphe dirigé G avec 2 noeuds, s et t , dans lequel il existe un chemin Hamiltonien entre s et t ssi ϕ est satisfaisable.

On commence la construction avec une formule 3CNF ϕ contenant k clauses:

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \dots \wedge (a_k \vee b_k \vee c_k)$$

où chaque a, b, c sont des littéraux x_i ou $\overline{x_i}$. Maintenant on va montrer comment convertir ϕ en un graphe G . On représente chaque variable x_i avec une structure en diamant qui contient une ligne de noeuds comme ceci:

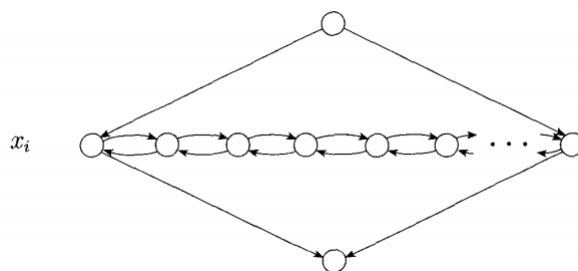


Figure 7: Représentation de x_i en structure en diamant

Chaque clause de ϕ est un noeud représenté par c_j . La figure suivante montre la structure global de G . Cela montre tous les éléments de G et leurs relations à l'exception des arêtes qui représentent la relation entre les variables et les clauses qui les contiennent.

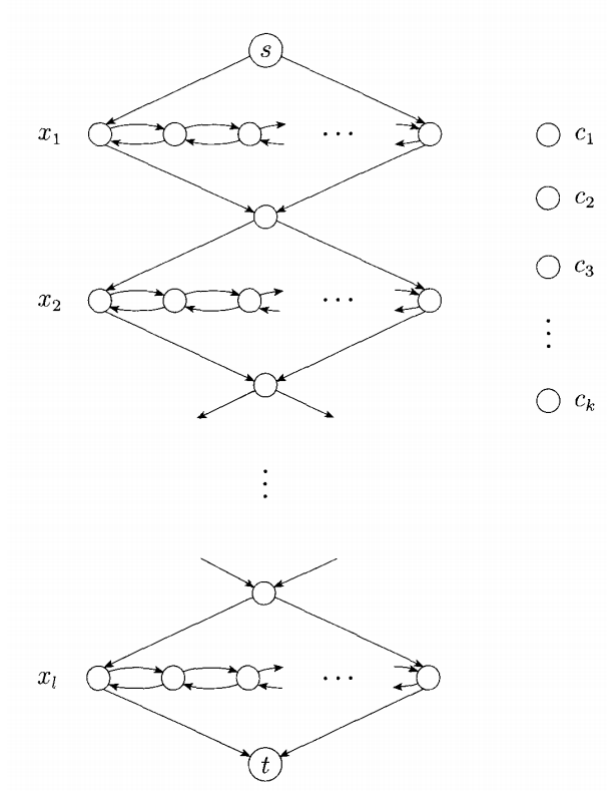


Figure 8: Structure de G

Ensuite nous montrons comment connecter les diamants représentant les variables aux noeuds représentant les clauses. Chaque diamant contient une ligne de noeuds connectés par des arêtes dans les deux directions. La ligne horizontale contient $3k + 1$ noeuds en plus des deux noeuds apparaissant à la fin qui appartiennent au diamant. Ces noeuds sont en fait groupés en paires adjacentes (une paire par clause) avec un noeud séparateur entre lesdites paires.

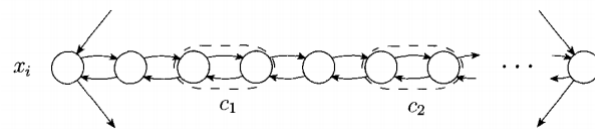


Figure 9: Ligne de noeuds dans un diamant

Si la variable x_i apparaît dans la clause c_j , on ajoute deux arêtes à partir de la j ème paire dans le i ème diamant vers le j ème noeud de la clause.

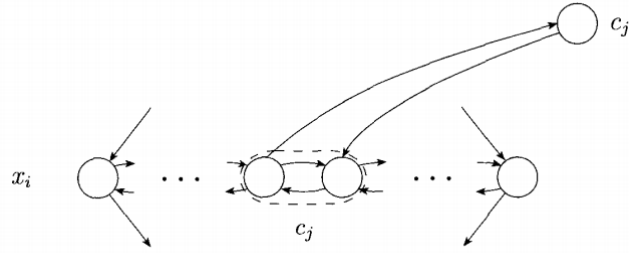


Figure 10: Arêtes quand la clause c_j contient x_i

Si $\overline{x_i}$ apparait dans la clause c_j , on fait pareil mais attention au sens des arêtes

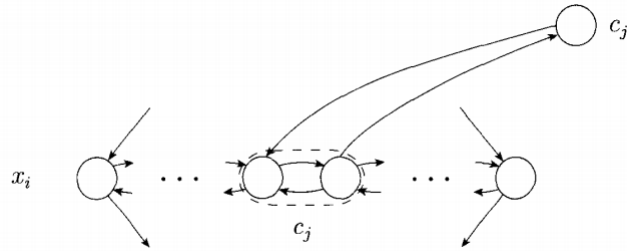


Figure 11: Arêtes quand la clause c_j contient $\overline{x_i}$

La construction du graph G est terminée. Il faut maintenant montrer que cette construction fonctionne, donc que si ϕ est satisfaisable, un chemin Hamiltonien existe bien entre s et t et inversement, si un tel chemin existe alors ϕ est satisfaisable.

On suppose que ϕ est satisfaisable et pour démontrer qu'un chemin Hamiltonien existe entre s et t , on ignore les noeuds des clauses. On traverse chaque diamant et si x_i est vrai, le chemin "zig-zag" à travers le diamant, dans le cas contraire, on "zag-zig" comme démontrer ci-dessous:

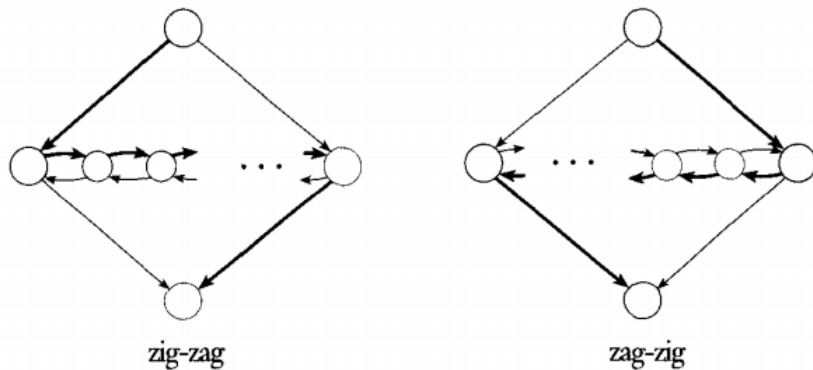


Figure 12: Zigzag zagzig

Comme on peut le voir, on arrive à couvrir tous les noeuds de G à l'exception des noeuds des clauses. Ce qui n'est pas un problème car on peut facilement ajouter un détour

à l'intérieur de la ligne de noeuds comme démontrer plus haut. Si on choisit x_i dans la clause c_j , on crée un détour dans la j ème pair provenant du i ème diamant. Cela est possible parce que x_i doit être vrai et donc le chemin "zig-zag" de gauche à droite à travers le diamant et donc les arêtes vers c_j sont dans le bon ordre et cela permet de faire un détour et de revenir. Même chose si on avait choisit $\overline{x_i}$, on aurait simplement fait de droite à gauche.

3.11 (1.3.11) Define the CLIQUE problem and prove it is NP-complete.

A clique in an undirected graph is a subset of vertices such that each vertex of the subset is adjacent to all the other vertices of the subset. Thus, this subset of vertices with their edges represents a complete subgraph. The clique problem is the problem of determine whether a graph contains a clique of size at least k .

Proof :

1. Showing CLIQUE is in NP:

The verifier takes a graph $G(V,E)$, k and a subset $S \subseteq V$. It checks if $|S| \geq k$ and then checks whether $(u,v) \in E$ for every $u, v \in S$. The verification is done in $O(n^2)$ time.

2. Showing CLIQUE is NP-HARD by doing a reduction from 3-SAT (known NP-complete) to CLIQUE:

Given an instance ϕ of 3-SAT, we will produce a graph $G(V, E)$ and an integer k such that G has a clique of size at least k if and only if ϕ is satisfiable.

We have $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_k$ k clauses, with C_i having 3 literals ($l_{1,i}, l_{2,i}, l_{3,i}$). One instance could be $\phi = C_1 \wedge C_2 \wedge C_3 = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$. To transform it into a k -clique: For each clause $C_i = (l_{1,i}, l_{2,i}, l_{3,i})$, we place the triple $l_{1,i}, l_{2,i}, l_{3,i}$ in the set V . Then, we put the edges in G between every literal that respects the conditions :

1. The two considered literals are not in the same clause.
2. The two considered literals are consistent \rightarrow not linking x and \overline{x} .

The reduction is in polynomial time.

NB: A graphical illustration can be found at http://www.ida.liu.se/opensa/OpenDSA/Books/Everything/html/threeSAT_to_clique.html

Now we have to show that

(1) ϕ is satisfiable $\Leftrightarrow G$ has a k -clique (2)

(1) If ϕ is satisfiable, then by the construction of the graph, there is a k -clique in it.

(2) If G has a k -clique. Let's consider the k vertices forming this k -clique. We know that every vertex is not connected to vertices in the same clause, neither to literals that are the negation of them. If we put a Boolean variable 1 in these vertices, we can conclude that there is at least one variable in each clause that is set to 1, and then ϕ is satisfied.

3.12 (1.3.12) Define the INDEPENDENT-SET problem and prove it is NP-complete. (Exercise)

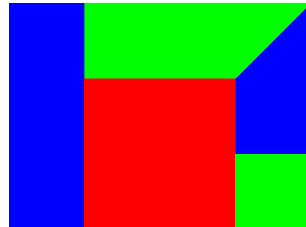
Independent Set Given a graph G , what is the biggest set of vertices which have no common edge?

1. $IS \in NP$? Obviously yes, given a subset of vertices $S \subseteq V$ of size k , one can check in poly-time whether there exists no edge between any of those vertices. In that case, the set is independent.
2. Then we prove that Clique reduces to I.S (and clique is NP-C)

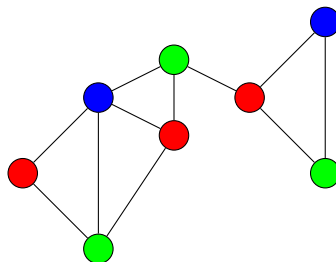
An I.S. of G can be defined as a clique of \overline{G} . Consequently, one can easily transform G to \overline{G} on polynomial time. At that point, we have reduced our I.S. problem to a maximal clique problem.

3.13 (1.3.13) Define the 3-COLORING problem and prove it is NP-complete. (Exercise 7.27)

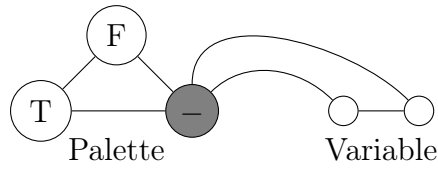
Définition Le problème des 3 couleurs consiste à colorier des formes géométriques juxtaposées de telle manière que deux formes côte à côte n'aient jamais la même couleur.



Modélisation Nous allons modéliser ce problème à l'aide d'un graph non dirigé où deux noeuds adjacents (deux noeuds sont adjacents lorsqu'ils sont reliés par une arête) ne peuvent pas avoir la même couleur.



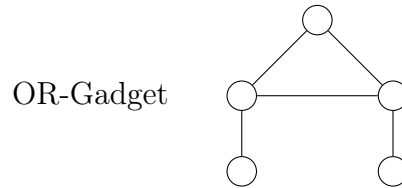
Démonstration Pour prouver que 3-COLORING est NP-Complet on va faire une réduction depuis 3-SAT, c-à-d 3 formules en forme normale conjonctives.



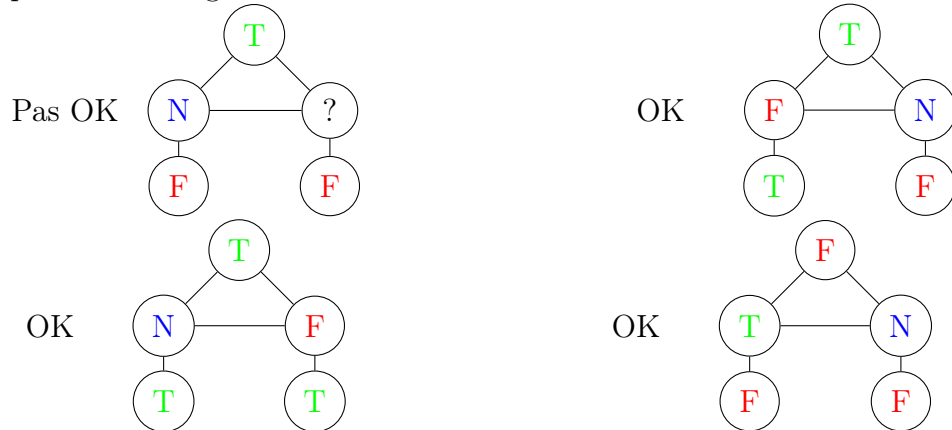
Mais avant cela, il nous faut définir plusieurs patternes:

Où T représente True, F représente False et $-$ représente le Neutre.

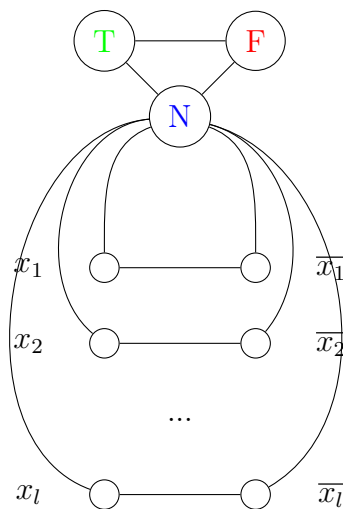
Définition également le "OR-GADGET":



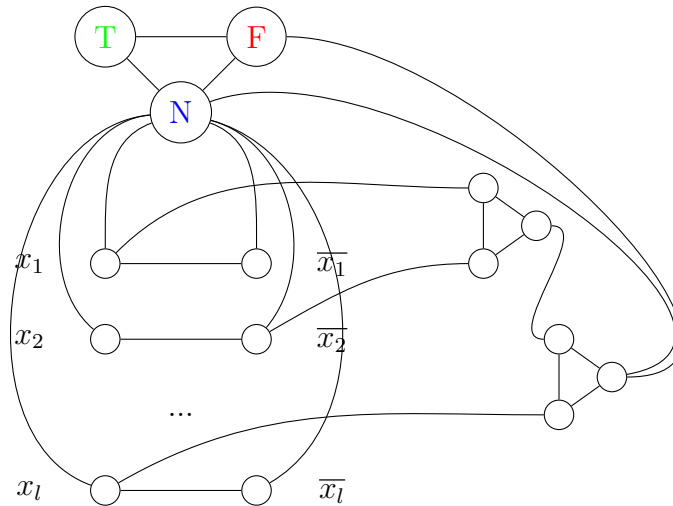
Il y a plusieurs configuration au "OR-GADGET":



Nous pouvons maintenant commencer la réduction. Pour ce faire, considérons 3 FNC ϕ , contenant les variables: x_1, \dots, x_l :

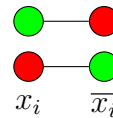


Par exemple pour la formule: $(x_1 \vee \overline{x_2} \vee x_l)$





En construisant nos formules en forme normal conjonctive de la sorte, on se retrouver avec un problème 3-SAT qui est satisfaisable si il s'agit également d'une solution au problème de 3-COLORING.

- Supposons que ϕ soit satisfaisable:
 \exists un SAT associé a $f : \{x_1, \dots, x_n\} \rightarrow \{T, F\}$. On peut donc colorer la palette en **rouge** **vert** **bleu** comme indiqué.



\forall variables x_i on colorie de manière suivante:

- Supposons qu'il existe un 3-COLORING colorer en **rouge** (F) **vert** (T) **bleu** (N).
 Pour le transformer en 3-SAT il suffit de mettre x_i à vrai si:  et à faux si 

3.14 (1.3.14) Define the SUBSET-SUM problem and prove it is NP-complete.

Réponse prélevée dans ce lien, contenant aussi des exemples illustrés pour chaque étape.

Définition Le problème SUBSET-SUM peut être défini ainsi : étant donnés un ensemble S de nombres entiers et un nombre entier t , existe-t-il un sous-ensemble $S' \subseteq S$ tel que la somme des nombres qu'il contient vaut t ? Plus formellement :

$$\text{SUBSET-SUM} = \left\{ \langle S, t \rangle \mid S = \{x_1, \dots, x_k\} \wedge \exists Y \subseteq \{1, \dots, k\}, \sum_{j \in Y} x_j = t \right\}$$

Théorème Le problème SUBSET-SUM est NP-complet.

Preuve Nous allons effectuer une réduction du problème 3-SAT en SUBSET-SUM. L'intuition de la réduction est que nous allons générer des nombres pour le problème SUBSET-SUM tels qu'une solution corresponde automatiquement à une assignation valable des variables dans 3-SAT.

Soient $x_1 \dots x_n$ les variables et $c_1 \dots c_m$ les clauses (conditions) d'une instance de 3-SAT. Pour chaque variable x_i , nous ajoutons à l'ensemble S les nombres t_i et f_i à $n + m$ digits, tels que :

- Le i -eme digit de t_i et de f_i vaut 1
- Pour $n + 1 \leq j \leq n + m$:
le j -eme digit de t_i vaut 1 si x_i est dans la clause c_{j-n} le j -eme digit de f_i vaut 1 si \bar{x}_i est dans la clause c_{j-n}
- Les autres digits de t_i et f_i valent 0

Ensuite, pour chaque clause c_j nous ajoutons à S les nombres x_j et y_j à $n + m$ digits, tels que :

- Leur $(n + j)$ -eme digit vaut 1
- Leurs autres digits valent 0

Finalement, nous construisons le total t à $n + m$ digits, tel que :

- Pour $1 \leq j \leq n$, le j -eme digit de t vaut 1
- pour $n + 1 \leq j \leq n + m$, le j -eme digit de s vaut 3

Montrons que si la formule 3-SAT a une assignation valable, alors l'instance de SUBSET-SUM correspondante a une solution :

- Si x_i est vrai, ajoutons t_i à Y ; sinon ajoutons f_i à Y
- Si c_j contient une ou deux variables vraies, ajoutons x_j à Y
- Si c_j contient une seule variable vraie, ajoutons y_j à Y

Il est facilement vérifiable que la somme des nombres de Y vaut t

- la somme de leurs i -eme digits vaut toujours 1 pour $i \leq n$, car une seule valeur f_i ou t_i a été choisie par variable (et ce sont les seuls nombres de S dont le i -eme digit vaut 1)
- la somme de leurs j -eme digits vaut toujours 3 pour $n < j \leq n + m$ car les valeurs x_j et y_j ont été choisies de manière à remplir cette condition.

Montrons que si l'instance de SUBSET-SUM a une solution, alors la formule 3-SAT correspondante a une assignation valable :

- Si $t_i \in Y$, assignons à x_i la valeur "vrai"; sinon assignons la valeur "faux"

De nouveau nous pouvons vérifier que l'assignation est valable :

- Chaque variable a exactement une assignation (autrement les premiers digits ne vaudraient pas 1)
- Chaque clause est satisfaite (autrement les derniers digits seraient inférieurs à 3)

3.15 (1.3.15) Give a dynamic programming algorithm for SUBSET-SUM and explain why it does not imply $P = NP$

Initialisation Supposons que $x[i]$ soient les nombres dans s , et t l'objectif de somme. La question est alors: est-il possible de créer un sous-ensemble avec les éléments de s telle que sa somme est t . Nous disposons d'un tableau T où $T[i][j] = True$ si et seulement s'il existe un sous-ensemble des i premiers nombres de s , tel que leur somme est j ($i = 1, \dots, |s|$; $j = 0, \dots, t$). La réponse au problème se trouve donc dans $T[|s|][t]$.

Algorithme Pour tout i allant de 1 à $|s|$, pour tout j allant de 0 à t , nous répétons l'opération :

$$[i][j] \leftarrow T[i-1][j] \vee T[i-1][j-x[i]]; \\ \text{return } T[|s|][t];$$

Concrètement, la somme des i premiers nombres de s donne j si la somme des $i-1$ premiers nombres de s donne également j ou $j-x[i]$ (le i^{me} nombre de s). Après avoir exécuté cette boucle pour tout i et tout j , la case du tableau contenant l'information qui nous intéresse est retournée.

Cela n'implique pas $P=NP$ Soit n la taille de l'input en nombre de bits. La complexité de l'algorithme de programmation dynamique décrit ci-dessus est : $|s| * t$. Il s'agit donc d'un algorithme à temps exponentiel.

exemple: Nous pouvons calculer n à l'aide de la formule : $n = \sum_{x_i \in s} \log_2 x_i$. Soient $s = \{5.000.000.000, 933.000.000, 333.212\}$ et $t = 5.933.333.212$. Nous savons alors que :

$$n \leq 4 * 10 * \log_2(10) \text{ et } |s| * t \approx 3 * 5 * 10^9$$

La complexité est donc nettement plus élevée que la taille de l'input. Bien que l'algorithme puisse vérifier une solution en temps polynomial (NP), rien ne garantit qu'il puisse trouver une solution en temps polynomial (P).

3.16 Show that if $P = NP$ you can determine a satisfying assignment to a 3-CNF formula in polynomial time, if one exists (Exercise 7.36).

If there exists a polynomial-time algorithm that solves an NP-complete problem, then 3SAT reduces in polynomial time to that problem. Consequently, 3SAT becomes decidable in polynomial time.

e.g: If SAT is decided in poly-time, one can translate 3SAT into SAT in $O(1)$, since 3SAT is a particular case of SAT, and 3SAT is decided in poly-time too.

3.17 Show that if $P = NP$ you can determine a maximum-size clique in a graph in polynomial time (Exercise 7.38).

Solution from the textbook. If you assume that $P = NP$, then $CLIQUE \in P$, and you can test whether G contains a clique of size k in polynomial time, for any value of k . By testing whether G contains a clique of each size, from 1 to the number of nodes in G , you can determine the size t of a maximum clique in G in polynomial time. Once you know t , you can find a clique with t nodes as follows. For each node x of G , remove x and calculate the resulting maximum clique size. If the resulting size decreases, replace x and continue with the next node. If the resulting size is still t , keep x permanently removed and continue with the next node. When you have considered all nodes in this way, the remaining nodes are a t -clique.

4 Space complexity

4.1 (1.4.1) Define the space complexity of deterministic and non-deterministic Turing machines, and give an example of a problem that can be solved in $O(t(n))$ space, but for which we do not know any $O(t(n))$ -time algorithm, for some function $t(n)$.

Definition 8.1 ²

Let M be a **deterministic** Turing machine that halts on all inputs. The space complexity of M is the function $t : N \rightarrow N$, where $t(n)$ is the maximum number of tape cells that M scans on any input of length n . If the space complexity of M is $t(n)$, we also say that M runs in space $t(n)$.

If M is a **nondeterministic** Turing machine wherein all branches halt on all inputs, we define its space complexity $t(n)$ to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n .

Example: SAT

We introduced the NP-complete problem SAT. Here, we show that SAT can be solved with a linear space algorithm. We believe that SAT cannot be solved with a polynomial time algorithm, much less with a linear time algorithm, because SAT is NP-complete. Space appears to be more powerful than time because **space can be reused**, whereas **time cannot**.

- $M =$ “On input $\langle \phi \rangle$, where ϕ is a Boolean formula:
1. For each truth assignment to the variables x_1, \dots, x_m of ϕ :
 2. Evaluate ϕ on that truth assignment.
 3. If ϕ ever evaluated to 1, accept ; if not, reject .”

Because SAT is NP-complete, we believe that it cannot be solved in polynomial-time. Machine M clearly runs in linear space because each iteration of the loop can reuse the same portion of the tape. The machine needs to store only the current truth assignment, and that can be done with $O(m)$ space. The number of variables m is at most n , the length of the input, so this machine runs in space $O(n)$.

4.2 (1.4.2) State and prove Savitch’s Theorem, and show it implies $PSPACE = NPSPACE$.

Recall that:

- A $f(n)$ space deterministic Turing machine is a TM that halts on all input and scans no more than $f(n)$ tape cells on any input of length n .
- A $f(n)$ space nondeterministic Turing machine is a TM wherein all computation branches halt on all input and that scans no more than $f(n)$ tape cells on any branch of its computation for any input of length n .

²Textbook 3rd edition

- Space complexity classes are defined as follows:

$\text{SPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space deterministic Turing machine}\}.$

$\text{NSPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space nondeterministic Turing machine}\}.$

Savitch's theorem For any function $f : \mathcal{N} \rightarrow \mathcal{R}^+$, where $f(n) \geq n$, $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$.

In other words, deterministic machines can simulate nondeterministic machines by using only the square of the space used by the nondeterministic machine.

One can prove that it holds whenever $f(n) \geq \log n$.

Recall that $\text{PSPACE} = \bigcup_k \text{SPACE}(n^k)$ and that NSPACE is the nondeterministic counterpart of PSPACE . Thus, since the square of a polynomial is still a polynomial, Savitch's theorem implies that $\text{NSPACE} = \text{PSPACE}$.

Proof idea. We need to simulate a $f(n)$ space NTM deterministically using at most $O(f^2(n))$ space.

Suppose we are given two configurations of the NTM, c_1 and c_2 , and a number t . The *yieldability problem* is to determine whether the NTM can get from c_1 to c_2 within n step. So by choosing c_1 the start configuration, c_2 the accept configuration and t the maximum number of steps that the NTM can use, we can determine whether the machine accepts the input.

Proof. Let N be a NTM deciding a language A in space $f(n)$. We construct a deterministic TM M deciding A .

Let w be a string considered as input to N . For configurations c_1 and c_2 of N on w , and an integer t , $\text{CANYIELD}(c_1, c_2, t)$ determine whether N can go from c_1 to c_2 in t or fewer steps along some nondeterministic path.

$\text{CANYIELD} = \text{"On input } c_1, c_2, t:$

1. If $t = 1$, *accept* if $c_1 = c_2$ or if c_1 yields c_2 in one step; *reject* otherwise.
2. If $t > 1$, for each configuration c_m of N on w using space $f(n)$:
 - (a) Run $\text{CANYIELD}(c_1, c_m, \frac{t}{2})$.
 - (b) Run $\text{CANYIELD}(c_m, c_2, \frac{t}{2})$.
 - (c) If both accept, then *accept*.
3. *reject*.

Let us now modify N so that when it accepts, it clears its tape and move the head to the leftmost cell thereby entering a configuration called c_{accept} . We can describe M as follows:

$M =$ “On input w of length n :

1. For $i \in \{1, 2, \dots\}$:
 - (a) If $\text{CANYIELD}(c_{\text{start}}, c_{\text{accept}}, i)$ accepts, *accept*.
 - (b) For each configuration c with length $i + 1$, run $\text{CANYIELD}(c_{\text{start}}, c, i + 1)$ and *reject* if none accept.

Obviously CANYIELD solves the yieldability problem. To see that M simulates N , just notice that the step (b) ensure termination by testing whether N uses at least space $i + 1$.

We now show that CANYIELD and M run in $O(f^2(n))$ space. CANYIELD use space to store the recursion stack of height at most $\log t$ where $t = 2^{O(f(n))}$ is the maximum time that the NTM may use on any branch (log because each recursion divide t in half). A configuration require $O(f(n))$ space to be stored and each recursion level need to store c_1 , c_2 , t and the stage number. So each level of the recursion stack has size $O(f(n))$ and the full stack has size at most $\log t \cdot O(f(n))$.

Moreover, one can select a constant d such that N has no more than $2^{df(n)}$ configurations using the $f(n)$ tape cells. Thus $t \leq 2^{df(n)}$ and $\log t \cdot O(f(n)) = O(f^2(n))$.

4.3 (1.4.3) Define the notion of PSPACE-completeness

A language B is PSPACE-complete if and only if:

1. $B \in \text{PSPACE}$
2. $\forall A \in \text{PSPACE}, A \leq_p B$

If B only satisfied the second condition, then B is PSPACE-hard.

4.4 (1.4.4) Define the TQBF problem and prove it is in PSPACE

A quantified Boolean formula is simply a Boolean formula with quantifiers: (\forall for all), (\exists there exists), for example :

$$\phi = \forall x \exists y [(x \vee y) \wedge (\neg x \vee \neg y)]$$

here ϕ is True, but it would be false if the quantifiers $\forall x$ and $\exists y$ was reversed, that's to say that the order of the quantifiers is important. When each variable of a formula appears within the scope of some quantifier, the formula is said to be fully quantified. A fully quantified Boolean formula is sometimes called a sentence and is always either true or false. For example, the preceding formula (ϕ) is fully quantified. However, if the initial part, ($\forall x$), of ϕ were removed, the formula would no longer be fully quantified and would be neither true nor false.

The TQBF problem is to determine whether a fully quantified Boolean formula is true or false. we define the language :

$$TQBF = \{\langle \phi \rangle \mid \phi \text{ is a true fully quantified Boolean formula.}\}$$

It is not too difficult to see that $TQBF \in_{reject} PSPACE$, we give a polynomial space algorithm (deciding $TQBF$) that assigns values to the variables and recursively evaluates the truth of the formula for those values. From that information, the algorithm can determine the truth of the original quantified formula. Observe that the depth of the recursion is at most the number of variables. At each level we need only store the value of one variable, so the total space used is $O(m)$, where m is the number of variables that appear in ϕ . Therefore, T runs in linear space.

$T =$ "On input $\langle \phi \rangle$, a fully quantified formula:

1. if ϕ contains no quantifiers, then it is an expression with only constants, so evaluate ϕ and *accept* if it is true; otherwise, *reject*.
2. If ϕ equals $\exists x \psi$, recursively call T on ψ , first with 0 substituted for x and then with 1 substituted for x . If either result is *accept*, then *accept* ; otherwise, *reject*.
3. If ϕ equals $\forall x \psi$, recursively call T on ψ , first with 0 substituted for x and then with 1 substituted for x . If both results are *accept*, then *accept* ; otherwise, *reject*."

4.5 (1.4.5) Prove that $TQBF$ is $PSPACE$ -complete.

(only thing to know is the proof idea, anyway I'll put the full proof straight from the book for those who want)

TQBF is PSPACE-complete.

PROOF IDEA To show that *TQBF* is in PSPACE we give a straightforward algorithm that assigns values to the variables and recursively evaluates the truth of the formula for those values. From that information the algorithm can determine the truth of the original quantified formula.

To show that every language A in PSPACE reduces to *TQBF* in polynomial time, we begin with a polynomial space-bounded Turing machine for A . Then we give a polynomial time reduction that maps a string to a quantified Boolean formula ϕ that encodes a simulation of the machine on that input. The formula is true iff the machine accepts.

As a first attempt at this construction, let's try to imitate the proof of the Cook–Levin theorem, Theorem 7.37. We can construct a formula ϕ that simulates M on an input w by expressing the requirements for an accepting tableau. A tableau for M on w has width $O(n^k)$, the space used by M , but its height is exponential in n^k because M can run for exponential time. Thus, if we were to represent the tableau with a formula directly, we would end up with a formula of exponential size. However, a polynomial time reduction cannot produce an exponential-size result, so this attempt fails to show that $A \leq_P \textit{TQBF}$.

Instead, we use a technique related to the proof of Savitch's theorem to construct the formula. The formula divides the tableau into halves and employs the universal quantifier to represent each half with the same part of the formula. The result is a much shorter formula.

PROOF First, we give a polynomial space algorithm deciding $TQBF$.

$T =$ “On input $\langle \phi \rangle$, a fully quantified Boolean formula:

1. If ϕ contains no quantifiers, then it is an expression with only constants, so evaluate ϕ and *accept* if it is true; otherwise, *reject*.
2. If ϕ equals $\exists x \psi$, recursively call T on ψ , first with 0 substituted for x and then with 1 substituted for x . If either result is *accept*, then *accept*; otherwise, *reject*.
3. If ϕ equals $\forall x \psi$, recursively call T on ψ , first with 0 substituted for x and then with 1 substituted for x . If both results are *accept*, then *accept*; otherwise, *reject*.”

Algorithm T obviously decides $TQBF$. To analyze its space complexity we observe that the depth of the recursion is at most the number of variables. At each level we need only store the value of one variable, so the total space used is $O(m)$, where m is the number of variables that appear in ϕ . Therefore T runs in linear space.

Next, we show that $TQBF$ is PSPACE-hard. Let A be a language decided by a TM M in space n^k for some constant k . We give a polynomial time reduction from A to $TQBF$.

The reduction maps a string w to a quantified Boolean formula ϕ that is true iff M accepts w . To show how to construct ϕ we solve a more general problem. Using two collections of variables denoted c_1 and c_2 representing two configurations and a number $t > 0$, we construct a formula $\phi_{c_1, c_2, t}$. If we assign c_1 and c_2 to actual configurations, the formula is true iff M can go from c_1 to c_2 in at most t steps. Then we can let ϕ be the formula $\phi_{c_{\text{start}}, c_{\text{accept}}, h}$, where $h = 2^{df(n)}$ for a constant d , chosen so that M has no more than $2^{df(n)}$ possible configurations on an input of length n . Here, let $f(n) = n^k$. For convenience, we assume that t is a power of 2.

The formula encodes the contents of tape cells as in the proof of the Cook–Levin theorem. Each cell has several variables associated with it, one for each tape symbol and state, corresponding to the possible settings of that cell. Each configuration has n^k cells and so is encoded by $O(n^k)$ variables.

If $t = 1$, we can easily construct $\phi_{c_1, c_2, t}$. We design the formula to say that either c_1 equals c_2 , or c_2 follows from c_1 in a single step of M . We express the equality by writing a Boolean expression saying that each of the variables representing c_1 contains the same Boolean value as the corresponding variable representing c_2 . We express the second possibility by using the technique presented in the proof of the Cook–Levin theorem. That is, we can express that c_1 yields c_2 in a single step of M by writing Boolean expressions stating that the contents of each triple of c_1 ’s cells correctly yields the contents of the corresponding triple of c_2 ’s cells.

If $t > 1$, we construct $\phi_{c_1, c_2, t}$ recursively. As a warmup let’s try one idea that doesn’t quite work and then fix it. Let

$$\phi_{c_1, c_2, t} = \exists m_1 [\phi_{c_1, m_1, \frac{t}{2}} \wedge \phi_{m_1, c_2, \frac{t}{2}}].$$

The symbol m_1 represents a configuration of M . Writing $\exists m_1$ is shorthand for $\exists x_1, \dots, x_l$, where $l = O(n^k)$ and x_1, \dots, x_l are the variables that encode m_1 . So this construction of $\phi_{c_1, c_2, t}$ says that M can go from c_1 to c_2 in at most t steps if some intermediate configuration m_1 exists, whereby M can go from c_1 to m_1 in at most $\frac{t}{2}$ steps and then from m_1 to c_2 in at most $\frac{t}{2}$ steps. Then we construct the two formulas $\phi_{c_1, m_1, \frac{t}{2}}$ and $\phi_{m_1, c_2, \frac{t}{2}}$ recursively.

The formula $\phi_{c_1, c_2, t}$ has the correct value; that is, it is TRUE whenever M can go from c_1 to c_2 within t steps. However, it is too big. Every level of the recursion involved in the construction cuts t in half but roughly doubles the size of the formula. Hence we end up with a formula of size roughly t . Initially $t = 2^{df(n)}$, so this method gives an exponentially large formula.

To reduce the size of the formula we use the \forall quantifier in addition to the \exists quantifier. Let

$$\phi_{c_1, c_2, t} = \exists m_1 \forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\} [\phi_{c_3, c_4, \frac{t}{2}}].$$

The introduction of the new variables representing the configurations c_3 and c_4 allows us to “fold” the two recursive subformulas into a single subformula, while preserving the original meaning. By writing $\forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\}$, we indicate that the variables representing the configurations c_3 and c_4 may take the values of the variables of c_1 and m_1 or of m_1 and c_2 , respectively, and that the resulting formula $\phi_{c_3, c_4, \frac{t}{2}}$ is true in either case. We may replace the construct $\forall x \in \{y, z\} [\dots]$ by the equivalent construct $\forall x [(x=y \vee x=z) \rightarrow \dots]$ to obtain a syntactically correct quantified Boolean formula. Recall that in Section 0.2 we showed that Boolean implication (\rightarrow) and Boolean equality ($=$) can be expressed in terms of AND and NOT. Here, for clarity, we use the symbol $=$ for Boolean equality instead of the equivalent symbol \leftrightarrow used in Section 0.2.

To calculate the size of the formula $\phi_{c_{\text{start}}, c_{\text{accept}}, h}$, where $h = 2^{df(n)}$, we note that each level of the recursion adds a portion of the formula that is linear in the size of the configurations and is thus of size $O(f(n))$. The number of levels of the recursion is $\log(2^{df(n)})$, or $O(f(n))$. Hence the size of the resulting formula is $O(f^2(n))$.

4.6 (1.4.6) Explain the interpretation of the TQBF problem as a game between an existential and a universal player.

Let $\phi = \exists x_1 \forall x_2 \exists x_3 \cdots Qx_k [\psi]$ be a quantified Boolean formula in prenex normal form. Here Q represents either a \forall or an \exists quantifier. We associate a game with ϕ as follows. Two players, called Player A and Player E, take turns selecting the values of the variables x_1, \dots, x_k . Player A selects values for the variables that are bound to \forall quantifiers and player E selects values for the variables that are bound to \exists quantifiers. The order of play is the same as that of the quantifiers at the beginning of the formula. At the end of play we use the values that the players have selected for the variables and declare that Player E has won the game if ψ , the part of the formula with the quantifiers stripped off, is now TRUE. Player A has won if ψ is now FALSE.

EXAMPLE 8.10

Say that ϕ_1 is the formula

$$\exists x_1 \forall x_2 \exists x_3 [(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})].$$

In the formula game for ϕ_1 , Player E picks the value of x_1 , then Player A picks the value of x_2 , and finally Player E picks the value of x_3 .

To illustrate a sample play of this game, we begin by representing the Boolean value TRUE with 1 and FALSE with 0, as usual. Let's say that Player E picks $x_1 = 1$, then Player A picks $x_2 = 0$, and finally Player E picks $x_3 = 1$. With these values for x_1, x_2 , and x_3 , the subformula

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$$

is 1, so Player E has won the game. In fact, Player E may always win this game by selecting $x_1 = 1$ and then selecting x_3 to be the negation of whatever Player A selects for x_2 . We say that Player E has a *winning strategy* for this game. A player has a winning strategy for a game if that player wins when both sides play optimally.

Now let's change the formula slightly to get a game in which Player A has a winning strategy. Let ϕ_2 be the formula

$$\exists x_1 \forall x_2 \exists x_3 [(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3})].$$

Player A now has a winning strategy because, no matter what Player E selects for x_1 , Player A may select $x_2 = 0$, thereby falsifying the part of the formula appearing after the quantifiers, whatever Player E's last move may be.

We next consider the problem of determining which player has a winning strategy in the formula game associated with a particular formula. Let

$$\text{FORMULA-GAME} = \{(\phi) \mid \text{Player E has a winning strategy in the formula game associated with } \phi\}.$$

4.7 (1.4.7) What do we know about the relations between the complexity classes P, NP, PSPACE, and EXPTIME ? Explain.

Let's examine the relationship of *PSPACE* with *P* and *NP*. We observe that $P \subset PSPACE$ because a machine that runs quickly cannot use a great deal of space. More precisely, for $t(n) > n$, any machine that operates in time $t(n)$ can use at most $t(n)$ space because a machine can explore at most one new cell at each step of its computation. Similarly, $NP \subset NPSPACE$, and so $NP \subset PSPACE$.

Conversely, we can bound the time complexity of a Turing machine in terms of its space complexity. For $f(n) > n$, a TM that uses $f(n)$ space can have at most $f(n)$ different configurations. A TM computation that halts may not repeat a configuration. Therefore a TM that uses space $f(n)$ must run in time $f(n)2^{O(f(n))}$ so $PSPACE \subset EXPTIME = \bigcup_k TIME(2^{n^k})$.

We summarize our knowledge of the relationships among the complexity classes defined so far in the series of containments $P \subset NP \subset PSPACE = NPSPACE \subset EXPTIME$.