Logique propositionnelle, **P**, **NP**

Jean Goubault-Larrecq

20 décembre 2007

Résumé

Ceci est la version 4 du poly du cours de logique et calculabilité, partie 2/4, datant du 20 décembre 2007. La version 3 datait du 12 décembre 2007, la version 2 du 20 novembre 2007. La version 1, datant du 14 novembre 2007, a été distribuée après le premier cours sur le site Web de l'auteur le 20 novembre 2007, et ne contenait pas les sections 3 et 4. Merci à François-Régis Sinot pour ses multiples relectures attentives, ainsi qu'à Maximilien Colange pour avoir détecté une imprécision.

Le but de ce cours est double : parler de la théorie de la logique propositionnelle (sans doute la plus élémentaire de toutes les logiques), et parler des classes de complexité **P** (temps polynomial) et **NP** (temps polynomial non déterministe). La connexion entre les deux sera fournie par le théorème de Cook-Levin : le problème de la satisfiabilité en logique propositionnelle est **NP**-complet.

Table des matières

1 Logique propositionnelle classique

La logique propositionnelle classique est la plus simple des logiques. En partant d'un ensemble de *formules atomiques* A, B, C, ..., qui peuvent être soit vraies soit fausses (celles-ci seront aussi appelées indifféremment *atomes* ou *variables propositionnelles*), on construit les *formules* à l'aide des connecteurs logiques $\wedge$ (la *conjonction*, "et"), $\vee$ (la *disjonction*, "ou"), $\neg$ (la *négation*, "non"), $\Rightarrow$ (l'*implication*, "implique"). On assimilera $\top$ (le *vrai*) et $\bot$ (le *faux*) à des connecteurs logiques à zéro argument — ce ne seront pas des formules atomiques. La grammaire est donnée par :

F	::=	A	formule atomique, atome, variable propositionnelle
	|	$\top$	vrai
	|	$\bot$	faux
	|	$F \wedge F$	conjonction
	|	$F \vee F$	disjonction
	|	$\neg F$	négation
	|	$F \Rightarrow F$	implication

Toute formule F est un arbre fini, dont les feuilles sont étiquetées par les formules atomiques et les nœuds internes sont étiquetés par des connecteurs. On les notera sous forme textuelle, en utilisant des parenthèses pour dissiper les ambiguïtés. On considérera que $\neg$ lie plus fort que $\wedge$, qui lie plus fort que $\vee$, qui lie plus fort que $\Rightarrow$. Par exemple, la formule $\neg A \wedge B \Rightarrow \neg B \vee C$, autrement dit $((\neg A) \wedge B) \Rightarrow ((\neg B) \vee C)$, est l'arbre de la figure 1.

1.1 Sémantique

Notons *Atom* l'ensemble de toutes les formules atomiques. On peut alors définir la *sémantique* $\mathcal{C}[\![F]\!]\rho$ des formules par récurrence structurelle sur F, comme suit, où l'*environnement* ρ est une fonction (totale) de *Atom* vers $\{0, 1\}$ — 0 représentant le faux, 1 le

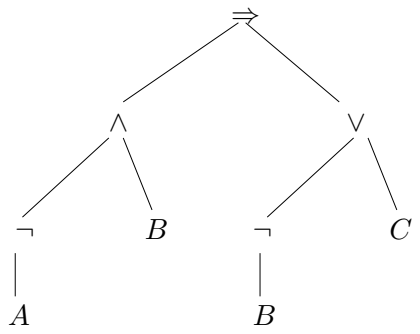


FIG. 1 – La formule $\neg A \wedge B \Rightarrow \neg B \vee C$, vue comme un arbre

vrai.

$$
\begin{aligned}
\mathcal{C}[\![A]\!]\rho &= \rho(A) \\
\mathcal{C}[\![\top]\!]\rho &= 1 \\
\mathcal{C}[\![\bot]\!]\rho &= 0 \\
\mathcal{C}[\![F_1 \wedge F_2]\!]\rho &= \begin{cases} 1 & \text{si } \mathcal{C}[\![F_1]\!]\rho = 1 \text{ et } \mathcal{C}[\![F_2]\!]\rho = 1 \\ 0 & \text{sinon} \end{cases} \\
\mathcal{C}[\![F_1 \vee F_2]\!]\rho &= \begin{cases} 1 & \text{si } \mathcal{C}[\![F_1]\!]\rho = 1 \text{ ou } \mathcal{C}[\![F_2]\!]\rho = 1 \\ 0 & \text{sinon} \end{cases} \\
\mathcal{C}[\![\neg F_1]\!]\rho &= \begin{cases} 1 & \text{si } \mathcal{C}[\![F_1]\!]\rho = 0 \\ 0 & \text{sinon} \end{cases} \\
\mathcal{C}[\![F_1 \Rightarrow F_2]\!]\rho &= \begin{cases} 1 & \text{si } \mathcal{C}[\![F_1]\!]\rho = 0 \text{ ou } \mathcal{C}[\![F_2]\!]\rho = 1 \\ 0 & \text{sinon} \end{cases}
\end{aligned}
$$

On note souvent aussi $\rho \models F$ ("F est vraie dans ρ", "ρ satisfait F") la relation $\mathcal{C}[\![F]\!]\rho = 1$. On remarquera que $\mathcal{C}[\![F_1 \Rightarrow F_2]\!]\rho = \mathcal{C}[\![\neg F_1 \vee F_2]\!]\rho$. En ce sens, l'implication $\Rightarrow$ peut être vue comme une abréviation d'une disjonction de deux formules, la première étant niée. De même, on peut voir $F_1 \vee F_2$ comme une abréviation de $\neg(\neg F_1 \wedge \neg F_2)$, ou bien $F_1 \wedge F_2$ comme une abréviation de $\neg(\neg F_1 \vee \neg F_2)$. Ces identités de sémantique ne seront plus valables en logique intuitionniste (section 5). On notera parfois $F_1 \Leftrightarrow F_2$ la formule $(F_1 \Rightarrow F_2) \wedge (F_2 \Rightarrow F_1)$ (*équivalence logique*).

Un des problèmes qui nous intéressera est celui, que nous appellerons FORM-SAT, de la satisfiabilité :
ENTRÉE : une formule propositionnelle F ;
QUESTION : F est-elle satisfiable, c'est-à-dire existe-t-il un environnement ρ tel que $\rho \models F$?

Ce problème est décidable, car on peut se restreindre à énumérer les variables qui apparaissent dans F uniquement, lesquelles sont en nombre fini. (Alors que *Atom*, qui est non spécifié, peut être infini.) Définissons cette notion de variables apparaissant dans F, ce sont les variables *libres* dans F :

Définition 1.1 (Libre) *L'ensemble* $\mathrm{FV}(F)$ *des* variables libres *dans* F *est défini par récurrence structurelle sur* F *par :*

$$\mathrm{FV}(A) = \{A\} \qquad \mathrm{FV}(\top) = \mathrm{FV}(\bot) = \emptyset \qquad \mathrm{FV}(\neg F) = \mathrm{FV}(F)$$
$$\mathrm{FV}(F_1 \wedge F_2) = \mathrm{FV}(F_1 \vee F_2) = \mathrm{FV}(F_1 \Rightarrow F_2) = \mathrm{FV}(F_1) \cup \mathrm{FV}(F_2)$$

Une récurrence structurelle immédiate sur F nous permet d'établir :

Lemme 1.2 *La valeur de vérité d'une formule* F *ne dépend que de ses variables libres : pour tous environnements* ρ *et* ρ' *qui coïncident sur* $\mathrm{FV}(F)$, $\mathcal{C}[\![F]\!]\,\rho = \mathcal{C}[\![F]\!]\,\rho'$.

Par abus de langage, on dira donc que $\varrho \models F$, où ϱ est une fonction *partielle* de *Atom* vers $\{0,1\}$, de domaine contenant $\mathrm{FV}(F)$, si et seulement si $\rho \models F$, où ρ est n'importe quelle extension de ϱ à tout *Atom*.

1.2 Compacité

Un théorème remarquable, et pas tout à fait trivial, est celui de compacité de la logique propositionnelle (voir ci-dessous). Pour le démontrer, nous aurons besoin du lemme de König, un grand classique. Un *arbre* est un ensemble d'objets, appels *nœuds*, muni d'une relation binaire $\rightarrow$, dite de *succession immédiate*, et d'un nœud r appelé *racine*, vérifiant :

- r n'est le successeur immédiat d'aucun nœud ;
- tout nœud n autre que r a un unique prédécesseur immédiat, c'est-à-dire qu'il existe un unique nœud m tel que $m \rightarrow n$;
- tout nœud n est successeur de r, c'est-à-dire $r \rightarrow^* n$, où $\rightarrow^*$ désigne la clôture réflexive transitive de $\rightarrow$.

Un arbre est *à branchement fini* si tous les nœuds n'ont qu'un nombre fini de successeurs immédiats. Il est *fini* si et seulement si l'ensemble de ses nœuds est fini. Une *branche* est une suite finie ou infinie de nœuds n_i, avec $n_0 = r$, et $n_i \rightarrow n_{i+1}$ pour tout i.

Lemme 1.3 (König) *Tout arbre à branchement fini et dont toutes les branches sont finies, est fini.*

Démonstration. Supposons que T, qui est à branchement fini, soit infini. On définit une branche infinie dans T, ce qui mènera à la conclusion. Pour ceci, on construit une suite infinie de nœuds n_i par récurrence sur i, tel que le sous-arbre de T de racine n_i (l'ensemble des nœuds n tels que $n_i \rightarrow^* n$) soit infini. Lorsque $i = 0$, on prend $n_0 = r$. Dans le cas de récurrence, on suppose que le sous-arbre de T de racine n_i est infini. Considérons les successeurs immédiats de n_i, disons $n_{i1}, \ldots, n_{ik}$: si tous étaient racines de sous-arbres finis, disons de cardinaux $p_1, \ldots, p_k$, alors il en serait de même de n_i (avec un cardinal d'au plus $p_1 + \ldots + p_k + 1$), contradiction. Donc l'un d'entre eux est le n_{i+1} recherché. □

Disons qu'un ensemble de formules (possiblement infini) S est *satisfiable* si et seulement s'il existe un environnement ρ tel que $\rho \models F$ pour tout $F \in S$. Il est *insatisfiable* sinon. Autrement dit, on voit un ensemble S comme une conjonction, possiblement infinie.

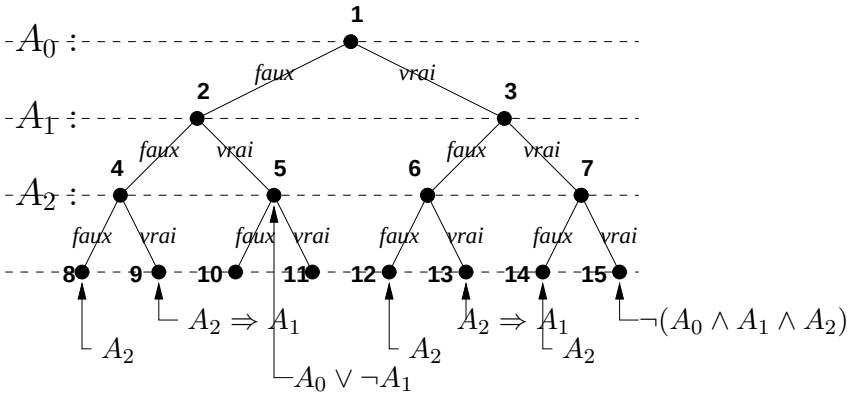


FIG. 2 – Un arbre sémantique

Théorème 1.4 (Compacité) *La logique propositionnelle classique est* compacte *: pour tout ensemble insatisfiable S de formules propositionnelles, il existe un sous-ensemble fini S_{fin} de S qui est déjà insatisfiable.*

Notons que si $S_{\text{fin}} \subseteq S$ est insatisfiable, alors S aussi. Ce théorème fournit une forme de réciproque.

Démonstration. Nous en donnons deux démonstrations. La première est élémentaire, mais ne fonctionne que lorsque *Atom* est dénombrable, ce qui sera le cas en général.

Première démonstration (cas dénombrable). Si *Atom* est dénombrable, on peut écrire $Atom = \{A_i \mid i \in \mathbb{N}\}$, avec les A_i deux à deux disjoints. Définissons un arbre T_0 dont les nœuds sont les environnements (partiels) ϱ dont le domaine est de la forme $\{A_0, A_1, \ldots, A_{i-1}\}$ pour un certain $i \in \mathbb{N}$. La racine est l'environnement vide, c'est-à-dire l'unique environnement (partiel) de domaine vide. La relation $\rightarrow$ est définie par : si ϱ est un environnement de domaine $\{A_0, A_1, \ldots, A_{i-1}\}$, alors ϱ a deux successeurs immédiats, de domaine $\{A_0, A_1, \ldots, A_{i-1}, A_i\}$, coïncidant avec ϱ sur $\{A_0, A_1, \ldots, A_{i-1}\}$, et donnant la valeur 0 ou 1 à A_i. Une façon moins formelle de le définir est de juste dessiner un arbre binaire infini : chaque fois que l'on descend d'un nœud étiqueté A_i, ceci revient à poser A_i faux si l'on descend à gauche, A_i vrai si l'on descend à droite. (Voir la figure 2, où l'on a d'une part représenté qu'une portion finie de cet arbre, et d'autre part numéroté les nœuds. La racine est le nœud 1.) T_0 est clairement à branchement fini. Chaque branche infinie décrit un unique environnement, qui est l'union des environnements partiels correspondant à chaque nœud. Réciproquement, tout environnement ρ définit une unique branche infinie, obtenue en descendant à gauche ou à droite selon les valeurs attribuées aux A_i, autrement dit dont le nœud numéro i est la restriction $\rho_{|\{A_0,A_1,\ldots,A_{i-1}\}}$ de ρ aux atomes $A_0, A_1, \ldots, A_{i-1}$. (C'est une construction importante, appelée *arbre sémantique*, ou *arbre de Herbrand*.)

Puisque S est insatisfiable, sur toute branche infinie, que nous identifions à l'environnement ρ associé, il existe une formule F_ρ de S telle que $\rho \not\models F_\rho$. Par exemple, dans la figure 2, avec S un ensemble contenant les formules A_2, $A_2 \Rightarrow A_1$, $A_0 \vee \neg A_1$, et $\neg(A_0 \wedge A_1 \wedge A_2)$,

en prenant pour ρ l'environnement qui à toute variable associe 0 (faux), on peut choisir $F_\rho = A_3$.

Mais il n'y a qu'un nombre fini d'atomes dans F_ρ, et l'on peut donc détecter que F_ρ est fausse dans ρ après avoir énuméré un nombre fini d'atomes de *Atom*. Formellement, il existe un indice i tel que $\mathrm{FV}(F_\rho) \subseteq \{A_0, A_1, \ldots, A_{i-1}\}$. Choisissons i minimal tel que $\rho_{|\{A_0,A_1,\ldots,A_{i-1}\}} \not\models F_\rho$: le nœud $\rho_{|\{A_0,A_1,\ldots,A_{i-1}\}}$ est appelé un *nœud d'échec pour la formule* F_ρ. Par exemple, le nœud 8 de la figure 2 est un nœud d'échec pour A_2. Appelons *nœud d'échec* (tout court) un nœud d'échec ϱ pour une formule, qui n'a aucun prédécesseur strict $\varrho' \subseteq \varrho$, $\varrho' \neq \varrho$, qui soit un nœud d'échec pour une autre formule. (Cette dernière notion n'est pas indispensable ici, en fait.) Par exemple, le nœud 10 de la figure 2 est un nœud d'échec pour A_2, mais comme 5 en est un prédécesseur strict de 10, et est un nœud d'échec pour $A_0 \vee \neg A_1$, seul 10 est un nœud d'échec (tout court) sur la branche $1, 2, 5, 10, \ldots$, et pas 10. Nous avons établi que toute branche infinie de T_0 contenait un (unique) nœud d'échec.

Considérons l'arbre T obtenu à partir de T_0 en tronquant la branche ρ juste après son nœud d'échec. Formellement, c'est la restriction de T_0 aux nœuds qui n'ont aucun prédécesseur strict qui soit un nœud d'échec. T est toujours à branchement fini, mais la construction ci-dessus montre que toutes les branches de T sont finies. Donc T est fini, par le lemme de König. En particulier, T n'a qu'un nombre fini de feuilles. Chacune de ces feuilles ϱ est un nœud d'échec pour une certaine formule $F_\varrho \in S$. L'ensemble $S_{\mathrm{fin}} = \{F_\varrho \mid \varrho \text{ nœud d'échec de } T_0\}$ est donc fini, inclus dans S, et insatisfiable par construction.

Seconde démonstration (cas général). La seconde démonstration se fonde sur des arguments topologiques, et justifie le nom du théorème. Cependant, elle est probablement moins accessible. Rappelons qu'une *topologie* sur un ensemble X est une collection de parties de X, appelées les *ouverts*, telle que toute union d'ouverts est ouverte, et toute intersection finie d'ouverts est ouverte. (Ceci inclut le vide et l'espace tout entier.) Un *fermé* est par définition le complémentaire d'un ouvert. Un *espace topologique* est un ensemble muni d'une topologie. Un *quasi-compact* de X est une partie K telle que, de tout recouvrement ouvert $(U_i)_{i\in I}$ de K (c'est-à-dire que les U_i sont tous ouverts, et que leur union contient K), on peut extraire un sous-recouvrement fini $(U_i)_{i\in I_{\mathrm{fin}}}$ (I_{fin} fini inclus dans I). Si $(X_j)_{j\in J}$ est une famille d'espaces topologiques, la *topologie produit* sur $\prod_{j\in J} X_j$ est la plus petite (pour l'inclusion des topologies) qui contienne tout produit $\prod_{j\in J} U_j$, avec U_j ouvert de X_j, et $U_j = X_j$ sauf pour un nombre fini d'indices $j \in J$. (C'est la plus petite topologie qui rende les projections $\pi_j : \prod_{j\in J} X_j \to X_j$ continues.) Le *théorème de Tychonoff* énonce que si tous les X_j sont quasi-compacts, alors leur produit $\prod_{j\in J} X_j$ aussi.

Considérons, pour chaque $A \in \mathit{Atom}$, l'espace $X_A = \{0, 1\}$ muni de la topologie discrète, c'est-à-dire celle qui contient toutes les parties de $\{0, 1\}$. X_A est quasi-compact, car ne contient qu'un nombre fini d'ouverts. L'espace $X = \prod_{A\in \mathit{Atom}} X_A$ est donc quasi-compact, par le théorème de Tychonoff. On remarque que c'est exactement l'espace de tous les environnements ρ. Notons aussi que, vu la définition de la topologie produit, il n'y a aucune raison que la topologie produit soit discrète. On démontre par récurrence structurelle sur la

formule F que $\{\rho \mid \mathcal{C}[\![F]\!]\rho = 0\}$ est à la fois ouvert et fermé dans X. Lorsque $F = A \in Atom$, c'est par définition de la topologie produit. Lorsque F est une conjonction ou $\top$, c'est parce que toute intersection finie d'ouverts est ouverte et toute union (finie) d'ouverts est ouverte. Lorsque F est une négation, c'est parce que le complémentaire d'un ouvert est fermé et le complémentaire d'un fermé est ouvert. Les autres cas s'en déduisent facilement. Pour chaque $F \in S$, posons $U_F = \{\rho \mid \mathcal{C}[\![F]\!]\rho = 0\}$. Le fait que S soit insatisfiable revient à dire que la famille $(U_F)_{F\in S}$ forme un recouvrement ouvert de X. Comme X est quasi-compact, on peut donc en extraire un sous-recouvrement fini $(U_F)_{F\in S_{\text{fin}}}$, ce qui revient à dire que S_{fin} est insatisfiable. □

Nous réutiliserons les techniques de la première démonstration plus loin. Il est donc nécessaire de bien étudier cet argument.

▷ **Exercice 1.1**

Soit S un ensemble de formules propositionnelles. Supposons que, pour chaque partie finie S' de S, on puisse trouver un environnement $\rho_{S'}$ qui satisfait S'. On ne supposera pas que les environnements $\rho_{S'}$ sont compatibles, autrement dit il n'y a aucune raison que $\rho_{S'}(A) = \rho_{S''}(A)$ ($A \in Atom$) pour deux parties finies S' et S'' distinctes. Montrer cependant qu'il existe un environnement ρ qui satisfait tout S.

1.3 Déduction naturelle, le système NK

La fonction sémantique $\mathcal{C}[\![\_]\!]$ définit ce que veut dire qu'une formule soit vraie, fausse, satisfiable, insatisfiable. À négation près, ceci définit aussi la notion de *validité* d'une formule : on dit que F est *valide*, ce que l'on note $\models F$, si et seulement si $\rho \models F$ pour tout environnement ρ. Il revient au même de dire que F est valide et que $\neg F$ est insatisfiable.

Plutôt que de rechercher si F est valide, en énumérant typiquement tous les environnements partiels de domaine $\mathrm{FV}(F)$, ce qui prend un temps exponentiel — c'est la méthode des *tables de vérité* —, on peut chercher une *démonstration* de F. Par exemple, on peut voir tout de suite qu'une formule de la forme $F \Rightarrow F$ est valide, même sans énumérer tous les environnements partiels sur $\mathrm{FV}(F)$. C'est le genre d'avantage que procurera un *système de déduction*. (On verra cependant à la section 4 que la question de la validité, ou de la prouvabilité d'une formule propositionnelle classique, est intrinsinquement difficile.)

Nous allons voir quelques systèmes de déduction, tous dûs à Gentzen dans les années 1930 (avec des variantes). Le premier est la *déduction naturelle*. Il définit des règles permettant de dériver des *jugements* de la forme $\Gamma \vdash F$, où F est une formule, et Γ un ensemble fini de formules. Intuitivement, un tel jugement est *vrai* si et seulement si l'implication $\bigwedge \Gamma \Rightarrow F$ est vraie, où $\bigwedge \Gamma$ est la conjonction de toutes les formules de Γ.

On peut se demander pourquoi recourir à une notion aussi compliquée que celle de jugement alors qu'on avait déjà la notion de formule. La raison première en est le traitement de l'implication : pour démontrer $\Gamma \vdash F_1 \Rightarrow F_2$, il suffira de démontrer $\Gamma, F_1 \vdash F_2$, où Γ, F_1 dénote l'union de Γ avec $\{F_1\}$. Ceci représente le mécanisme naturel de raisonnement consistant à démontrer $F_1 \Rightarrow F_2$ en posant F_1 comme hypothèse et en démontrant F_2 sous cette hypothèse. La partie à gauche du signe "thèse" $\vdash$ représente ainsi l'ensemble des hypothèses que l'on fait pour démontrer la formule de droite.

$$\frac{}{\Gamma, F \vdash F}\,(Ax) \qquad \frac{}{\Gamma \vdash \top}\,(\top I) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash G}\,(\bot E) \qquad \frac{\Gamma \vdash \neg\neg F}{\Gamma \vdash F}\,(\neg\neg E)$$

$$\frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \wedge F_2}\,(\wedge I) \qquad \frac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_1}\,(\wedge E_1) \qquad \frac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_2}\,(\wedge E_2)$$

$$\frac{\Gamma \vdash F_1}{\Gamma \vdash F_1 \vee F_2}\,(\vee I_1) \qquad \frac{\Gamma \vdash F_2}{\Gamma \vdash F_1 \vee F_2}\,(\vee I_2) \qquad \frac{\Gamma \vdash F_1 \vee F_2 \quad \Gamma, F_1 \vdash G \quad \Gamma, F_2 \vdash G}{\Gamma \vdash G}\,(\vee E)$$

$$\frac{\Gamma, F \vdash \bot}{\Gamma \vdash \neg F}\,(\neg I) \qquad \frac{\Gamma \vdash \neg F \quad \Gamma \vdash F}{\Gamma \vdash G}\,(\neg E)$$

$$\frac{\Gamma, F_1 \vdash F_2}{\Gamma \vdash F_1 \Rightarrow F_2}\,(\Rightarrow I) \qquad \frac{\Gamma \vdash F_1 \Rightarrow F_2 \quad \Gamma \vdash F_1}{\Gamma \vdash F_2}\,(\Rightarrow E)$$

FIG. 3 – Le système de déduction naturelle **NK**

Les règles du système **NK** de *déduction naturelle classique* sont présentées à la figure 3. À part l'*axiome* (Ax), elles sont structurées en règles d'*introduction* (colonne de gauche, le nom de la règle contenant un I) et d'*élimination* (colonne de droite, le nom de la règle contenant un E). Par exemple, la règle $(\Rightarrow I)$ permet de démontrer $\Gamma \vdash F_1 \Rightarrow F_2$, comme annoncé, à condition de démontrer d'abord la *prémisse* $\Gamma, F_1 \vdash F_2$.

Les règles d'introduction d'un connecteur, par exemple $\Rightarrow$, permettent de démontrer une formule qui, vue sous forme d'un arbre comme à la figure 1, voit sa racine étiquetée par le connecteur en question. (On dit que cet opérateur qui étiquette la racine de la formule est le *connecteur de tête* de la formule.) On note qu'il n'y a pas de règle d'introduction du faux $\bot$ — ce serait malsain. Il y a d'autre part deux règles d'introduction du "ou" $\vee$, selon que l'on a démontré le membre gauche ou le membre droit de la disjonction. (Il peut sembler surprenant que l'on n'autorise pas d'autre façon de démontrer une disjonction. Voir l'exercice 1.2, ou l'exercice 1.6 pour réaliser que ceci n'est pas une limitation.)

Les règles d'élimination d'un connecteur permettent de partir d'une démonstration d'une formule ayant un connecteur de tête, et d'en déduire une autre formule. Pour ceci, on a parfois besoin d'une *prémisse auxiliaire*. Par exemple, à partir d'une démonstration de la *prémisse principale* $\Gamma \vdash F_1 \Rightarrow F_2$, où $\Rightarrow$ est le connecteur de tête, on peut démontrer F_2 à condition de disposer d'une démonstration de la prémisse auxiliaire $\Gamma \vdash F_1$, par la règle $(\Rightarrow E)$. La règle d'élimination du "ou" est un peu particulière, et correspond à une forme de raisonnement connue sous le nom de *raisonnement par cas* (ignorons Γ pour les besoins de l'explication) : si on peut démontrer $F_1 \vee F_2$, et que l'on peut démontrer G que ce soit sous l'hypothèse F_1 ou sous l'hypothèse F_2, alors on peut en déduire G dans tous les cas. De même, la règle $(\bot E)$ peut être vue comme une analyse de cas avec zéro cas : si l'on

démontrer le faux, pas la peine d'examiner de cas, on peut directement démontrer G, où G est n'importe quelle formule. La règle d'élimination du "non", $(\neg E)$, peut s'expliquer de la même façon, sachant que $\neg F$ est équivalent à $F \Rightarrow \bot$ (utiliser $(\Rightarrow E)$ puis $(\bot E)$).

Finalement, on a une règle étrange d'*élimination de la double négation*, $\neg\neg E$, dont le style détone. Nous y reviendrons à la section 5.

Une *dérivation* (ici, en **NK**) est un arbre fini dont les nœuds sont étiquetés par des jugements, et dont la relation de successeur immédiat est défini par les règles, au sens où les successeurs immédiats d'un jugement $\mathcal{J}$ sont nécessairement exactement les prémisses d'une règle dont la conclusion (sous la barre) soit $\mathcal{J}$. On trace en général l'arbre avec sa racine en bas. Par exemple, voici une dérivation de $\neg(\neg F_1 \wedge \neg F_2) \vdash F_1 \vee F_2$ en **NK**, où Γ est l'ensemble $\neg(\neg F_1 \wedge \neg F_2), \neg(F_1 \vee F_2)$, et la démonstration omise à droite (les $\vdots$) est identique à la démonstration de $\Gamma \vdash \neg F_1$, à ceci près que l'on utilise $(\vee I_2)$ plutôt que $(\vee I_1)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \neg(\neg F_1 \wedge \neg F_2)}(Ax) \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, F_1 \vdash \neg(F_1 \vee F_2)}(Ax) \quad \dfrac{\dfrac{}{\Gamma, F_1 \vdash F_1}(Ax)}{\Gamma, F_1 \vdash F_1 \vee F_2}(\vee I_1)}{\Gamma, F_1 \vdash \bot}(\neg E)}{\Gamma \vdash \neg F_1}(\neg I) \qquad \vdots}{\Gamma \vdash \neg F_1 \wedge \neg F_2}(\wedge I)}{\underbrace{\neg(\neg F_1 \wedge \neg F_2), \neg(F_1 \vee F_2)}_{\Gamma} \vdash \bot}(\neg E)}{\neg(\neg F_1 \wedge \neg F_2) \vdash \neg\neg(F_1 \vee F_2)}(\neg I)}{\neg(\neg F_1 \wedge \neg F_2) \vdash F_1 \vee F_2}(\neg\neg E)$$

L'implication réciproque est donnée par la dérivation :

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash F_1 \vee F_2}(Ax) \qquad \dfrac{\dfrac{\dfrac{}{\Delta, F_1 \vdash \neg F_1 \wedge \neg F_2}(Ax)}{\Delta, F_1 \vdash \neg F_1}(\wedge E_1) \quad \dfrac{}{\Delta, F_1 \vdash F_1}(Ax)}{\Delta, F_1 \vdash \bot}(\neg E) \qquad \vdots}{\underbrace{F_1 \vee F_2, \neg F_1 \wedge \neg F_2}_{\Delta} \vdash \bot}(\vee E)}{F_1 \vee F_2 \vdash \neg(\neg F_1 \wedge \neg F_2)}(\neg I)$$

où de même la sous-dérivation manquante de $\Delta, F_2 \vdash \bot$ est similaire à celle de $\Delta, F_1 \vdash \bot$, en remplaçant $(\wedge E_1)$ par $(\wedge E_2)$.

▷ **Exercice 1.2**

Donner une dérivation du jugement $\vdash (F_1 \vee F_2) \Rightarrow (F_2 \vee F_1)$ en **NK**.

▷ **Exercice 1.3**
Montrer que la règle d'*affaiblissement* :

$$\frac{\Gamma \vdash F}{\Gamma, \Delta \vdash F}\,(Aff)$$

est *admissible* au sens où l'on peut transformer toute dérivation de $\Gamma \vdash F$ en une de $\Gamma, \Delta \vdash F$. (On note Γ, Δ l'union des deux ensembles de formules Γ et Δ.)

▷ **Exercice 1.4**
Montrer que la règle $(\bot E)$ est superflue dans **NK**, au sens où elle est déjà admissible dans **NK** privé de $(\bot E)$.

▷ **Exercice 1.5**
Sur le même principe que les démonstrations données en exemple plus haut, donner une dérivation en **NK** de $\neg(F_1 \vee F_2) \vdash \neg F_1 \wedge \neg F_2$. On fournira pour ceci deux dérivations, une de $\neg(F_1 \vee F_2) \vdash \neg F_1$, l'autre de $\neg(F_1 \vee F_2) \vdash \neg F_2$, et l'on utilisera $(\wedge I)$. On demande de plus à ne pas utiliser la règle $(\neg\neg E)$.

▷ **Exercice 1.6**
Déduire de l'exercice 1.5 une démonstration en **NK** (utilisant $(\neg\neg E)$ cette fois-ci) de la loi du *tiers exclu* : $\vdash F \vee \neg F$.

Disons que $\rho \models \Gamma \vdash F$ si et seulement si $\rho \models \bigwedge \Gamma \Rightarrow F$, autrement dit si et seulement si, dès que $\rho \models G$ pour toute formule G de Γ, alors $\rho \models F$. De façon équivalente, si $\rho \models F$ ou $\rho \not\models G$ pour au moins une formule $G \in \Gamma$. Le jugement est *valide*, en notation $\models \Gamma \vdash F$, si et seulement si $\rho \models \Gamma \vdash F$ pour tout environnement ρ.

Lemme 1.5 (Correction) *Le système* **NK** *est* correct *: tout jugement dérivable en* **NK** *est valide.*

Démonstration. On démontre par récurrence structurelle sur une dérivation π que sa conclusion est un jugement valide. Tous les cas sont faciles, et laissés au lecteur. □

Ce qui est surtout intéressant est le résultat réciproque :

Théorème 1.6 (Complétude) *Le système* **NK** *est* complet *: tout jugement valide est dérivable en* **NK***.*

Démonstration. On va démontrer le résultat pour des jugements de la forme spéciale $\Gamma \vdash \bot$. On en déduira le résultat général comme suit : si $\Gamma \vdash F$ est valide, alors $\Gamma, \neg F \vdash \bot$ aussi, donc ce dernier sera dérivable. On construira alors la dérivation :

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, \neg F \vdash \bot\end{array}}{\Gamma \vdash \neg\neg F}\,(\neg I)}{\Gamma \vdash F}\,(\neg\neg E)$$

Supposons donc $\Gamma \vdash \bot$ valide, c'est-à-dire Γ insatisfiable, et construisons une démonstration de $\Gamma \vdash \bot$ en **NK**. Numérotons $A_0, A_1, \ldots, A_{n-1}$ les variables libres dans Γ. La construction

de l'arbre fini T_0 de la première démonstration du théorème 1.4 (voir la figure 2) peut aider à visualiser ce que l'on fait.

Pour tout environnement partiel ϱ de domaine $\{A_0, A_1, \ldots, A_{i-1}\}$ (avec $0 \leq i \leq n$), notons Δ_ϱ l'ensemble de formules contenant A_j si $\varrho(A_j) = 1$, $\neg A_j$ si $\varrho(A_j) = 0$ $(0 \leq j \leq i)$ et aucune autre. Par exemple, si ϱ envoie A_0 et A_1 sur 0, A_2 sur 1, et A_3 sur 0, alors Δ_ϱ est $\neg A_0, \neg A_1, A_2, \neg A_3$. Ceci permet donc d'associe à tout nœud de T un ensemble fini de formules Δ_ϱ. Noter que dans notre cas, T est fini par construction.

On commence par démontrer que **NK** est capable d'évaluer correctement la valeur de vérité de toute formule F avec $\mathrm{FV}(F) \subseteq \{A_0, A_1, \ldots, A_{n-1}\}$, au sens où : $(*)$ pour tout environnement partiel ϱ de domaine $\{A_0, A_1, \ldots, A_{n-1}\}$, si $\varrho \models F$, alors $\Delta_\varrho \vdash F$ est dérivable, et si $\varrho \not\models F$ alors $\Delta_\varrho \vdash \neg F$ est dérivable. C'est par récurrence structurelle sur F. Si F est un atome A_j, alors de $\varrho \models F$ on déduit que $A_j \in \Delta_\varrho$, et l'on utilise (Ax) ; de $\varrho \not\models F$ on déduit que $\neg A_j \in \Delta_\varrho$, et l'on utilise (Ax) de nouveau. Si F est de la forme $F_1 \wedge F_2$, alors soit $\varrho \models F_1 \wedge F_2$, donc $\varrho \models F_1$ et $\varrho \models F_2$, et l'on peut produire la dérivation :

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ \Delta_\varrho \vdash F_1 & \Delta_\varrho \vdash F_2\end{array}}{\Delta_\varrho \vdash F_1 \wedge F_2}\,(\wedge I)$$

en utilisant l'hypothèse de récurrence pour trouver les dérivations manquantes ; soit $\varrho \not\models F_1 \wedge F_2$, et alors $\varrho \not\models F_1$ ou $\varrho \not\models F_2$. Traitons du premier cas, le second étant similaire. Par hypothèse de récurrence, on dispose d'une dérivation de $\Delta_\varrho \vdash \neg F_1$. En utilisant la règle d'affaiblissement (exercice 1.3) pour fabriquer la dérivation omise ($\vdots$) ci-dessous, on construit :

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \dfrac{}{\Delta_\varrho, F_1 \wedge F_2 \vdash F_1 \wedge F_2}\,(Ax) \\ \Delta_\varrho, F_1 \wedge F_2 \vdash \neg F_1 & \Delta_\varrho, F_1 \wedge F_2 \vdash F_1\end{array}}{\Delta_\varrho, F_1 \wedge F_2 \vdash \bot}\,(\neg E)}{\Delta_\varrho \vdash \neg(F_1 \wedge F_2)}\,(\neg I)$$

Si F est de la forme $\neg F_1$, soit $\varrho \models F$, donc $\varrho \not\models F_1$, d'où l'on déduit une dérivation de $\Delta_\varrho \vdash \neg F_1$ par hypothèse de récurrence ; soit $\varrho \not\models F$, donc $\varrho \models F_1$, alors l'hypothèse de récurrence nous fournit une dérivation de $\Delta_\varrho \vdash F_1$, donc une de $\Delta_\varrho, \neg F_1 \vdash F_1$ par affaiblissement (exercice 1.3), et ensuite :

$$\dfrac{\dfrac{\dfrac{}{\Delta_\varrho, \neg F_1 \vdash \neg F_1}\,(Ax) \qquad \begin{array}{c}\vdots \\ \Delta_\varrho, \neg F_1 \vdash F_1\end{array}}{\Delta_\varrho, \neg F_1 \vdash \bot}\,(\neg E)}{\Delta_\varrho \vdash \neg\neg F_1}\,(\neg I)$$

On procède de même pour les autres connecteurs (exercice).

L'affirmation $(*)$ permet de voir que pour tout nœud ϱ à la ligne du bas de l'arbre T (voir de nouveau la figure 2), si $\varrho \not\models F$ alors $\Delta_\rho \vdash \neg F$ est dérivable. C'est en particulier le cas pour toutes les formules F de Γ. Or, Γ étant insatisfiable, il existe une formule F de Γ telle que $\varrho \not\models F$. On en déduit une dérivation de $\Gamma, \Delta_\varrho \vdash \bot$ par :

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma, \Delta_\varrho \vdash \neg F\end{array} \quad \dfrac{}{\Gamma, \Delta_\varrho \vdash F}\,(Ax)}{\Gamma, \Delta_\varrho \vdash \bot}\,(\neg E)$$

où la dérivation manquante ($\vdots$) est obtenue à partir de celle de $\Delta_\varrho \vdash \neg F$ par affaiblissement (exercice 1.3). Ceci est vrai pour tout nœud ϱ du bas de l'arbre, c'est-à-dire de domaine égal à $\{A_0, A_1, \ldots, A_{n-1}\}$. On montre que c'est vrai pour tous les autres nœuds, autrement dit que $\Gamma, \Delta_\varrho \vdash \bot$ est dérivable pour tout ϱ de domaine $\{A_0, A_1, \ldots, A_{i-1}\}$, $0 \leq i \leq n$, par récurrence sur $n - i$. Nous venons de traiter le cas $n - i = 0$. Sinon, on produit :

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, \Delta_\varrho, \neg A_i \vdash \bot\end{array}}{\Gamma, \Delta_\varrho \vdash \neg\neg A_i}\,(\neg I) \quad \dfrac{\begin{array}{c}\vdots\\ \Gamma, \Delta_\varrho, A_i \vdash \bot\end{array}}{\Gamma, \Delta_\varrho \vdash \neg A_i}\,(\neg I)}{\Gamma, \Delta_\varrho \vdash \bot}\,(\neg E)$$

Finalement, lorsque $i = 0$, on obtient ainsi une dérivation de $\Gamma \vdash \bot$, et l'on conclut. □

▷ **Exercice 1.7**

On a établi $(*)$, dans la démonstration du théorème 1.6, en ne traitant que les cas où F est un atome, une conjonction ou une négation. Traiter des autres cas.

▷ **Exercice 1.8**

On considère des jugements infinis $S \vdash F$, où S est un ensemble, possiblement infini, de formules propositionnelles. On définit $\rho \models S \vdash F$ par si et seulement si $\rho \models F$ ou il existe une formule $G \in S$ telle que $\rho \not\models G$. On définit comme plus haut la notion de validité d'un jugement infini. Démontrer que **NK** est aussi correct et complet pour les jugements infinis, au sens où un jugement infini $S \vdash F$ est valide si et seulement s'il existe Γ fini inclus dans S tel que $\Gamma \vdash F$ est dérivable en **NK**.

1.4 Calcul des séquents, le système LK

On l'aura sans doute constaté, il n'est pas toujours facile de trouver une dérivation d'un jugement donné en **NK**. L'automatisation de la recherche de dérivation en **NK** est elle-même difficile (mais pas impossible : voir le cours de logique et informatique du second semestre). On pourrait en effet penser chercher une dérivation du bas vers le haut, en examinant quelles règles peuvent démontrer un jugement donné. Par exemple, pour démontrer $F_1 \vee F_2 \vdash F_2 \vee F_1$, on peut chercher à appliquer $(\vee I_1)$ (et il ne reste plus qu'à tenter de démontrer $F_1 \vee F_2 \vdash F_2$) ou $(\vee I_2)$ (et il ne reste plus qu'à tenter de démontrer $F_1 \vee F_2 \vdash F_1$). Mais de nombreuses

$$\frac{}{\Gamma, A \vdash A, \Delta}\,(Ax_{Atom}) \qquad \frac{\Gamma \vdash F, \Delta \quad \Gamma', F \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,(Cut)$$

$$\frac{}{\Gamma \vdash \top, \Delta}\,(\vdash \top) \qquad \frac{}{\Gamma, \bot \vdash \Delta}\,(\bot \vdash)$$

$$\frac{\Gamma \vdash F_1, \Delta \quad \Gamma \vdash F_2, \Delta}{\Gamma \vdash F_1 \wedge F_2, \Delta}\,(\vdash \wedge) \qquad \frac{\Gamma, F_1, F_2 \vdash \Delta}{\Gamma, F_1 \wedge F_2 \vdash \Delta}\,(\wedge \vdash)$$

$$\frac{\Gamma \vdash F_1, F_2, \Delta}{\Gamma \vdash F_1 \vee F_2, \Delta}\,(\vdash \vee) \qquad \frac{\Gamma, F_1 \vdash \Delta \quad \Gamma, F_2 \vdash \Delta}{\Gamma, F_1 \vee F_2 \vdash \Delta}\,(\vee \vdash)$$

$$\frac{\Gamma, F \vdash \Delta}{\Gamma \vdash \neg F, \Delta}\,(\vdash \neg) \qquad \frac{\Gamma \vdash F, \Delta}{\Gamma, \neg F \vdash \Delta}\,(\neg \vdash)$$

$$\frac{\Gamma, F_1 \vdash F_2, \Delta}{\Gamma \vdash F_1 \Rightarrow F_2, \Delta}\,(\vdash \Rightarrow) \qquad \frac{\Gamma, F_2 \vdash \Delta \quad \Gamma \vdash F_1, \Delta}{\Gamma, F_1 \Rightarrow F_2 \vdash \Delta}\,(\Rightarrow \vdash)$$

FIG. 4 – Le système de calcul des séquents **LK**

autres règles peuvent aussi s'appliquer. Par exemple, la dernière règle pourrait être $(\Rightarrow E)$ (et plus généralement, toute règle d'élimination), auquel cas il resterait à démontrer $F_1 \vee F_2 \vdash G \Rightarrow F_2 \vee F_1$ et $F_1 \vee F_2 \vdash G$, pour une certaine formule G à inventer. Dans cet exemple, on est en fait obligé d'utiliser comme dernière règle une règle d'élimination : il n'y aucun espoir de démontrer $F_1 \vee F_2 \vdash F_2$ ou $F_1 \vee F_2 \vdash F_1$ en général, car sinon on aurait $\models (F_1 \vee F_2) \Rightarrow F_2$, resp. $\models (F_1 \vee F_2) \Rightarrow F_1$. Le problème n'est pas tellement de savoir quelle règle d'élimination utiliser, mais de savoir quelle est la formule G à inventer qui mènera à une dérivation, si tant est qu'il y en ait une.

Pour corriger ce problème, Gentzen a ensuite inventé un autre style de système de déduction : le *calcul des séquents*. La principale différence avec la déduction naturelle est qu'au lieu de structurer la déduction autour de principes d'introduction et d'élimination des connecteurs logiques, et toujours à droite du signe $\vdash$, on ne va se donner que des règles d'introduction, mais des deux côtés de $\vdash$.

Pour des raisons techniques ou d'élégance, nous autoriserons désormais nos jugements à contenir plusieurs formules à droite de $\vdash$: $\Gamma \vdash \Delta$ aura alors la même sémantique que $\bigwedge \Gamma \Rightarrow \bigvee \Delta$, où $\bigvee \Delta$ est la disjonction des formules de Δ. (La virgule n'a donc pas le même sens à gauche et à droite du signe thèse.)

On définit donc un *séquent* comme étant une expression de la forme $\Gamma \vdash \Delta$, où Γ et Δ sont deux ensembles finis de formules propositionnelles. Il existe une autre variante, où Γ et Δ sont des multi-ensembles, nous y reviendrons plus tard. Les règles de déduction du système

LK du calcul des séquents classique, beaucoup plus symétriques que **NK**, sont montrées à la figure 4.

La règle axiome (Ax_{Atom}) est restreinte de sorte que A soit une formule atomique. Ceci n'a pas beaucoup d'importance. L'exercice 1.9 montre que la règle générale d'axiome est admissible. Réciproquement, nous montrerons que **LK**, tel que défini plus haut, reste complet malgré la restriction sur (Ax_{Atom}).

On note que l'on ne peut introduire $\top$ qu'à droite ($(\vdash \top)$) et $\bot$ qu'à gauche ($(\bot \vdash)$).

La règle (Cut) de *coupure* est désormais la seule où l'on ait besoin d'inventer une formule lors d'une recherche de dérivation de bas en haut — à savoir la formule F, dite *de coupure*. La bonne nouvelle est que (Cut) n'est en fait jamais nécessaire : le théorème d'élimination des coupures montrera que si un séquent est dérivable en **LK**, il l'est aussi sans jamais utiliser (Cut).

▷ **Exercice 1.9**
Montrer que la règle d'axiome générale :

$$\frac{}{\Gamma, F \vdash F, \Delta}\ (Ax_{Gnrl})$$

est admissible dans **LK**, avec ou sans (Cut).

Disons que ρ *satisfait* le séquent $\Gamma \vdash \Delta$, en notation $\rho \models \Gamma \vdash \Delta$, si et seulement si $\rho \not\models F$ pour une formule F de Γ, ou $\rho \models G$ pour une formule G de Δ. Autrement dit, si et seulement si ρ satisfait $\bigwedge \Gamma \Rightarrow \bigvee \Delta$. De façon équivalente, si ρ satisfait au moins une formule de Δ dès que ρ satisfait toutes les formules de Γ. Disons que $\Gamma \vdash \Delta$ est *valide* si et seulement si tout environnement le satisfait.

Lemme 1.7 (Correction) *Le système* **LK** *est* correct *: tout séquent dérivable est valide.*

Démonstration. Par récurrence sur la dérivation π. Traitons de (Cut) d'abord. Par hypothèse de récurrence sur la prémisse de gauche, pour tout environnement ρ satisfaisant toutes les formules de Γ et de Γ', ρ satisfait une formule de F, Δ. Si $\rho \models F$, alors par hypothèse de récurrence sur la prémisse de droite, ρ satisfait une formule de Δ'. Sinon, ρ satisfait une formule de Δ. Dans les deux cas, ρ satisfait bien une formule de Δ, Δ'. Les autres cas sont au moins aussi faciles, et laissés en exercice. □

▷ **Exercice 1.10**
Une autre façon de montrer que **LK** est correct est de traduire les dérivations de **LK** en dérivations de **NK**. Montrer que l'on peut (et même algorithmiquement, et en temps polynomial) transformer toute dérivation π de $\Gamma \vdash \Delta$ dans **LK** en une dérivation de $\Gamma, \neg\Delta \vdash \bot$ dans **NK**, où $\neg\Delta$ est l'ensemble des négations de formules de Δ.

Comme promis, on peut démontrer la complétude, même sans utiliser la règle (Cut).

Théorème 1.8 (Complétude) *Le système* **LK***, et même son sous-système* $\mathbf{LK}_{\mathrm{cf}}$ *obtenu en interdisant l'utilisation de la coupure* (Cut)*, est complet : tout séquent valide y est dérivable.*

Démonstration. On produit une démonstration du séquent valide $\Gamma \vdash \Delta$ par récurrence sur la taille de $\Gamma \vdash \Delta$, où la taille est définie par : $|A| = 1$ pour tout $A \in \mathit{Atom}$, $|\top| = |\bot| = 1$, $|\neg F| = |F| + 1$, $|F_1 \wedge F_2| = |F_1 \vee F_2| = |F_1 \Rightarrow F_2| = |F_1| + |F_2| + 1$, $|\Gamma| = \sum_{F \in \Gamma} |F|$, $|\Gamma \vdash \Delta| = |\Gamma| + |\Delta|$. On notera que, comme Γ est un ensemble, toute formule F n'y apparaît qu'une fois. Par exemple, si l'on cherche à démontrer $F_1, F_1 \wedge F_2 \vdash F_3$ en utilisant $(\wedge \vdash)$, la prémisse sera $F_1, F_2 \vdash F_3$ (les deux copies de F_1 à gauche étant fusionnées).

Par "récurrence", nous entendons ici le principe de récurrence complète : pour démontrer une propriété d'un séquent de taille n, il suffit de la démontrer en présence de l'hypothèse de récurrence énonçant qu'elle est vraie de tout séquent de taille strictement inférieure à n. Ce principe de récurrence complète est équivalent au principe de récurrence usuel.

Si $\Gamma \vdash \Delta$ est un séquent *atomique*, c'est-à-dire tel que $\Gamma, \Delta \subseteq \mathit{Atom}$, on vérifie qu'il est nécessairement dérivable par une instance de (Ax_{Atom}). Sinon, Γ n'intersecterait pas Δ, et l'environnement partiel qui à tout atome de Γ associe 1 et à tout atome de Δ associe 0 serait bien défini, et ne satisferait pas $\Gamma \vdash \Delta$. Donc (Ax_{Atom}) s'applique.

Sinon, une des formules de Γ ou de Δ a un connecteur de tête, et l'on peut appliquer la règle correspondante ($(\wedge \vdash)$ si c'est $\wedge$, dans Γ, etc.). Il est facile de voir que toutes les prémisses de cette règle sont strictement plus petites que $\Gamma \vdash \Delta$. D'autre part, toutes les règles autres que (Ax_{Atom}) et (Cut) sont *inversibles* : si la conclusion est valide, toutes les prémisses sont valides. On peut donc appliquer l'hypothèse de récurrence et conclure. □

Corollaire 1.9 (Élimination des coupures, version faible) *Tout jugement dérivable en* **LK** *est dérivable en* $\mathbf{LK}_{\mathrm{cf}}$*, c'est-à-dire sans utiliser la coupure* (Cut)*.*

Démonstration. Tout jugement dérivable en **LK** est valide par le lemme 1.7, donc dérivable en $\mathbf{LK}_{\mathrm{cf}}$ par le théorème 1.8. □

▷ **Exercice 1.11**

Le principe de récurrence sur les entiers naturels est : pour toute propriété P des entiers, si $P(0)$ et $P(n)$ implique $P(n+1)$ pour tout $n \in \mathbb{N}$, alors $P(n)$ pour tout $n \in \mathbb{N}$:

$$P(0) \wedge (\forall n \in \mathbb{N} \cdot P(n) \Rightarrow P(n+1)) \Rightarrow \forall n \in \mathbb{N} \cdot P(n)$$

Le *principe de récurrence complète* est, lui :

$$(\forall n \in \mathbb{N} \cdot (\forall m < n \cdot P(m)) \Rightarrow P(n)) \Rightarrow \forall n \in \mathbb{N} \cdot P(n)$$

Autrement dit, disons qu'une propriété P est *héréditaire* si et seulement si, pour tout entier n, $P(n)$ est vrai dès que $P(m)$ est vrai pour tout entier $m < n$; alors le principe de récurrence complète énonce que toute propriété héréditaire est vraie de tout entier.

Démontrer que ces deux principes sont équivalents, au sens où l'on peut démontrer l'un à partir de l'autre à l'aide uniquement de principes logiques élémentaires, et des seuls faits arithmétiques suivants :

- tout entier est de la forme 0 ou $n+1$, $n \in \mathbb{N}$;
- $m < 0$ n'est vrai pour aucun $m \in \mathbb{N}$;
- si $m < n+1$ est vrai, alors $m = n$ ou bien $m < n$;
- $m < m+1$ pour tout $m \in \mathbb{N}$.

Indication : on pourra considérer la propriété $Q(n) = (\forall m \leq n \cdot P(m))$, où $m \leq n$ abrège "$m < n$ ou $m = n$". Cet exercice justifie notre utilisation du principe de récurrence complète dans la démonstation du théorème 1.8, avec $P(n) =$ "tout séquent valide de taille n est dérivable en **NK**".

▷ **Exercice 1.12**

Une autre façon de montrer que **LK** est complet est de traduire les dérivations de **NK** en dérivations de **LK**, et d'utiliser le théorème de complétude de **NK**. (Ceci n'a bien sûr que peu d'intérêt en tant que tel, vu la complexité relative des deux démonstrations de complétude.) Montrer que cette traduction est algorithmique, et se fait en temps polynomial, à condition d'utiliser la coupure.

1.5 Élimination des coupures

Le théorème d'élimination des coupures se démontre en général différemment, par un système de réécriture des preuves. Cette façon de faire a l'avantage de montrer une version plus forte : il existe un *algorithme* (une fonction totale récursive) qui transforme toute dérivation en **LK** en une dérivation du même jugement sans coupure. Dans le cas d'une logique aussi simple que la logique propositionnelle, ceci n'a que peu d'intérêt. La procédure d'élimination des coupures a été imaginée par Gentzen en 1934 pour traiter d'un problème bien plus difficile : l'arithmétique de Peano du premier ordre $\mathbf{PA}_1$ admet un calcul des séquents, et l'élimination des coupures démontre d'une part que $\mathbf{PA}_1$ est non contradictoire ; et d'autre part que l'on peut démontrer tout principe de récurrence le long d'un ordinal $\alpha < \epsilon_0$, mais pas la récurrence selon ϵ_0 lui-même. Pour plus de détails, consulter Schwichtenberg [11].

Voici comment cette procédure d'élimination des coupures fonctionne ; le cas propositionnel renferme en fait l'essentiel des difficultés présentes pour le cas plus intéressant de $\mathbf{PA}_1$. En se référant à la figure 4, appelons la formule atomique A dans (Ax_{Atom}) la *formule d'axiome* ; la formule F dans (Cut) est la *formule de coupure* ; dans les autres règles, la formule distinguée dans la conclusion ($\top$ dans $(\vdash \top)$, $F_1 \wedge F_2$ dans $(\vdash \wedge)$ et $(\wedge \vdash)$, etc.) est la *formule principale*, et les formules distinguées (F_1, F_2 ; F dans $(\vdash \neg)$ et $(\neg \vdash)$; aucune dans $(\vdash \top)$ et $(\bot \vdash)$) sont les *formules actives*.

La procédure d'élimination des coupures réécrit toute dérivation de sorte à faire remonter les instances de (Cut). Lorsqu'une instance de (Cut) est remontée suffisamment haut pour que l'une des prémisses soit une instance de (Ax_{Atom}), elle disparaît par l'une des règles de réécriture :

$$\dfrac{\dfrac{}{\Gamma, A \vdash A, \Delta}\,(Ax_{Atom}) \qquad \begin{array}{c}\vdots\,\pi\\ \Gamma', A \vdash \Delta'\end{array}}{\Gamma, \Gamma', A \vdash \Delta, \Delta'}\,(Cut) \quad \rightsquigarrow \quad \begin{array}{c}\vdots\,\pi'\\ \Gamma, \Gamma', A \vdash \Delta, \Delta'\end{array} \tag{1}$$

$$\dfrac{\dfrac{}{\Gamma, A \vdash A, F, \Delta}\,(Ax_{Atom}) \qquad \begin{array}{c}\vdots\,\pi\\ \Gamma', F \vdash \Delta'\end{array}}{\Gamma, A, \Gamma' \vdash A, \Delta, \Delta'}\,(Cut) \quad \rightsquigarrow \quad \dfrac{}{\Gamma, A, \Gamma' \vdash A, \Delta, \Delta'}\,(Ax_{Atom})$$

où π' est obtenu à partir de π par affaiblissement — lequel est admissible en **LK** et en $\mathbf{LK}_{\mathrm{cf}}$. Le premier cas est celui où la formule de coupure est la formule d'axiome A, le second cas est celui où la formule de coupure est une autre formule. Nous n'avons représenté que les

deux cas où c'est la prémisse gauche de (Cut) qui est obtenue par (Ax_{Atom}). Les deux cas où c'est la prémisse droite sont similaires.

Tant qu'il reste une instance de (Cut) dans une dérivation donnée π, il existe une instance de (Cut) la plus haute. Ses deux prémisses sont alors obtenues par des dérivations sans coupure. Il nous reste donc à examiner les cas de coupures entre deux règles autres que l'axiome ou la coupure — lesquelles ont donc des formules principales.

On évacue d'abord un cas trivial : celui où l'une des deux prémisses a une formule principale qui n'est pas celle de coupure. Dans ce cas, la règle de coupure permute simplement au-dessus de la règle utilisée pour dériver cette prémisse, en dupliquant éventuellement certaines dérivations. Par exemple, si cette règle est $(\vdash \wedge)$, on opère la réécriture :

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi_1}{\Gamma \vdash F_1, G, \Delta} \quad \dfrac{\vdots\,\pi_2}{\Gamma \vdash F_2, G, \Delta}}{\Gamma \vdash F_1 \wedge F_2, G, \Delta}(\vdash \wedge) \quad \dfrac{\vdots\,\pi_3}{\Gamma', G \vdash \Delta'}}{\Gamma, \Gamma' \vdash F_1 \wedge F_2, \Delta, \Delta'}(Cut) \tag{2}$$

$$\rightsquigarrow \dfrac{\dfrac{\dfrac{\vdots\,\pi_1}{\Gamma \vdash F_1, G, \Delta} \quad \dfrac{\vdots\,\pi_3}{\Gamma', G \vdash \Delta'}}{\Gamma, \Gamma' \vdash F_1, \Delta, \Delta'}(Cut) \quad \dfrac{\dfrac{\vdots\,\pi_2}{\Gamma \vdash F_2, G, \Delta} \quad \dfrac{\vdots\,\pi_3}{\Gamma', G \vdash \Delta'}}{\Gamma, \Gamma' \vdash F_2, \Delta, \Delta'}(Cut)}{\Gamma, \Gamma' \vdash F_1 \wedge F_2, \Delta, \Delta'}(\vdash \wedge)$$

On note tout de même que ceci peut remplacer une instance de (Cut) par plusieurs.

Il ne reste alors qu'une famille de cas, la plus intéressante : celle où la formule de coupure est principale dans les *deux* prémisses. Ceci bloque le processus de remontée de la coupure. Pour continuer, nous devons en quelque sorte dissoudre le bloquage. Dans tous les cas, ceci supprimera les instances des règles gauche et droite introduisant les prémisses de la coupure, mais en introduisant de nouvelles instances de (Cut). Selon que le connecteur principal de la formule de coupure est $\wedge$, $\vee$, $\neg$, ou $\Rightarrow$, on opère les transformations décrites à la figure 5 (les cas $\top$ et $\bot$ ne se présentent pas : pourquoi ?).

On en déduit le résultat souhaité :

Proposition 1.10 (Élimination des coupures) *Il existe une machine de Turing qui, sur toute dérivation π d'un séquent en* **LK***, termine et calcule une dérivation du même séquent en* $\mathbf{LK}_{\mathrm{cf}}$*, c'est-à-dire sans coupure.*

Démonstration. La difficulté principale est de montrer que les règles de transformation définies ci-dessus *terminent*. Il se trouve que, quelle que soit la stratégie de choix d'une coupure à faire remonter à chaque étape, le processus termine effectivement, mais c'est relativement difficile à démontrer. (Un tel résultat sera conséquence des résultats du cours de logique et informatique, au second semestre.) À la place, nous allons démontrer que la stratégie qui consiste à faire remonter les coupures les plus hautes, c'est-à-dire entre deux

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi_1}{\Gamma\vdash F_1,\Delta}\quad\dfrac{\vdots\,\pi_2}{\Gamma\vdash F_2,\Delta}}{\Gamma\vdash F_1\wedge F_2,\Delta}(\vdash\wedge)\quad\dfrac{\vdots\,\pi_3}{\Gamma',F_1,F_2\vdash\Delta'}(\wedge\vdash)}{\Gamma,\Gamma'\vdash\Delta,\Delta'}(Cut)\quad\rightsquigarrow\quad\dfrac{\dfrac{\vdots\,\pi_2}{\Gamma\vdash F_2,\Delta}\quad\dfrac{\dfrac{\vdots\,\pi_1}{\Gamma\vdash F_1,\Delta}\quad\dfrac{\vdots\,\pi_3}{\Gamma',F_1,F_2\vdash\Delta'}}{\Gamma,F_2,\Gamma'\vdash\Delta,\Delta'}(Cut)}{\Gamma,\Gamma'\vdash\Delta,\Delta'}(Cut)$$

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi_3}{\Gamma\vdash F_1,F_2,\Delta}}{\Gamma\vdash F_1\vee F_2,\Delta}(\vdash\vee)\quad\dfrac{\dfrac{\vdots\,\pi_1}{\Gamma',F_1\vdash\Delta'}\quad\dfrac{\vdots\,\pi_2}{\Gamma',F_2\vdash\Delta'}}{\Gamma',F_1\vee F_2\vdash\Delta'}(\vee\vdash)}{\Gamma,\Gamma'\vdash\Delta,\Delta'}(Cut)\quad\rightsquigarrow\quad\dfrac{\dfrac{\dfrac{\vdots\,\pi_3}{\Gamma\vdash F_1,F_2,\Delta}\quad\dfrac{\vdots\,\pi_1}{\Gamma',F_1\vdash\Delta'}}{\Gamma,\Gamma'\vdash F_2,\Delta,\Delta'}(Cut)\quad\dfrac{\vdots\,\pi_2}{\Gamma',F_2\vdash\Delta'}}{\Gamma,\Gamma'\vdash\Delta,\Delta'}(Cut)$$

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi_1}{\Gamma,F\vdash\Delta}}{\Gamma\vdash\neg F,\Delta}(\vdash\neg)\quad\dfrac{\dfrac{\vdots\,\pi_2}{\Gamma'\vdash F,\Delta'}}{\Gamma',\neg F\vdash\Delta'}(\neg\vdash)}{\Gamma,\Gamma'\vdash\Delta,\Delta'}(Cut)\quad\rightsquigarrow\quad\dfrac{\dfrac{\vdots\,\pi_2}{\Gamma'\vdash F,\Delta'}\quad\dfrac{\vdots\,\pi_1}{\Gamma,F\vdash\Delta}}{\Gamma,\Gamma'\vdash\Delta,\Delta'}(Cut)$$

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi_3}{\Gamma,F_1\vdash F_2,\Delta}}{\Gamma\vdash F_1\Rightarrow F_2,\Delta}(\vdash\Rightarrow)\quad\dfrac{\dfrac{\vdots\,\pi_2}{\Gamma',F_2\vdash\Delta'}\quad\dfrac{\vdots\,\pi_1}{\Gamma'\vdash F_1,\Delta'}}{\Gamma',F_1\Rightarrow F_2\vdash\Delta'}(\Rightarrow\vdash)}{\Gamma,\Gamma'\vdash\Delta,\Delta'}(Cut)\quad\rightsquigarrow\quad\dfrac{\dfrac{\vdots\,\pi_1}{\Gamma'\vdash F_1,\Delta'}\quad\dfrac{\dfrac{\vdots\,\pi_2}{\Gamma',F_2\vdash\Delta'}\quad\dfrac{\vdots\,\pi_3}{\Gamma,F_1\vdash F_2,\Delta}}{\Gamma,F_1,\Gamma'\vdash\Delta,\Delta'}(Cut)}{\Gamma,\Gamma'\vdash\Delta,\Delta'}(Cut)$$

FIG. 5 – Les principaux cas dans l'élimination des coupures en **LK**

dérivations sans coupure, termine. La difficulté principale est que les règles de transformation définies plus haut peuvent remplacer une coupure par plusieurs.

On démontre donc en premier que l'on peut transformer toute dérivation π qui se termine par une instance de la coupure entre deux prémisses dérivées sans coupure, en une dérivation sans coupure. Ceci se démontre par récurrence sur le couple $(|G|, |\pi|)$ de la taille $|G|$ de la formule de coupure G et de la taille $|\pi|$ de π, ordonné dans l'ordre lexicographique $<_{lex}$ ($(m, n) <_{lex} (m', n')$ si et seulement si $m < m'$, ou bien $m = m'$ et $n < n'$). Ceci est correct, car l'ordre lexicographique est bien fondé. Une autre façon de le dire est que nous opérons une *récurrence double*, c'est-à-dire deux raisonnements par récurrence imbriqués. Autrement dit, nous montrons par récurrence sur m que : $(*)$ toute dérivation π qui se termine par une instance de coupure entre deux prémisses dérivées sans coupure, et dont la formule de coupure G est telle que $|G| = m$, se transforme par les transformations définies plus haut en une dérivation sans coupure. Pour ceci, à m fixé, on démontre $(*)$ par récurrence sur la taille de π. L'hypothèse de la récurrence externe est que $(*)$ est vrai pour toute formule de coupure de taille strictement plus petite que m. L'hypothèse de la récurrence interne est que $(*)$ est vrai lorsque la formule de coupure est de taille exactement m, mais où la taille de dérivation est strictement plus petite que celle de π. Autrement dit, pour démontrer le résultat $(*)$ qui nous intéresse, il suffit de le démontrer, sous l'hypothèse de récurrence combinée que $(*)$ est vrai pour toute formule de coupure G' et toute dérivation π' de la forme indiquée avec $(|G'|, |\pi'|) <_{lex} (|G|, |\pi|)$.

Si l'une des prémisses de la coupure à la fin de π est obtenue par (Ax_{Atom}), les transformations (1) et leurs variantes fournissent directement une dérivation sans coupure. Sinon, et si l'une des prémisses de la coupure a pour formule principale une autre formule que la formule de coupure (par exemple la transformation (2)), on conclut par l'hypothèse de récurrence (interne), la formule de coupure restant la même, mais la taille de la dérivation diminuant. Finalement, si les deux prémisses ont comme formule principale la formule de coupure (les cas de la figure 5), on conclut par l'hypothèse de récurrence (externe), la formule de coupure étant strictement plus petite dans chacune des coupures engendrées sur le côté droit du signe de réécriture $\rightsquigarrow$.

Ayant démontré $(*)$, on démontre le résultat par récurrence sur le nombre d'instances de (Cut) dans la dérivation donnée π. S'il n'y en a pas, on démontré la proposition. Sinon, on considère une coupure la plus haute possible, on l'élimine grâce à $(*)$, puis on applique l'hypothèse de récurrence sur la dérivation transformée. □

1.6 Une méthode de démonstration automatique par tableaux

La démonstration du théorème 1.8 utilise le fait que toutes les règles de $\mathbf{LK}_{\mathrm{cf}}$ sont inversibles. On en déduit une procédure très simple de recherche de dérivation d'un séquent : si le séquent est atomique, accepter s'il existe un même atome des deux côtés du séquent ; sinon, choisir une formule non atomique dans le séquent (peu importe laquelle), appliquer la règle correspondante, et chercher à démontrer les prémisses par un appel récursif. Concrètement, on peut écrire, en Caml, le programme suivant :

```
type form = A of string
  | FAUX
  | VRAI
  | NON of form
  | ET of form * form
  | OU of form * form
  | IMP of form * form;;

type sign = G | D;; (* a gauche ou a droite du sequent *)

type sequent = (sign * form) list;;

let rec prove1 seq g d =
    match seq with
        [] -> false
      | (D, A x)::rest ->
        List.mem x g || prove1 rest g (x::d)
      | (G, A x)::rest ->
        List.mem x d || prove1 rest (x::g) d
      | (D, VRAI)::rest -> true
      | (G, FAUX)::rest -> true
      | (D, NON f)::rest -> prove1 ((G,f)::rest) g d
      | (G, NON f)::rest -> prove1 ((D,f)::rest) g d
      | (D, ET (f1,f2))::rest ->
        prove1 ((D,f1)::rest) g d && prove1 ((D,f2)::rest) g d
      | (G, ET (f1,f2))::rest ->
        prove1 ((G, f1)::(G,f2)::rest) g d
      | (D, OU (f1,f2))::rest ->
        prove1 ((D,f1)::(D,f2)::rest) g d
      | (G, OU (f1,f2))::rest ->
        prove1 ((G,f1)::rest) g d && prove1 ((G,f2)::rest) g d
      | (D, IMP (f1,f2))::rest ->
        prove1 ((D,f2)::(G,f1)::rest) g d
      | (G, IMP (f1,f2))::rest ->
        prove1 ((G,f2)::rest) g d && prove1 ((D,f1)::rest) g d;;

let prove seq = prove1 seq [] [];;
```

Le type `form` est celui des formules propositionnelles. Le type `sequent` représente un séquent $\Gamma \vdash \Delta$, avec éventuellement des formules répétées : Γ est l'ensemble des formules F telles que $(\texttt{G}, F) \in \texttt{sequent}$ (`G` comme "gauche"), Δ l'ensemble des formules G telles que $(\texttt{D}, G) \in \texttt{sequent}$ (`D` comme "droite"). Essayez par exemple :

```
prove [(G, OU (A "A1", A "A2")); (D, OU (A "A2", A "A1"))];;
```

pour vérifier que $A_1 \vee A_2 \vdash A_2 \vee A_1$ est dérivable en $\mathbf{LK}_{\mathrm{cf}}$.

La fonction `prove1` est appelée sur trois arguments, `seq` de type `sequent`, `g` et `d` de type `string list`. Un invariant est que `g` et `d` sont des listes de (noms de) formules atomiques d'intersection vide, et `prove1` tente de trouver une dérivation du séquent obtenu à partir de `seq` en ajoutant à gauche tous les atomes de `g` et à droite tous ceux de `d`. On notera (les deux appels à `List.mem`) que l'on teste l'applicabilité de (Ax_{Atom}) dès qu'on le peut.

On a dit plus haut qu'un `sequent` peut contenir des formules dupliquées, alors que nos séquents ne le peuvent pas. Nos théorémes de correction et de complétude pour $\mathbf{LK}_{\mathrm{cf}}$ ne s'appliquent donc pas tels quels pour établir que la fonction `prove` est correcte et complète. Cependant, la démonstration de complétude est suffisamment simple pour s'adapter facilement au cas de la fonction `prove`.

La fonction `prove` est une instance de la famille des techniques de démonstration automatique par *tableaux*. Une *branche* est un ensemble ou une liste finie de formules signées, et représente un séquent. Un *tableau* est un ensemble fini de branches. Une méthode par tableaux choisit une branche, et sur cette branche une formule signée à expanser, d'une façon ou d'une autre. L'expansion de certaines formules se contente d'allonger la branche, par exemple la règle $(\vdash \vee)$ revient à remplacer une formule signée $(\mathtt{D}, F_1 \vee F_2)$ par deux formules signées $(\mathtt{D}, F_1)$ et $(\mathtt{D}, F_2)$. D'autres découpent la branche en plusieurs, par exemple la règle $(\vdash \wedge)$ remplace une branche $S \cup \{(\mathtt{D}, F_1 \wedge F_2)\}$ par deux branches $S \cup \{(\mathtt{D}, F_1)\}$ et $S \cup \{(\mathtt{D}, F_2)\}$.

2 Les classes P, NP, et le problème SAT

La fonction `prove` fonctionne, dans le pire des cas, en temps exponentiel. En fait, malgré les efforts que nous avons déployés, il y a même des cas où elle fonctionnera plus longtemps que l'algorithme naïf. Considérons par exemple les formules $\pm A_1 \vee \ldots \vee \pm A_n$, où A_1, …, A_n sont n atomes distincts, et les signes $\pm$ désignent soit la présence soit une absence de négation. (On considère un parenthésage arbitraire.) Il y a 2^n formules de ce type, à associativité et commutativité près. Soit Γ l'ensemble de ces 2^n formules. On vérifie que $\Gamma \vdash \bot$ est valide : pour tout environnement partiel ϱ de domaine $\{A_1, \ldots, A_n\}$, la formule obtenue en mettant un signe $\neg$ devant A_i si $\varrho(A_i) = 1$ et aucun si $\varrho(A_i) = 0$ est dans Γ, et est fausse dans ϱ. La méthode des tables de vérité énumère 2^n environnements partiels, et évalue la valeur de vérité de $\bigwedge \Gamma \Rightarrow \bot$ en un temps de la forme $p(n)2^n$, où $p(n)$ est un polynôme en n, représentant essentiellement le temps d'évaluation de chaque formule. Le méthode des tables de vérité prend donc un temps $p(n)2^{2n}$. En particulier, elle ne prend ici qu'un temps *polynomial* (quadratique en gros) en la taille de la formule $\bigwedge \Gamma \Rightarrow \bot$ en entrée.

La méthode des tableaux, dans le pire des cas, va devoir appliquer la règle $(\vee \vdash)$ sur chacune des 2^n formules de Γ, $n-1$ fois pour chacune, ce qui va fournir n^{2^n} séquents atomiques, qui seront tous des instances de (Ax_{Atom}) (par complétude). Même si l'on cherche une stratégie intelligente d'application des règles, on peut montrer qu'une dérivation de $\Gamma \vdash \bot$ contient nécessairement au moins $n!$ instances de (Ax_{Atom}) [3]. Or $n! \sim e^{n \log n - n}\sqrt{2\pi n}$ par la formule de Stirling, et ceci n'est *pas* polynomial en 2^n. (Tous les polynômes en 2^n sont

majorés par un 2^{kn}, où k est une constante ; mais $\log n$ tend vers $+\infty$.)

Soyons rassurés : il existe aussi des familles de formules telle que toute démonstration par table de vérité prend un temps super-polynomial en le temps pris par une méthode de tableaux.

En tout cas, il est important de réaliser qu'il existe une énorme différence de rapidité entre un algorithme en temps polynomial et un algorithme en temps exponentiel. En tant qu'expérience de pensée, voici le temps pris par un algorithme en temps n, n^2, n^3, n^{10}, et 2^n, l'unité de temps étant la pico-seconde (1 ps $= 10^{-12}$ s), pour quelques valeurs de la taille n de l'entrée :

n	10	20	30	40	50	60	70	80	90	100
n	10ps	20ps	30ps	40ps	50ps	60ps	70ps	80ps	90ps	100ps
n^2	100ps	400ps	900	1,6ns	2,5ns	3,6ns	4,9ns	6,4ns	8,1ns	10ns
n^3	1ns	8ns	27ns	64ns	125ns	216ns	343ns	512ns	729ns	1μs
n^{10}	10ms	10,24s	9,8min	2,9h	1,13j	7,0j	32,7j	4,14mois	1,1an	3,17ans
2^n	1,0ns	1,0μs	1,1ms	1,1s	18,8min	13,3j	37,4ans	38,3k.ans	39,3M.ans	2,7univers

où 1 univers = 15 milliards d'années, la durée de vie actuelle de l'univers. Au vu de la différence de temps entre les algorithmes en temps polynomial (même avec un degré de l'ordre de 10) et ceux en temps exponentiel, on considère qu'un langage est *tractable*, c'est-à-dire décidable efficacement, si et seulement si on peut le décider en temps polynomial, c'est-à-dire majoré par un polynôme fixé en la taille n de l'entrée. La classe des langages tractables est notée **P**.

2.1 La classe NP et SAT

Il reste que nous ne connaissons aucune méthode de démonstration automatique des formules propositionnelles qui termine en temps polynomial. Autrement dit, on ne sait pas si FORM-SAT est dans **P**.

En revanche, si l'on s'autorise le non-déterminisme, c'est facile. Rappelons (partie 1 du cours) qu'une machine de Turing non déterministe $\mathcal{M}$ est un quintuplet $(Q, q_0, \Sigma, \delta, \{B, \$\})$, où Q est un ensemble fini dit d'*états internes* (ou *de contrôle*), $q_0 \in Q$ est l'*état initial*, Σ est l'alphabet de bandes[1], et la *relation de transition* δ est une fonction de $Q \times \Sigma$ vers $\mathbb{P}^*(Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\})$, où $\mathbb{P}^*$ désigne l'ensemble des parties non vides. (On demande de plus que si $(q', a', dir) \in \delta(q, \$)$, alors $dir \neq \leftarrow$, autrement dit on ne va pas à gauche du marqueur de début de bande, et si $dir = \downarrow$ alors $a' = \$$, c'est-à-dire qu'on ne réécrit jamais le marqueur de début de bande.) Une configuration de $\mathcal{M}$ est un triplet $\gamma = (w, q, aw')$ où $w \in \$\Sigma^*$, $w' \in \Sigma^*$, $a \in \Sigma$, $q \in Q \cup \{\mathbf{accept}, \mathbf{reject}\}$. La machine évolue à partir d'une configuration γ en lisant le caractère a sous la tête, en *devinant* ensuite un triplet (q', a', dir) tel que $(q', a', dir) \in \delta(q, a)$, puis passe à l'état interne q' en écrivant a' sous la tête, enfin se déplace dans la direction dir ; si $dir = \leftarrow$, en écrivant $w = w_1 c$, on passe à la configuration $(w_1, q', ca'w')$; si $dir = \downarrow$, on passe à $(w, q', a'w')$; si $dir = \rightarrow$, on passe à (wa', q', w') si w'

[1] J'appellerai bande ce que vous avez appelé ruban dans la première partie du cours. C'est un autre nom pour la même chose.

est non vide, ou à (wa', q', B) si w' est vide (on ajoute un blanc). La configuration initiale sur l'entrée x est $(\epsilon, q_0, \$x)$.

On dira qu'un langage est dans **NP** (Non déterministe Polynomial) si et seulement s'il est décidable par une machine de Turing *non déterministe* en temps polynomial. Une exécution d'une machine de Turing non déterministe est une suite maximale de configurations γ_0, γ_1, ..., γ_k, ..., où γ_0 est une configuration initiale, et γ_i est reliée à γ_{i+1} pour tout i par la relation d'évolution décrite ci-dessus. Une telle exécution est *temps t* si et seulement si elle est finie, et de longueur au plus t, c'est-à-dire de la forme γ_0, γ_1, ..., γ_k avec $k \leq t$ et γ_k est une configuration acceptante ou rejetante. (Nous considérons ici que **accept** et **reject** sont des états internes de la machine, ce qui simplifie la définition. Une configuration est acceptante si et seulement si son état interne est **accept**, rejetant si son état interne est **reject**.) Une machine de Turing *termine en temps t* sur l'entrée x si et seulement *toute* exécution partant de la configuration initiale sur l'entrée x est en temps t. Une machine de Turing est *en temps* $f(n)$ si et seulement si elle termine en temps $f(n)$ sur toute entrée x, où n est la taille de x. Elle est *en temps polynomial* si et seulement s'il existe un polynôme p telle qu'elle soit en temps $p(n)$.

Finalement, le langage L est dans **NP** si et seulement s'il existe une machine de Turing non déterministe en temps polynomial telle que, pour toute entrée x, $x \in L$ si et seulement s'*il existe* une exécution de la machine qui part de la configuration initiale sur l'entrée x et qui about à une configuration acceptante, c'est-à-dire dont l'état interne est **accept**.

Proposition 2.1 *Le problème FORM-SAT de la satisfiabilité des formules propositionnelles :*
ENTRÉE : une formule propositionnelle F ;
QUESTION : F est-elle satisfiable ?
est dans la classe **NP** *des problèmes décidables en temps polynomial sur une machine de Turing non déterministe.*

Démonstration. La machine devine les valeurs de vérité de chaque atome de F, puis évalue F.

Plus formellement, nous supposerons que les atomes de F sont numérotés de 1 à n, et écrits en binaire. Sinon, il est facile de transformer F sous une telle forme, en gardant sur une bande la liste d'association de chaque variable à son numéro. la machine non déterministe que nous construisons va réserver une bande de travail β pour contenir les valeurs de vérité d'un environnement partiel ϱ. Dans une première phase, la machine lit son entrée, et va calculer le numéro n de la plus grande variable apparaissant dans F (0 si F n'a pas de variable libre). Représentons ce numéro en unaire sur la bande β. La machine devine ensuite n valeurs booléennes, vrai ou faux, et en remplit les n cases de β. (Ce sera le seul endroit où la machine utilisera le non-déterminisme.) Puis la machine calcule la valeur de vérité de F, en relisant son entrée et en consultant la bande β pour connaître la valeur des atomes. Finalement, elle accepte si la valeur de F est 1, elle rejette sinon. Pour calculer la valeur de vérité de F, un programme récursif est ce qui est le plus pratique. On simule la récursivité en gérant la pile de récursion sur une deuxième bande auxiliaire. (Le projet de programmation I a dû vous convaincre que c'était possible.) Finalement, le temps pris est clairement polynomial en la taille de F, qui est supérieure ou égale à n. □

Nous avons utilisé dans la démonstration des machines de Turing (non déterministes) à k bandes, dont une d'entrée à lecture seule, et une, éventuelle, de sortie, en écriture seule. (Les autres bandes, et il nous en faut au moins une, s'appellent les *bandes de travail*.) Or nous avons défini les machines de Turing sur une seule bande. Ceci n'a aucune importance, par la proposition suivante. On définit les machines non déterministes à k bandes dont une d'entrée et une de sortie comme les machines déterministes du même type, en remplaçant comme ci-dessus la fonction de transition par une relation de transition.

Proposition 2.2 *Les langages décidables en temps polynomial sur une machine déterministe, resp. non déterministe, à k bandes dont une d'entrée et éventuellement une de sortie (mais au moins une de travail) sont exactement ceux décidables en temps polynomial sur une machine de Turing déterministe, resp. non déterministe, à une bande.*

Démonstration. On a déjà vu l'argument pour les machines déterministes, et sans prêter attention à la complexité, dans la partie 1 du cours. Nous redonnons l'essentiel de l'argument. Nous montrons plus généralement que l'on peut simuler une machine de Turing $\mathcal{M}$ à k bandes dont une d'entrée et éventuellement une de sortie, qui fonctionne en temps $f(n)$, par une machine ordinaire à une bande $\mathcal{M}'$, en temps $O(f(n)^2)$. $\mathcal{M}'$ sera déterministe ou non selon que $\mathcal{M}$ l'est ou non.

D'abord, pour décider d'un langage, on peut ignorer la bande de sortie si elle est présente, puisque qu'elle est en écriture seule. Supposons donc que $\mathcal{M}$ n'a pas de bande de sortie. Une configuration de $\mathcal{M}$ est donnée par un état interne $q \in Q \cup \{\mathbf{accept}, \mathbf{reject}\}$, et des mots w_0, a_0w_0' (bande d'entrée, séparée au niveau de la tête), $w_1, a_1w_1', \ldots, w_{k-1}, a_{k-1}w_{k-1}'$ (les $k-1$ bandes de travail). On code cette configuration par le mot $w_0\#a_0w_0' \dagger\, w_1\#a_1w_1' \dagger \ldots \dagger\, w_{k-1}\#a_{k-1}w_{k-1}'\dagger$, codé sur l'unique bande de $\mathcal{M}'$, où $\#$ et $\dagger$ sont deux nouveaux symboles, distincts. Pour simuler une étape de $\mathcal{M}$, d'abord (étape 1) $\mathcal{M}'$ parcourt le mot de gauche à droite et collecte dans son état interne les lettres $a_1, \ldots, a_{k-1}$ (l'état interne de $\mathcal{M}'$ est donc un n-uplet contenant non seulement l'état interne q de la machine simulée $\mathcal{M}$, mais aussi un $(k-1)$-uplet de lettres de Σ, et en général d'autres composantes, voir la partie 1 du cours). Il se peut que certaines des bandes simulées doivent être étendues par un blanc B (si la tête sur cette bande est à l'extrémité droite, et la machine souhaite se déplacer encore à droite). Pour simplifier la démonstration, en étape 2, nous allons faire revenir $\mathcal{M}'$ à gauche de sa bande en insérant un blanc devant chaque symbole $\dagger$, pour être sûr que les mouvements à droite de l'étape 3 auront toujours un symbole de reste. Pour ceci, $\mathcal{M}'$ écrit $k-1$ blancs à la fin de sa bande, et revient à l'extrémité gauche en déplaçant chaque caractère du bon nombre de cases à droite en insérant les blancs aux bonnes positions. Concrètement, $\mathcal{M}'$ maintient dans son état interne un compteur i initialisé à $k-1$, puis répétitivement : revient i cases à gauche, récupère le caractère lu a, revient i cases à droite, y écrit a ; $\mathcal{M}'$ va ensuite une case à gauche ; si de plus $a = \dagger$, $\mathcal{M}'$ écrit ici un blanc B, va encore une case à gauche, fait décroître i ; et ceci jusqu'à ce que $\mathcal{M}'$ arrive au début de la bande (sur le caractère \$). Dans l'étape 3, $\mathcal{M}'$ choisit une transition dans la relation de transition, en fonction de q et des $a_1, \ldots, a_{k-1}$ collectés en étape 1, et reparcourt toute la bande de gauche à droite pour mettre à jour les w_i et les w_i'. En étape 4, $\mathcal{M}'$ revient de nouveau à gauche de la bande.

Si $\mathcal{M}$ termine en $f(n)$ étapes, elle n'utilise qu'un espace au plus $kf(n)$, et les étapes 1-4 ne prennent alors qu'un temps de l'ordre de $k'f(n)$, où k' est une constante. Le total du temps consommé par $\mathcal{M}'$ est alors $k'f(n)^2$. □

Il existe d'autre part d'innombrables variations sur la définition des machines de Turing. On peut par exemple demander, dans la version à k bandes, que la machine soit en fait déterministe, mais avec une bande supplémentaire en lecture seule dite *de choix*, et sur laquelle on ne peut se déplacer qu'à droite, et qui représente la suite des (numéros de) tous les choix qui seront faits à chaque étape du calcul (ceci suppose de les numéroter, de façon arbitraire). La machine $\mathcal{M}$ accepte alors l'entrée x si et seulement s'il existe un mot y sur la bande de choix tel que $\mathcal{M}$ accepte, au sens déterministe usuel, le couple (x, y). On en déduit notamment :

Proposition 2.3 *La classe* **NP** *est exactement la classe des langages de la forme* $\{x \in \Sigma^* \text{ de taille } n \mid \exists y \text{ de taille au plus } p(n) \cdot (x, y) \in L'\}$*, où* $L' \in \mathbf{P}$ *et* $p(n)$ *est un polynôme en* n*.*

Une autre caractérisation est par des machines qui n'ont, dans une configuration d'état interne q, et où la lettre lue est a, qu'au plus deux configurations successeur, et repérées par un booléen, 0 ou 1. S'il n'y a qu'une configuration successeur, les deux configurations successeurs sont juste identiques.

On pourrait dire que c'est tricher que d'utiliser un formalisme aussi inimplémentable, "magique", que les machines de Turing non déterministes. Le théorème de Cook-Levin énonce que, réciproquement, le problème de satisfiabilité FORM-SAT est en fait le plus compliqué de tous les problèmes de **NP**. Plus précisément, *tout* langage L que l'on peut décider avec une machine non-déterministe en temps polynomial est tel qu'on peut aussi décider $x \in L$ en construisant une formule propositionnelle $F^L(x)$ en temps polynomial, et en testant sa satisfiabilité. Ce n'est donc pas une tricherie : FORM-SAT a *exactement* le même pouvoir expressif que n'importe quelle machine de Turing non déterministe en temps polynomial.

Le théorème dit même mieux : on peut demander que $F^L(x)$ ait une forme spéciale, et soit un ensemble de *clauses*. Un *littéral* L est un atome A ou la négation $\neg A$ d'un atome. Par commodité, on notera $+A$ l'atome A vu comme littéral, $-A$ la formule $\neg A$ vue comme littéral. Une *clause* C est un ensemble fini de littéraux $L_1, L_2, \ldots, L_m$. (Cet ensemble sera représentée par une liste.) La sémantique d'une clause est celle de la disjonction de ses littéraux, et l'on notera donc, par abus de langage, cette clause $L_1 \vee L_2 \vee \ldots \vee L_m$. Lorsque $m = 0$, on obtient ainsi la clause vide, qu'il est naturel de noter $\bot$, mais qui est traditionnellement notée □.

Le problème SAT est :
ENTRÉE : une liste finie, S, de clauses ;
QUESTION : S est-elle satisfiable ?

On identifie ici une liste de clauses à une représentation d'un ensemble de clauses, que nous appellerons encore S. Rappelons que S est satisfiable si et seulement s'il existe un environnement ρ tel que $\rho \models C$ pour toute clause $C \in S$. (On peut demander que ρ soit remplacé par un environnement partiel ϱ, de domaine contenant $\mathrm{FV}(S)$, comme d'habitude.)

Lemme 2.4 $SAT \in \mathbf{NP}$.

Démonstration. Informellement, SAT est un cas particulier du problème de satisfiabilité de la proposition 2.1. Formellement, ce n'est pas tout à fait le cas. On doit d'abord vérifier que l'entrée est bien la représentation sous forme de mot d'un ensemble de clauses fini. On doit ensuite traduire cet ensemble de clauses en une formule, en insérant des signes $\wedge$ entre chaque clause. □

La démonstration détaillée du lemme 2.4 utilise un argument classique : pour montrer que l'on sait décider un langage L au moins aussi facilement qu'un langage L', il suffit de trouver une fonction facilement calculable f telle que $x \in L$ si et seulement si $f(x) \in L'$. Dans le lemme 2.4, cette fonction insère des signes $\wedge$ entre chaque clause. (Plus précisément, cette fonction vérifie d'abord que l'entrée x est bien un ensemble de clauses bien formaté ; si ce n'est pas le cas, $f(x)$ fournit par exemple une formule mal formatée, ou insatisfiable.)

Ceci porte un nom : une *réduction en temps polynomial* du langage L vers le langage L' est une fonction f des mots dans les mots, calculable en temps polynomial, et telle que pour tout mot x, $x \in L$ si et seulement si $f(x) \in L'$.

On dit que L est *réductible en temps polynomial* à L', et l'on note $L \preceq_{\mathbf{P}} L'$, si et seulement s'il existe une réduction en temps polynomial de L vers L'.

Si $L \preceq_{\mathbf{P}} L'$, L est intuitivement au moins aussi simple à décider que L' — à temps polynomial près. Il existe d'autres notions de réductibilité, notamment la notion plus fine de réductibilité en espace logarithmique, que nous aurons le temps de voir en cours de complexité avancée (MPRI, M1).

Lemme 2.5 *$\preceq_{\mathbf{P}}$ est un préordre sur l'ensemble des langages, c'est-à-dire une relation réflexive et transitive.*

Cette relation n'est pas une relation d'ordre, autrement dit $L \preceq_{\mathbf{P}} L'$ et $L' \preceq_{\mathbf{P}} L$ n'implique pas $L = L'$. En notant $\equiv_{\mathbf{P}}$ la relation d'équivalence définie par $L \equiv_{\mathbf{P}} L'$ si et seulement si $L \preceq_{\mathbf{P}} L'$ et $L' \preceq_{\mathbf{P}} L$, ceci signifie que $\equiv_{\mathbf{P}}$ n'est pas l'égalité. Par exemple, tous les langages de la classe $\mathbf{P}$ des langages décidables en temps polynomial (déterministe) sont équivalents pour $\equiv_{\mathbf{P}}$.

Lemme 2.6 *Les classes $\mathbf{P}$, $\mathbf{NP}$ sont stables par réductibilité en temps polynomial : si $L' \in \mathbf{P}$ (resp., $L' \in \mathbf{NP}$), et $L \preceq_{\mathbf{P}} L'$, alors $L \in \mathbf{P}$ (resp., $L \in \mathbf{NP}$).*

2.2 Le théorème de Cook-Levin

La proposition clé, due à Cook et indépendamment à Levin, est la suivante :

Proposition 2.7 *SAT est* $\mathbf{NP}$-difficile *: pour tout $L \in \mathbf{NP}$, $L \preceq_{\mathbf{P}}$ SAT.*

Démonstration. L'idée est la suivante. On doit trouver une réduction f en temps polynomial, telle que $f(x)$ soit un ensemble fini de clauses, et que $f(x)$ soit satisfiable si et seulement si $x \in L$. Pour ceci, on exploite le fait que, comme $L \in \mathbf{NP}$, il existe une machine non déterministe $\mathcal{M}$, en temps majoré par un polynôme $p(n)$ en la taille n de x, qui accepte x si et seulement si $x \in L$. (Par la proposition 2.2, il suffit de considérer une machine ordinaire

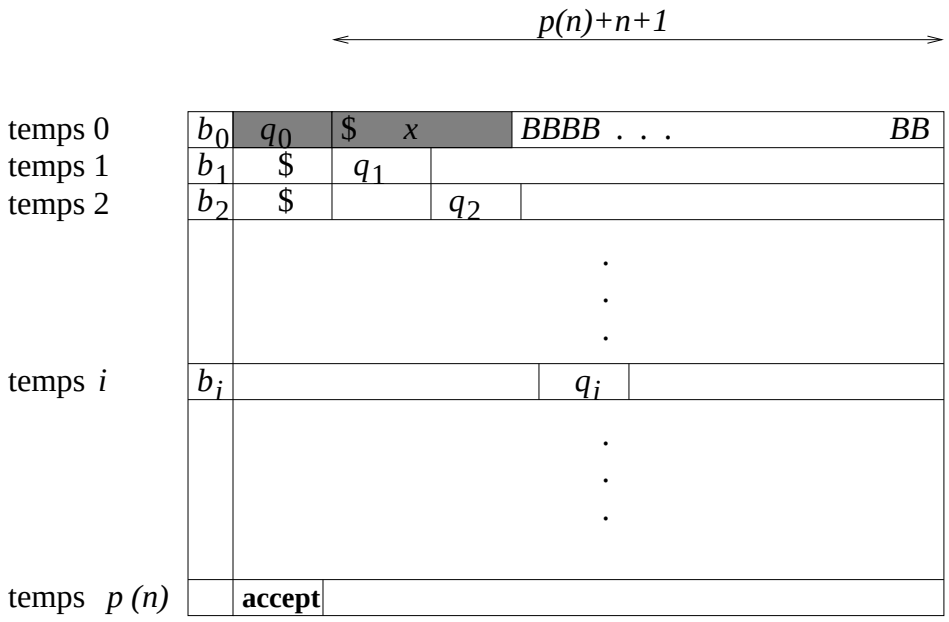


FIG. 6 – La construction du théorème de Cook-Levin

à une bande.) $\mathcal{M}$ ne peut pas consommer plus qu'un espace $p(n) + n + 1$, et l'on peut donc supposer que toutes les configurations de $\mathcal{M}$ sont de taille exactement $p(n) + n + 1$, en ajoutant des blancs B au besoin à droite de la bande. Ceci demande aussi à ce que la configuration initiale soit $(\epsilon, q_0, \$xB^{p(n)})$. Nous coderons les configurations (w, q, aw') sous forme du mot concaténé $wqaw'$, où Q est considéré comme un alphabet disjoint de l'alphabet de la bande. On peut aussi supposer sans perdre en généralité que **accept** et **reject** sont en fait des états internes de Q, et que chaque configuration n'a qu'au plus deux configurations successeurs, et repérées par un booléen, 0 ou 1, que l'on appellera la *devinette*. On peut finalement supposer que la machine ne s'arrête pas lorsqu'elle atteint un de ces deux états, mais boucle indéfiniment. On a alors $x \in L$ si et seulement si l'on peut dessiner un tableau comme à la figure 6, obéissant aux contraintes suivantes : (a) les devinettes b_0, b_1, ..., $b_{p(n)}$ dans la colonne occupent la colonne de gauche, le reste du tableau est un empilement de mots de longueur $p(n) + n + 2$ (la taille $p(n) + n + 1$, plus un caractère pour coder l'état interne), sur l'alphabet $Q \uplus \Sigma$; (b) on trouve la lettre q_0 dans la case en haut à gauche, (c) le mot $\$xB^{p(n)}$ juste à sa droite, (d) la lettre **accept** en bas à gauche ; et (e) pour tout $i \geq 1$, la ligne i est reliée à la ligne $i - 1$ au moyen de la relation de transition. Il ne reste plus qu'à coder tout ce tableau en binaire, à réserver une variable propositionnelle par bit du tableau (il y en aura au plus $O((p(n) + n)^2)$), et à écrire les contraintes (a)–(e) sous forme de formules logiques, que l'on convertira en ensembles de clauses.

Dans la suite, on notera $\vec{x}$ un vecteur de variables propositionnelles, distinctes deux à deux, d'une longueur qui sera déterminée par le contexte. On notera x_j la variable numéro j du vecteur $\vec{x}$. On code les lettres $q \in Q$ ou $a \in \Sigma$ sur m bits (une constante), ce qui revient à dire que l'on fabrique $p(n) + 1$ vecteurs $\vec{z}_i$, $0 \leq i \leq p(n)$, de $m(p(n) + n + 2)$ variables propositionnelles chacun, pour représenter les configurations au temps i, $0 \leq i \leq p(n)$.

Fabriquons aussi $p(n)+1$ variables propositionnelles b_i, $0 \leq i \leq n$. Écrivons maintenant les contraintes :

(a) il n'y a rien à écrire ;

(b) pour simplifier, supposons que l'écriture en binaire de q_0 soit 0^m ; on écrit alors les clauses $-z_{00}$, $-z_{01}$, …, $-z_{0(m-1)}$ exprimant que les bits 0 à $m-1$ de la configuration au temps 0 sont nuls.

(c) pour chaque position j, $0 \leq j \leq p(n)$, dans le mot $\$xB^{p(n)}$, pour chaque indice k de bit ($0 \leq k \leq m-1$), on écrit la clause $-z_{0(mj+k)}$ si le bit k de la lettre numéro j de $\$xB^{p(n)}$ vaut 0, $+z_{0(mj+k)}$ sinon.

(d) pour simplifier, supposons que **accept** s'écrive 1^m en binaire ; on écrit alors les clauses $+z_{p(n)0}$, $+z_{p(n)1}$, …, $+z_{p(n)(m-1)}$.

(e) Nous en arrivons au codage de la relation de transition, la partie la plus intéressante. Nous aurons besoin de quelques abréviations, pour nous simplifier la vie. Nous écrirons $\vec{z}[j]$ le sous-vecteur des m variables propositionnelles représentant la lettre numéro j dans la bande représentée par le vecteur $\vec{z}$, c'est-à-dire le sous-vecteur $z_{jm}, z_{jm+1}, \ldots, z_{jm+m-1}$. Pour tout vecteur de m bits (constants) $\vec{a}$, on écrira $\vec{z}[j] \neq \vec{a}$ la clause $\pm_0 z_{jm} \vee \pm_1 z_{jm+1} \vee \ldots \vee \pm_{m-1} z_{jm+m-1}$, où le signe $\pm_k$ est $-$ si $a_k = 1$, $+$ si $a_k = 0$. (Cette clause est fausse si et seulement si le vecteur des valeurs de $\vec{z}[j]$ égale le vecteur $\vec{a}$.) Par souci de clarté, plutôt que d'écrire des clauses de la forme $\vec{z}[j] \neq \vec{a} \vee \vec{z}[j'] \neq \vec{a}' \vee C$ (où C est une clause), on écrira $\vec{z}[j] = \vec{a} \wedge \vec{z}[j'] = \vec{a}' \Rightarrow C$. On identifiera les lettres de $Q \uplus \Sigma$ avec les vecteurs des bits de leurs représentations binaires. On utilisera des conventions similaires pour les bits de devinettes, et on écrira ainsi $b_i = b \Rightarrow \ldots$ pour $b_i \neq b \vee \ldots$, et ainsi de suite.

Pour chaque numéro de ligne i, $1 \leq i \leq p(n)$, pour chaque position j dans la configuration ($0 \leq j \leq p(n)+n+1$), on va commencer par écrire ce qui se passe si la lettre à la position j de la ligne $i-1$ est un état interne. Pour ceci, on énumère les états internes, ainsi que la valeur de la devinette b_{i-1}. Pour chaque $q \in Q$, pour chaque booléen $b \in \{0,1\}$, pour chaque lettre $a \in \Sigma$, notons $(q', a', dir) \in \delta(q, a)$ la transition numéro b, et écrivons :

1. si $dir = \leftarrow$, alors soit $j = 0$, cas qui ne correspond à aucune exécution possible de la machine de Turing, et on n'écrit alors aucune clause, soit $j \geq 1$, et on écrit des clauses exprimant que toutes les lettres aux positions $0, \ldots, j-2$ et $j+1, \ldots, p(n)+n+1$ restent inchangées, que les lettres aux positions $j-1$, j et $j+1$ de la ligne i sont respectivement q', la lettre à la position $j-1$ de la ligne $i-1$, et a' respectivement :
   - pour chaque j' entre 0 et $j-2$ ou entre $j+1$ et $p(n)+n+1$, pour chaque k, $0 \leq k \leq m_1$, on souhaiterait écrire $\vec{z}_{i-1}[j] = q \wedge b_{i-1} = b \Rightarrow \vec{z}_i[j'] = \vec{z}_{i-1}[j']$, mais ceci n'est pas une clause. En revanche, on peut écrire les clauses suivantes, pour tout k, $0 \leq k \leq m-1$:

$$\begin{array}{l} \vec{z}_{i-1}[j] = q \wedge b_{i-1} = b \Rightarrow -z_{(i-1)(j'm+k)} \vee +z_{i(j'm+k)} \\ \vec{z}_{i-1}[j] = q \wedge b_{i-1} = b \Rightarrow +z_{(i-1)(j'm+k)} \vee -z_{i(j'm+k)} \end{array} \quad (3)$$

– À la position $j-1$, on doit trouver q', on serait donc tenté d'écrire :

$$\vec{z}_{i-1}[j] = q \wedge b_{i-1} = b \Rightarrow z_i[j-1] = q' \tag{4}$$

Techniquement, ce n'est pas une clause, car $z_i[j-1] = q'$ est, intuitivement, une conjonction. Décidons que ceci est une abréviation commode pour les m clauses $\vec{z}_{i-1}[j] = q \wedge b_{i-1} = b \Rightarrow z_{i((j-1)m+k)} = q'_k$, $0 \leq k \leq m-1$, où $z_{i((j-1)m+k)} = q'_k$ dénote $+z_{i((j-1)m+k)}$ si $q'_k = 1$, $-z_{i((j-1)m+k)}$ si $q'_k = 0$.

– À la position j, on doit écrire la lettre de la position $j-1$ de la ligne $i-1$, donc on écrira, de façon similaire à (3) :

$$\begin{array}{l} \vec{z}_{i-1}[j] = q \wedge b_{i-1} = b \Rightarrow -z_{(i-1)((j-1)m+k)} \vee +z_{i(jm+k)} \\ \vec{z}_{i-1}[j] = q \wedge b_{i-1} = b \Rightarrow +z_{(i-1)((j-1)m+k)} \vee -z_{i(jm+k)} \end{array} \tag{5}$$

– À la position $j+1$, on doit trouver la lettre a', et on écrit les m clauses suivantes, qui obéissent à la même convention que (4) :

$$\vec{z}_{i-1}[j] = q \wedge b_{i-1} = b \Rightarrow z_i[j+1] = a' \tag{6}$$

2. Lorsque $dir = \downarrow$ ou $dir = \rightarrow$, on utilise le même genre de codage. Notons que lorsque $dir = \rightarrow$, on n'a pas à prévoir le cas où il faudra insérer un blanc B en fin de bande, car les bandes ont été prévues suffisamment larges.

On note que l'on peut produire toutes ces clauses en effectuant des boucles imbriquées sur i, j, q, a, b, j', k, et ce pour $O(p(n)(p(n)+n)^2)$ tours. Chaque clause fabriquée est de longueur constante, et est produite en temps polynomial elle-même.

Si la machine accepte, on déduit un environnement partiel qui satisfait toutes les clauses ci-dessus, en affectant à b_i le numéro du choix effectué à l'étape i, et à chaque autre variable propositionnelle le bit correspondant du tableau de la figure 6. Réciproquement, si ρ satisfait toutes ces clauses, les clauses (e) garantissent que les bits correspondant décrivent une exécution de la machine de Turing, (b) et (c) que la configuration de départ soit la configuration initiale pour l'entrée x, et (d) que la machine accepte en au plus $p(n)$ étapes. Comme la machine est en temps $p(n)$, il est équivalent de demander qu'elle accepte ou qu'elle accepte en au plus $p(n)$ étapes.

La fonction qui à x associe l'ensemble des clauses ci-dessus est donc la réduction en temps polynomial cherchée. □

On en déduit [2] :

Théorème 2.8 (Cook-Levin) *SAT est* **NP**-complet*, autrement dit c'est le plus compliqué de tous les problèmes de* **NP***, à* $\preceq_{\mathbf{P}}$ *près. De façon équivalente, SAT est à la fois dans* **NP** *et* **NP***-difficile.*

Il sera important de se rappeler qu'un problème **NP**-complet n'est pas juste **NP**-difficile, mais (ce qu'on a trop facilement tendance à oublier) dans **NP**.

▷ **Exercice 2.1**

Montrer qu'il existe un langage qui est trivialement **NP**-complet : le langage LNA (Linear Non-deterministic machine Acceptance) des triplets $(< \mathcal{M} >, x, 1^n)$, où $\mathcal{M}$ est une machine de Turing non-déterministe à une bande, et $\mathcal{M}$ accepte x en au plus n étapes. (La notation 1^n est une autre façon d'exprimer que n est écrit en unaire.)

Ceci ne nous dit pas si SAT est réellement difficile à résoudre dans l'absolu. Mais ceci implique un résultat assez fort. Rappelons que **P** est la classe des langages décidables en temps polynomial *déterministe*, c'est-à-dire sur une machine de Turing ordinaire, déterministe. (La définition ne dépend pas du nombre de bandes de la machine.) On note que, clairement, $\mathbf{P} \subseteq \mathbf{NP}$.

Proposition 2.9 *Les deux questions suivantes sont équivalentes :*
- *SAT est décidable en temps polynomial ;*
- $\mathbf{P} = \mathbf{NP}$.

Démonstration. Si SAT est décidable en temps polynomial, c'est-à-dire si SAT $\in \mathbf{P}$, alors tout langage de **NP** est réductible en temps polynomial à un problème de **P** (à savoir SAT), donc est lui-même dans **P** par le lemme 2.6. Comme d'autre part il est clair que $\mathbf{P} \subseteq \mathbf{NP}$, on a $\mathbf{P} = \mathbf{NP}$. La réciproque, que si $\mathbf{P} \subseteq \mathbf{NP}$ alors SAT $\in \mathbf{P}$, est par le lemme 2.4. □

On connaît plusieurs centaines de problèmes **NP**-complets, et plus probablement plusieurs milliers. (Voir le livre [6], qui en contient un catalogue, datant de 1979.) Le résultat $\mathbf{P} = \mathbf{NP}$, ou le fait que l'un quelconque de ces problèmes soit dans **P**, impliquerait que tous les autres seraient résolubles en temps polynomial. On pense généralement que $\mathbf{P} \neq \mathbf{NP}$, mais c'est un problème qui a défié toutes les tentatives depuis maintenant presque 40 ans.

Un autre problème **NP**-complet est 3-SAT. On notera qu'une fois connu un problème **NP**-complet, il suffit de le réduire à un autre, L, pour montrer que L est **NP**-difficile. C'est pratiquement toujours ainsi que nous établirons la **NP**-complétude de langages.

Proposition 2.10 (3-SAT) *On appelle* 3-clause *une clause contenant au plus* 3 *littéraux. Le problème 3-SAT suivant est* **NP***-complet :*
ENTRÉE : une liste finie, S, de 3*-clauses ;*
QUESTION : S est-elle satisfiable ?

Démonstration. 3-SAT étant un cas particulier de SAT (et la vérification du format étant en temps polynomial), 3-SAT est dans **NP**. Réciproquement, on réduit toute instance S de SAT à 3-SAT comme suit. Pour chaque clause $C \in S$ ayant au moins 4 littéraux, écrivons C sous la forme $L_1 \vee L_2 \vee C'$, où C' est le reste de la clause. On crée une variable propositionnelle fraîche, on produit la 3-clause $L_1 \vee L_2 \vee +A$ et on continue le processus sur la clause $-A \vee C'$ tant qu'elle est de longueur au moins 4. Plus synthétiquement, on convertit $L_1 \vee L_2 \vee \ldots \vee L_n$ $(n \geq 4)$ en les clauses $L_1 \vee L_2 \vee +A_1$, $-A_1 \vee L_3 \vee +A_2$, $-A_2 \vee L_4 \vee +A_3$, $\ldots$, $-A_{n-2} \vee L_n \vee +A_{n-1}$: ceci permet de voir que la transformation s'effectue en temps polynomial.

Si $\varrho \models C$, où $C = L_1 \vee L_2 \vee C'$, on peut étendre ϱ de sorte à attribuer à la variable fraîche A la valeur de C' dans ϱ. Ceci rend automatiquement $-A \vee C'$ vraie. Si A est vrai, la clause

$L_1 \vee L_2 \vee +A$ aussi ; sinon, A est faux, donc tous les littéraux de C' sont faux dans ϱ ; mais C étant satisfaite par ϱ, l'un des littéraux L_1 ou L_2 est vrai, donc la clause $L_1 \vee L_2 \vee +A$ est encore vraie. Ceci montre que si S est satisfiable, l'ensemble de 3-clauses S' obtenu à partir de S l'est aussi.

Réciproquement, si S' est satisfait par un environnement partiel ϱ, ϱ satisfait aussi S. Il suffit de réaliser que si $\varrho \models L_1 \vee L_2 \vee +A$ et $\varrho \models -A \vee C'$, alors $\varrho \models L_1 \vee L_2 \vee C'$. En effet, si $\varrho(A)$ est faux, la première hypothèse implique que L_1 ou L_2 est vrai dans ϱ ; sinon, la seconde hypothèse implique qu'un des littéraux de C' est vrai. □

Nous verrons quelques autres problèmes importants qui sont **NP**-complets aussi à la section 4. Il est remarquable que la plupart des problèmes dans **NP** que l'on connaisse soient dans **P** ou **NP**-complets. Il n'existe que de rares exceptions, comme le problème de l'isomorphisme de graphes, ou bien la question DDH (Decisional Diffie-Hellman, une question importante en cryptographie ; la question est, étant donné un générateur g de $\mathbb{Z}/p\mathbb{Z}$, où p est premier, et trois nombres g^a, g^b et g^c modulo p, c est-il égal à ab modulo $p-1$), qui sont dans **NP** mais dont on ne sait pas s'ils sont dans **P**, ni s'ils sont **NP**-complets. (Il y a intérêt à ce que DDH ne soit pas dans **P**, sinon un certain nombre de constructions cryptographiques ne seront plus sûres. On a d'autre part un certain nombre d'indices laissant à penser que l'isomorphisme de graphes n'est pas **NP**-complet.)

▷ **Exercice 2.2**

Reconsidérons le problème FORM-SAT de la satisfiabilité de formules propositionnelles générales F. Montrer que FORM-SAT est **NP**-complet.

▷ **Exercice 2.3**

Le problème 3-SAT-3-OCC est le suivant :
ENTRÉE : une liste finie, S, de 3-clauses, où chaque variable propositionnelle apparaît au plus 3 fois ;
QUESTION : S est-elle satisfiable ?

Montrer que 3-SAT-3-OCC est **NP**-complet.

2.3 Degrés intermédiaires : le théorème de Ladner

Pour ce qui est des variantes du problème de la satisfiabilité SAT, le *théorème de Schaefer* [10] énonce que toutes les variantes définies syntaxiquement (par un procédé naturel) sont soit dans **P** soit **NP**-complètes. De plus, on connaît les variantes de **P**, qui sont en nombre fini.

En général, cependant, il y a nécessairement des problèmes de **NP** qui ne sont ni dans **P** ni **NP**-complets. C'est le *théorème de Ladner* [9]. Notons $\prec_{\mathbf{P}}$ la partie stricte du préordre $\preceq_{\mathbf{P}}$, c'est-à-dire $L \prec_{\mathbf{P}} L'$ si et seulement si $L \preceq_{\mathbf{P}} L'$ et $L' \not\preceq_{\mathbf{P}} L$.

Proposition 2.11 (Ladner) *Pour tout langage récursif $L \notin \mathbf{P}$, il existe un langage L' tel que $L' \prec_{\mathbf{P}} L$ et $L' \notin \mathbf{P}$. En particulier, Si $\mathbf{P} \neq \mathbf{NP}$, il existe des langages qui ne sont ni dans **P** ni **NP**-complets. Il en existe même une infinité non équivalents pour $\equiv_{\mathbf{P}}$.*

Démonstration. La démonstration est par une diagonalisation relativement étrange. La démonstration de Ladner est complexe, et nous en donnons une fondée sur une idée non publiée d'Impagliazzo [1]. L'idée est d'utiliser la technique du *bourrage* ("padding"), c'est-à-dire de considérer le langage $L' = \{x\#1^{f(n)-n} \mid x \in L, \text{ où } n = |x|\}$, où # est un nouveau symbole, pour une certaine fonction f calculable en temps polynomial de n (l'entrée n étant en unaire, et la sortie étant en binaire) et telle que $f(n) \geq n$ pour tout n. On note $|x|$ la taille de x. Le fait que l'on répète 1 un nombre de fois égal à $f(n) - n$ a pour effet que $x\#1^{f(n)-n}$ est de longueur $f(n) + 1$. Le principal intérêt de la technique de bourrage est qu'elle fait de L' un langage plus simple à décider que L. Par exemple, si $f(n)$ est de l'ordre de 2^n, et L est décidable en temps 2^n, alors L' est décidable en temps polynomial : sur l'entrée y, on vérifie que y ne contient qu'un symbole # ; ceci fournit la longueur n de x en unaire, ce qui nous permet de calculer $f(n)$; on vérifie alors que la longueur de y est exactement $f(n)+1$, et que tous les symboles à droite de # sont des 1 ; finalement, on décide si x, la partie de y qui est à gauche de #, est dans L en temps $2^n = |y|$... c'est-à-dire en temps linéaire en la taille de l'entrée y. Un point délicat est la vérification que la longueur de y est exactement $f(n) + 1$. Une solution serait d'écrire $f(n) + 1$ blancs sur une bande et de comparer la longueur de y et celle de cette bande ; mais cette bande peut être alors de longueur non polynomiale en $|y|$. À la place, on maintient un compteur en binaire sur une bande β, initialisé à 0, et on voyage de gauche à droite sur la bande y, en incrémentant le compteur à chaque caractère. Si l'on arrive à la fin de y et que le compteur a atteint $f(n) + 1$, la longueur est la bonne ; sinon, et si on arrive à la fin de y ou que le compteur atteigne $f(n) + 1$ sans que l'autre condition soit satisfaite, la longueur n'est pas la bonne. (Il est facile d'incrémenter un nombre en binaire : tant que l'on voit des 1, les mettre à 0 ; si l'on voit un 0, le mettre à 1 et s'arrêter ; si l'on arrive en fin de bande, ajouter un 1 à droite.)

Le même argument montre, dans le cas général, que $L' \preceq_{\mathbf{P}} L$. Nous allons construire f suffisamment grande, de sorte que $L' \not\equiv_{\mathbf{P}} L$, c'est-à-dire que L' soit réellement strictement plus simple à décider que L ; mais pas trop grande, pour être sûr que $L' \notin \mathbf{P}$; et nous devons nous assurer que f sera calculable en temps polynomial.

Commençons par remarquer que l'on peut énumérer en temps polynomial toutes les machines de Turing (à une bande), c'est-à-dire calculer en temps polynomial une fonction qui prend un entier en binaire i, et retourne le code $< \mathcal{M}_i^0 >$ d'une machine de Turing $\mathcal{M}_i^0$, telle que toutes les machines de Turing se retrouvent ainsi énumérées. En effet, par exemple, on peut coder une machine de Turing $\mathcal{M}$ en numérotant les états internes et les lettres de l'alphabet des bandes, et en décrivant juste la fonction de transition, sous forme d'une table. Cette table peut être décrite comme un mot, avec des séparateurs adéquats entre les entrées. Ce mot, ensuite, peut être écrit en binaire, notons-le $\ulcorner\mathcal{M}\urcorner$ et préfixé par un 1 pour former un nombre en binaire unique. On peut définir la fonction qui à i associe $< \mathcal{M} >$ si i est le nombre qui, écrit en binaire, est le mot $1\ulcorner\mathcal{M}\urcorner$, et sinon associe à i le code d'une machine donnée, par exemple qui accepte toujours sans faire de calcul.

On ne peut pas énumérer de même toutes les machines en temps polynomial, mais on peut contourner le problème comme suit. Pour tout polynôme p à coefficients entiers positifs, à partir de toute machine $\mathcal{M}$, on peut construire une machine $\mathcal{M}/p$ qui maintient un compteur

sur une bande auxiliaire, l'initialise à $p(n)$, où n est la taille de l'entrée, puis simule $\mathcal{M}$ et décrémente le compteur à chaque étape de $\mathcal{M}$; lorsque le compteur passe à 0, $\mathcal{M}/p$ rejette. On peut alors, sur le même principe que plus haut, définir une fonction $i \mapsto < \mathcal{M}_i/p_i >$ en temps polynomial qui énumère les codes de toutes les machines $\mathcal{M}/p$ lorsque $\mathcal{M}$ parcourt les machines de Turing et p les polynômes de la forme $p(n) = n^j$. On a ici besoin de décoder i comme un couple formé d'un indice entier dénotant le numéro de la machine $\mathcal{M}_i$, et de l'entier j, de sorte que l'opération de décodage se fasse en temps polynomial. La fonction β de Gödel n'est pas tout à fait adéquate. En revanche, la fonction couple $\langle \_, \_ \rangle : \mathbb{N}^2 \to \mathbb{N}$ définie de sorte que $\langle m, n \rangle$ soit obtenu en intercalant les bits de m et de n écrits en binaire (formellement, si $m = \sum_i m_i 2^i$, $n = \sum_i n_i 2^i$, alors $\langle m, n \rangle = \sum_i m_i 4^i + 2 \sum_i n_i 4^i$), convient.

On définit la fonction $f(n)$, en même temps que le langage $L' = \{x\#1^{f(n)-n} \mid x \in L, \text{ où } n = |x|\}$, comme suit. Pour rendre les choses plus claires, on notera L'_n le langage des mots de longueur $f(n)$ de L', c'est-à-dire $L'_n = \{x\#1^{f(n)-n} \mid x \in L, |x| = n\}$. Rappelons que f prend un entier n en unaire, et doit retourner un résultat en binaire. Nous définissons $f(n)$ et L'_n par récurrence sur n. Autrement dit, sur l'entrée n, nous calculons $f(0)$, $f(1)$, …, $f(n)$ successivement en rangeant à chaque fois la valeur $f(i)$, $0 \leq i \leq n$, à l'indice i d'une table T. En supposant $f(0)$, $f(1)$, …, $f(n-1)$ calculés et rangés en $T[0]$, $T[1]$, …, $T[n-1]$, on calcule $f(n)$ par :

1. Initialiser une variable i à 1.
2. Convertir n d'unaire en binaire sur une bande auxiliaire β. (Incrémenter un compteur en binaire, tout en voyageant de gauche à droite sur la bande de n.)
3. Pour tout j (écrit en binaire) de 1 à n, considérer j comme le mot $1x$ (ceci revient à énumérer tous les mots x sur $\{0,1\}^*$ de taille de l'ordre de $\log n$), puis :
   (a) Tester si $\mathcal{M}_i/p_i$ accepte x mais $x \notin L'$, ou $\mathcal{M}_i/p_i$ rejette x mais $x \in L'$. Le point délicat est le test d'appartenance à L', puisque L' n'est pas encore défini… mais L'_m l'est, pour tout $m < n$. On procède donc comme suit. Comme dans l'argument qui établit $L' \preceq_{\mathbf{P}} L$ donné plus haut, on vérifie que x est (le codage binaire d'un mot qui contient) un unique # suivi de caractères 1, et tel que le préfixe précédant # est dans L ; enfin, on vérifie que, si m est la longueur de ce préfixe, alors la longueur de x est exactement $T[m] + 1$ $(= f(m) + 1)$.
   (b) Si le test 3a réussit (on a trouvé un mot de taille logarithmique qui distingue L' du langage de $\mathcal{M}_i/p_i$), incrémenter i et écrire n^i sur la bande β (autrement dit, additionner n fois le contenu de β ; on laisse la définition de l'addition binaire en exercice) ; sinon laisser i et β tels quels.
4. Retourner n^i, qui est écrit sur la bande β.

On posera dans la suite $i(n)$ la valeur finale de i, de sorte que $f(n) = n^{i(n)}$.

On vérifie d'abord que $f(n)$ **se calcule en temps polynomial**. La phase d'itération sur j se fait en maintenant j écrit en binaire sur une bande, initialisé à 1, et en voyageant de gauche à droite sur la bande représentant n (en unaire) ; chaque fois qu'on va à droite sur cette bande, on incrémente j. Ceci étant précisé, la boucle sur j ne fait que n tours, et comme n est en unaire, la taille de n est n lui-même. Comme i vaut au plus n et que

l'énumération $i \mapsto < \mathcal{M}_i/p_i >$ est en temps polynomial en (la taille de) i, on vérifie aisément que f est calculable en temps polynomial. Il est important que $f(n)$, c'est-à-dire le contenu de la bande β, soit écrit en binaire : si β était écrit en unaire, sa taille égalerait sa valeur, qui peut aller jusqu'à n^i, i pouvant aller jusqu'à n. Notons aussi qu'il est important que n soit écrit en unaire, ce qui permet à la liste $T[0]$, $T[1]$, ..., $T[n-1]$, de ne prendre qu'une taille polynomiale en la taille de n, c'est-à-dire n lui-même.

Montrons que L' **n'est pas dans P.** Si L' était dans **P**, L' serait décidable en temps $p(n)$ pour un certain polynôme p en la taille n de l'entrée. Pour j égal au degré de p plus 1, on peut donc décider L' en temps n^j pour n assez grand, disons $n \geq n_0$, par une machine de Turing $\mathcal{M}_0$. On peut alors décider L' en temps n^i pour tout n, en modifiant la machine de Turing $\mathcal{M}_0$, comme suit. On construit d'abord une table de toutes les entrées x de tailles inférieures à n_0, associées à un booléen, vrai si $x \in L'$, faux sinon. On construit ensuite une machine de Turing qui parcourt son entrée x de gauche à droite ; si sa longueur est strictement inférieure à n_0, la machine lit l'entrée x de la table est répond ensuite en temps constant ; sinon, la machine calcule comme $\mathcal{M}_0$. La nouvelle machine calcule alors en temps n^i pour tout n.

Puisque l'on peut décider L' en temps n^i, il existe donc un i_0 tel que $\mathcal{M}_{i_0}/p_{i_0}$ décide L'. La valeur de i calculée dans l'algorithme de f ne peut donc pas dépasser i_0 : si i atteint i_0, le test de l'étape 3a échouera toujours, et i ne sera donc pas incrémenté à l'étape 3b. Donc $f(n) \leq n^{i_0}$ pour tout n. Mais alors $L \preceq_{\mathbf{P}} L'$: on réduit L à L' en concaténant à l'entrée x un $\#$, puis $f(n)-n$ caractères 1 — comme $f(n) \leq n^{i_0}$, on peut effectivement écrire les $f(n)-n$ caractères 1 en temps polynomial. Puisque $L \preceq_{\mathbf{P}} L'$ et $L' \in \mathbf{P}$ par hypothèse, L serait aussi dans **P** par le lemme 2.6, contradiction.

Rappelons que $f(n)$ s'écrit $n^{i(n)}$, où $i(n)$ est la valeur de i à la fin de l'algorithme f. Or, pour tout $i \in \mathbb{N}$, il n'existe qu'un nombre fini d'entiers n tels que $i(n) = i$. En effet, comme $L' \notin \mathbf{P}$, pour chaque entier i, $\mathcal{M}_i/p_i$ ne peut pas décider L', donc il existe une entrée x telle que $\mathcal{M}_i/p_i$ accepte sur l'entrée x mais $x \notin L'$, ou $\mathcal{M}_i/p_i$ rejette sur x mais $x \in L'$. En conséquence l'étape 3a doit échouer dès que n est supérieur strictement à $2^{|x|}$, donc $i(n) \neq i$. Ceci montre que $i(n)$ **tend vers** $+\infty$ lorsque n tend vers $+\infty$, quoique très lentement — au mieux comme $\log \log n$.

Rappelons que $L' \preceq_{\mathbf{P}} L$. Montrons que $L \not\preceq_{\mathbf{P}} L'$. Sinon, il existerait une réduction g en temps polynomial de L vers L'. L'idée est d'itérer cette réduction, qui réduit fortement la taille de l'entrée, jusqu'à ce que la taille tombe en-dessous d'une constante n_0, et nous décidons ensuite toutes ces petites instances en tabulant les réponses.

Il existe un entier j tel que, pour tout n assez grand, disons $n \geq n_0$, g termine en temps majoré par n^j sur les entrées x de taille n. On va supposer que la constante n_0 est suffisamment grande, de plus, de sorte que pour tout $n' \geq n_0$, $i(n') \geq j+1$. C'est possible, car $i(n')$ tend vers $+\infty$ lorsque n' tend vers $+\infty$. De plus, $x \in L$ si et seulement si $g(x)$ est de la forme $x'\#1^{f(n')-n'}$, où $n' = |x'|$ et $x' \in L$. Lorsque $n \geq n_0$, donc, $f(n')+1 \leq n^j$, ce qui implique $n'^{i(n')} \leq n^j$, donc $i(n') \log n' \leq j \log n$. Comme $i(n') \geq j+1$, $\log n' \leq j/(j+1)\ \log n$.

On peut donc décider si $x \in L'$ en itérant cette réduction au plus k fois, dès que la taille de x est telle que $[j/(j+1)]^k \log |x| \leq \log n_0$. Autrement dit, on n'a à itérer cette

réduction qu'au plus $\log(\log n_0/\log|x|)/\log(j/(j+1))$, ce qui est (très nettement) majoré par un polynôme en $|x|$ pour $|x| \geq n_0$. On décide toutes les instances de taille inférieure à n_0 en regardant la bonne réponse dans une table. Ceci est possible, car il n'y a qu'un nombre fini, fixé, de mots de longueur inférieure à n_0. Cet algorithme de réduction répétée déciderait alors L en temps polynomial, contradiction.

Ceci démontre la première partie du théorème. Pour la seconde partie, si $\mathbf{P} \neq \mathbf{NP}$, par la proposition 2.9, SAT$\notin \mathbf{P}$, et l'on applique la première partie du théorème à $L = \text{SAT}$. Pour la troisième partie, on construit une suite infinie décroissante de langages L_i, $i \in \mathbb{N}$, par récurrence sur i, où $L_0 = \text{SAT}$, aucun L_i n'est dans $\mathbf{P}$, et L_{i+1} est obtenu à partir de L_i en utilisant la première partie du théorème. □

3 Algorithmes de démonstration automatique

Nous avons déjà vu deux algorithmes de démonstration automatique en logique propositionnelle. Le test par force brute de tous les environnements partiels ϱ, et la fonction `prove` de recherche de preuve par la méthode des tableaux. Il en existe d'autres, et nous allons en voir trois : la méthode de Davis-Putnam-Logemann-Loveland (DPLL, datant de deux articles, un de 1960 et un de 1965), la résolution (Robinson, 1965), et les diagrammes de décision binaires (BDD ; inventés par Akers en 1976, c'est réellement Bryant en 1986 qui a popularisé l'outil). Ceci nous en fera cinq au total… on a le choix ! Et encore, il en existe d'autres, comme la méthode de Stålmarck (1989), la méthode par plans de coupures de Cook et Reckhow ("cutting planes") par exemple. En pratique, les versions optimisées de DPLL sont celles qui sont les plus efficaces pour décider SAT, notamment sur des problèmes durs servant lors de concours. On considère que DPLL peut traiter des instances de SAT difficiles ayant plusieurs milliers de variables.

3.1 Formes clausales

La méthode DPLL et la résolution résolvent le problème SAT, c'est-à-dire prennent non pas une formule générale F en entrée, mais un ensemble de clauses S. On a vu que SAT était **NP**-complet, et FORM-SAT aussi (exercice 2.2). On peut donc en principe traduire en temps polynomial toute formule F en un ensemble de clauses S tel que S est satisfiable si et seulement si F est satisfiable : c'est juste l'énoncé FORM-SAT $\preceq_{\mathbf{P}}$ SAT, qui est dû au fait que FORM-SAT est dans **NP** et SAT est **NP**-complet. Mais la traduction obtenue par le théorème de Cook-Levin est relativement inefficace, et produit un ensemble de clauses S qui décrit l'exécution d'une machine de Turing plutôt que la sémantique de F, et dont il est en général difficile de tester la satisfiabilité.

Une traduction plus directe est obtenue en appliquant les règles de transformation suivantes, qui ont pour effet de pousser les négations tout en bas des formules, et de distribuer les disjonctions sur les conjonctions. Lorsque ceci termine, on a obtenu une formule équivalente, qui est un "et" de "ou" de littéraux (le "et" de zéro formule étant $\top$, le "ou" de zéro formule

étant $\bot$), c'est-à-dire une conjonction finie de clauses.

$$\begin{array}{c} \neg\neg F \to F \quad \neg\top \to \bot \quad \neg\bot \to \top \\ \neg(F_1 \vee F_2) \to \neg F_1 \wedge \neg F_2 \quad \neg(F_1 \wedge F_2) \to \neg F_1 \vee \neg F_2 \\ (F_1 \Rightarrow F_2) \to (\neg F_1 \vee F_2) \\ F_1 \vee \top \to \top \quad \top \vee F_1 \to \top \quad F_1 \vee \bot \to F_1 \quad \bot \vee F_1 \to F_1 \\ F_1 \wedge \top \to F_1 \quad \top \wedge F_1 \to F_1 \quad F_1 \wedge \bot \to \bot \quad \bot \wedge F_1 \to \bot \\ (F_1 \wedge F_2) \vee F_3 \to (F_1 \vee F_3) \wedge (F_2 \vee F_3) \quad F_3 \vee (F_1 \wedge F_2) \to (F_3 \vee F_1) \wedge (F_3 \vee F_2) \end{array} \tag{7}$$

On applique ces règles en remplaçant toute sous-formule d'une formule donnée qui est un côté gauche de règle par le côté droit correspondant, jusqu'à terminaison. (L'ensemble des *sous-formules* d'une formule est le plus petit ensemble qui contient la formule elle-même et toutes les sous-formules de ses sous-formules immédiates. Les seules *sous-formules immédiates* de $F_1 \wedge F_2$, $F_1 \vee F_2$ et $F_1 \Rightarrow F_2$ sont F_1 et F_2. L'unique sous-formule immédiate de $\neg F$ est F. A, $\top$, $\bot$ n'ont pas de sous-formule immédiate.) Par exemple :

$$\begin{array}{rcl} (A \Rightarrow B) \Rightarrow C & \to & \neg(\neg A \vee B) \vee C \\ & \to & (\neg\neg A \wedge \neg B) \vee C \\ & \to & (A \wedge \neg B) \vee C \\ & \to & (A \vee C) \wedge (\neg B \vee C) \end{array}$$

On notera $F \longrightarrow G$ si l'on obtient G à partir de F en appliquant l'une des règles à une sous-formule de F, et $F \longrightarrow^* G$ si $F = F_0 \longrightarrow F_1 \longrightarrow \ldots \longrightarrow F_n = G$ pour une certaine suite F_0, F_1, $\ldots$, F_n ($n \geq 0$). Il est clair que ces règles préservent la sémantique : si $F \longrightarrow G$ alors $\mathcal{C}\,[\![F]\!]\,\rho = \mathcal{C}\,[\![G]\!]\,\rho$ pour tout environnement ρ.

Il est d'autre part clair que toute forme normale, c'est-à-dire toute formule irréductible par ces règles, est une conjonction finie de disjonctions finies de littéraux, donc, à peu de choses près, un ensemble fini de clauses.

Le fait que ce système de réécriture termine, et donc que l'on puisse trouver une formule G telle que $F \longrightarrow^* G$ et G soit en forme normale, nous permettra de conclure à l'existence d'un algorithme qui convertit toute formule en une forme clausale équivalente. Encore faut-il démontrer que ceci termine, et ce n'est pas tout à fait trivial.

Définissons plutôt une procédure de mise en forme clausale directe. Celle-ci applique essentiellement les règles (7), mais de façon optimisée. (On rappelle ici que $\Box$ est la clause vide.)

Proposition 3.1 *La fonction cl, définie récursivement comme suit, où s est un signe, + ou*

$-$:

$$\begin{aligned}
cl(s, A) &= \{s\ A\} \\
cl(+, \top) &= \emptyset \\
cl(-, \top) &= \{\Box\} \\
cl(+, \bot) &= \{\Box\} \\
cl(-, \bot) &= \emptyset \\
cl(+, \neg F_1) &= cl(-, F_1) \\
cl(-, \neg F_1) &= cl(+, F_1) \\
cl(+, F_1 \wedge F_2) &= cl(+, F_1) \cup cl(+, F_2) \\
cl(-, F_1 \wedge F_2) &= \mathit{shuffle}(cl(-, F_1), cl(-, F_2)) \\
cl(+, F_1 \vee F_2) &= \mathit{shuffle}(cl(+, F_1), cl(+, F_2)) \\
cl(-, F_1 \vee F_2) &= cl(-, F_1) \cup cl(-, F_2) \\
cl(+, F_1 \Rightarrow F_2) &= \mathit{shuffle}(cl(-, F_1), cl(+, F_2)) \\
cl(-, F_1 \Rightarrow F_2) &= cl(+, F_1) \cup cl(-, F_2)
\end{aligned}$$

où $\mathit{shuffle}(S_1, S_2) = \{C_1 \vee C_2 \mid C_1 \in S_1, C_2 \in S_2\}$*, est telle que* $\rho \models F$ *si et seulement si* $\rho \models cl(+, F)$ *pour tout environnement* ρ*, et est calculable.*

Démonstration. La calculabilité est évidente, à condition de savoir dérouler une fonction récursive, à l'aide d'une pile auxiliaire, sous forme d'une machine de Turing. Pour l'équivalence, on démontre par récurrence structurelle sur F que non seulement $\rho \models F$ si et seulement si $\rho \models cl(+, F)$, mais encore que $\rho \not\models F$ si et seulement si $\rho \models cl(-, F)$. La nécessité de démontrer ces deux faits simultanément est dûe aux formules niées, ainsi qu'aux implications.

Le cas le plus intéressant est celui de $cl(+, F_1 \vee F_2)$: si $\rho \models F_1 \vee F_2$ alors $\rho \models F_1$ ou $\rho \models F_2$; dans le premier cas, ρ satisfait toutes les clauses C_1 de $cl(+, F_1)$ par hypothèse de récurrence, donc aussi toutes les clauses de la forme $C_1 \vee C_2$, pour tout $C_2 \ldots$ donc celles de $\mathit{shuffle}(cl(+, F_1), cl(+, F_2)) = cl(+, F_1 \vee F_2)$. De même si $\rho \models F_2$. Réciproquement, si $\rho \models cl(+, F_1 \vee F_2)$, c'est-à-dire si $\rho \models \mathit{shuffle}(S_1, S_2)$ avec $S_1 = cl(+, F_1)$ et $S_2 = cl(+, F_2)$, il suffit de démontrer que $\rho \models S_1$ ou $\rho \models S_2$. Supposons par contradiction qu'il existe une clauses C_1 de S_1 telle que $\rho \not\models C_1$, et une clauses C_2 de S_2 telle que $\rho \not\models C_2$: alors $\rho \not\models C_1 \vee C_2$, et comme $C_1 \vee C_2$ est dans $\mathit{shuffle}(S_1, S_2)$, on aurait $\rho \not\models \mathit{shuffle}(S_1, S_2)$, contradiction. Les autres cas sont similaires ou faciles. □

La traduction de la proposition 3.1 est classique, mais ne répond pas tout à fait à la question posée en début de section : elle n'est *pas* en temps polynomial. Le coupable est la fonction *shuffle*, qui fabrique à partir de S_1 et de S_2 un ensemble de clauses dont la taille est au moins le produit des tailles de S_1 et de S_2.

Concrètement, posons F_n la formule $(A_1 \wedge \neg A_1) \vee (A_2 \wedge \neg A_2) \vee \ldots \vee (A_n \wedge \neg A_n)$, de taille quasi-linéaire en n. (Elle n'est pas de taille linéaire, car il faut de l'ordre de $\log n$ bits pour coder les numéros de chaque variable A_i, lorsqu'il y en a n. Cette formule est en fait de

taille de l'ordre de $n \log n$.) Il est facile de voir que $clauses(+, F_n)$ est l'ensemble de toutes les clauses $\pm_1 A_1 \vee \pm_2 A_2 \vee \ldots \vee \pm_n A_n$, où les signes $\pm_i$ $(1 \leq i \leq n)$ sont pris parmi $+$, $-$. Cet ensemble est donc de cardinal 2^n. Il n'y a en particulier aucun moyen de fabriquer cet ensemble en temps polynomial — rappelons qu'en une étape de calcul, une machine de Turing ne peut allouer qu'une nouvelle case sur sa bande, donc elle ne peut produire que des objets de taille polynomiale en temps polynomial.

Il existe cependant une traduction en forme clausale qui ne prend qu'un temps polynomial. Mais l'ensemble résultant S de clauses, au lieu d'être équivalent à la formule de départ F, ne sera plus qu'*équisatisfiable* avec F : S sera satisfiable si et seulement si F l'est. (C'est exactement ce que le fait FORM-SAT $\preceq_{\mathbf{P}}$ SAT nous garantissait, pas plus.) Cette traduction est due à Tseitin dans les années 1950.

Proposition 3.2 *Pour toute formule propositionnelle F, soit x_G une variable propositionnelle fraîche pour chaque sous-formule non variable de F (hors de $\mathrm{FV}(F)$ et distinctes deux à deux). Posons $x_A = A$ par convention, pour tout $A \in \mathrm{FV}(F)$. Pour toute sous-formule G non variable de F, soit $def(G)$ la formule :*

- $x_G \Leftrightarrow x_{G_1} \wedge x_{G_2}$ *si* $G = G_1 \wedge G_2$;
- $x_G \Leftrightarrow x_{G_1} \vee x_{G_2}$ *si* $G = G_1 \vee G_2$;
- $x_G \Leftrightarrow \neg x_{G_1}$ *si* $G = \neg G_1$;
- $x_G \Leftrightarrow (x_{G_1} \Rightarrow x_{G_2})$ *si* $G = G_1 \Rightarrow G_2$;
- $x_\top$ *si* $G = \top$;
- $\neg x_\bot$ *si* $G = \bot$.

Posons $ts(F)$ l'union des $cl(+, def(G))$ lorsque G parcourt les sous-formules non variables de F, et de la clause $+x_F$.

Alors $ts(F)$ est un ensemble de clauses calculable en temps polynomial à partir de F, et $ts(F)$ est satisfiable si et seulement si F l'est.

Démonstration. Il n'y a qu'un nombre linéaire de sous-formules G de F, et pour chacune on fabrique la forme clausale $cl(+, def(G))$ d'une formule $def(G)$ n'ayant qu'un nombre *constant* de symboles. (Au plus deux connecteurs logiques, et trois variables.) Donc $ts(F)$ se calcule en temps polynomial.

Si F est satisfiable, soit ϱ un environnement partiel de domaine $\mathrm{FV}(F)$ qui satisfait F. On étend ϱ en un environnement partiel ϱ' dont le domaine contienne en outre toutes les variables x_G, G sous-formule non variable de F, en posant $\varrho'(x_G) = \mathcal{C}[\![G]\!]\,\varrho$. Alors ϱ' satisfait toutes les formules $def(G)$ par définition, donc aussi $cl(+, def(G))$ par la proposition 3.1. Finalement, ϱ' satisfait $+x_F$, puisque $\varrho'(x_F) = \mathcal{C}[\![F]\!]\,\varrho = 1$.

Réciproquement, si $ts(F)$ est satisfiable, soit ρ un environnement qui le satisfait. En utilisant la proposition 3.1, ρ satisfait toutes les formules $def(G)$, donc par récurrence sur les sous-formules G de F, $\rho \models G$ si et seulement si $\rho(x_G) = 1$. Comme ρ satisfait $+x_F$, ρ satisfait donc F, en prenant $G = F$. □

3.2 La méthode de Davis-Putnam-Logemann-Loveland (DPLL)

La méthode DPLL est essentiellement une recherche par force brute d'un environnement partiel ϱ satisfaisant toutes les clauses C de l'ensemble de clauses en entrée. La règle de base est celle de *splitting*, qui consiste à choisir une variable A libre dans l'ensemble courant de clauses S, et à tester si S est satisfaite par un environnement qui rend A vraie, ou bien par un environnement qui rend A fausse, récursivement. Pour ceci, on remplace A par vrai, resp. faux, dans S, et l'on simplifie. Notons $\rho[A := 0]$, $\rho[A := 1]$ l'environnement qui à A associe 0, resp. 1, et à toute autre variable B associe $\rho(B)$.

Lemme 3.3 *Pour toute variable A, et tout ensemble de clauses S, on note $S[A := \bot]$ l'ensemble obtenu en enlevant de S toutes les clauses contenant le littéral $-A$ et en effaçant le littéral $+A$ dans les clauses restantes ; on note $S[A := \top]$ l'ensemble obtenu en enlevant de S toutes les clauses contenant le littéral $+A$ et en effaçant le littéral $-A$ dans les clauses restantes.*

Pour tout environnement ρ, $\rho[A := 0] \models S$ si et seulement si $\rho \models S[A := \bot]$, et $\rho[A := 1] \models S$ si et seulement si $\rho \models S[A := \top]$.

En particulier, S est satisfiable si et seulement si $S[A := \top]$ ou $S[A := \bot]$ est satisfiable.

Démonstration. Si $\rho[A := 0]$ satisfait une clause C, alors soit A n'est pas libre dans C, et alors $\rho \models C$ par le lemme 1.2 ; soit C contient $-A$ et C n'apparaît pas dans $S[A := \bot]$; soit C ne contient pas $-A$ mais contient $+A$, c'est-à-dire s'écrit $C' \vee +A$, et alors $\rho[A := 0] \models C'$ (puisque $\rho[A := 0] \not\models +A$), donc $\rho \models C'$ par le lemme 1.2. On en déduit que si $\rho[A := 0] \models S$, alors ρ satisfait toutes les clauses de $S[A := \bot]$. Réciproquement, si $\rho \models S[A := \bot]$, alors pour chaque clause C de S : si A n'est pas libre dans C, alors $\rho[A := 0] \models C$ par le lemme 1.2 ; si C contient $-A$, $\rho[A := 0] \not\models A$ donc $\rho[A := 0] \models C$; et si C contient $+A$ mais pas $-A$, alors C s'écrit $C' \vee +A$, avec $\rho \models C'$ donc $\rho[A := 0] \models C'$ par le lemme 1.2, donc $\rho[A := 0] \models C$. De même pour $\rho[A := 1]$ et $S[A := \top]$.

Si S est satisfiable, disons $\rho \models S$, alors soit $\rho(A) = 0$, donc $\rho = \rho[A := 0]$, $\rho[A := 0] \models S$, donc $\rho \models S[A := \bot]$; soit $\rho(A) = 1$ et donc $\rho \models S[A := \top]$. Réciproquement, si $S[A := \top]$ est satisfiable, disons $\rho \models S[A := \top]$, alors $\rho[A := 1] \models S$; de même, si $S[A := \bot]$ est satisfiable, alors $\rho[A := 0] \models S$. □

On teste alors la satisfiabilité de S en choisissant $A \in \mathrm{FV}(S)$, et en testant récursivement si $S[A := \top]$ ou $S[A := \bot]$ est satisfiable. Ceci s'arrête lorsque S contient la clause vide $\Box$, auquel cas S est insatisfiable, ou bien lorsque S ne contient pas la clause vide mais n'a aucune variable libre : alors S est vide, et donc satisfiable.

Le choix de A à chaque étape est arbitraire. Il existe différentes stratégies de choix de A, de sorte à accélérer la recherche. Une de celles-ci est l'heuristique de Jeroslow-Wang, décrite plus bas.

La règle de splitting seule ramène la satisfiabilité d'un ensemble de clauses à m variables à la satisfiabilité de deux ensembles à $m - 1$ variables. Une procédure fondée sur la règle de splitting seule est donc en temps exponentiel, et pas seulement dans le cas le pire.

La force de la procédure DPLL est de reconnaître certaines situations particulières où l'on peut progresser en évitant la règle de splitting. La plus importante est la *résolution*

unitaire (un cas particulier de la résolution, voir la section 3.3) :

Lemme 3.4 *Soit S un ensemble de clauses, et supposons que S contienne une clause* unitaire*, c'est-à-dire une clause contenant un unique littéral, $+A$ ou $-A$. Alors S est satisfiable si et seulement si $S[A := \top]$ l'est (si $+A \in S$), resp. si $S[A := \bot]$ l'est (si $-A \in S$).*

Démonstration. Supposons $+A \in S$, l'autre cas étant similaire. Si $\rho \models S$, alors $\rho \models +A$, donc $\rho(A) = 1$. Alors $\rho = \rho[A := 1]$, donc $\rho \models S[A := \top]$ par le lemme 3.3. Réciproquement, si $\rho \models S[A := \top]$ alors $\rho[A := 1] \models S$. □

On préférera appliquer la règle de résolution unitaire, c'est-à-dire remplacer S par $S[A := \top]$ (si $+A \in S$) ou par $S[A := \bot]$ (si $-A \in S$), en priorité, avant d'appliquer la règle de splitting. La règle de résolution unitaire, en effet, ne fait que simplifier le problème. De plus, appliquer la règle de résolution unitaire peut permettre de la réappliquer : par exemple, en partant des clauses $+A$, $-A \vee +B$, et $-B \vee +C$, l'application de la règle de résolution unitaire sur $+A$ fournit $+B$ et $-B \vee +C$, et on peut la réappliquer sur $+B$ pour obtenir $+C$.

Une dernière règle de simplification, proposée par Davis, Logemann et Lovelant, est l'*élimination de clauses pures*. On dit qu'un atome A est *pur* dans S si et seulement s'il apparaît toujours avec le même signe ; autrement dit, si $A \in \mathrm{FV}(S)$, et soit $-A$ n'apparaît pas dans S, soit $+A$ n'apparaît pas dans S. On dit alors que $+A$ (resp., $-A$) est un *littéral pur* dans S, et que les clauses C de S contenant $+A$ (resp., $-A$) sont *pures*. On a :

Lemme 3.5 *Soit S un ensemble de clauses, et P le sous-ensemble des clauses de S qui sont pures dans S. Alors S est satisfiable si et seulement si $S \setminus P$ est satisfiable.*

Démonstration. Si S est satisfiable, $S \setminus P$ l'est aussi, en tant que sous-ensemble. Réciproquement, supposons que A soit pur dans S, disons que $-A$ n'apparaisse pas dans S. (Le cas de $+A$ est similaire.) Alors $S \setminus P$ est juste $S[A := \top]$. Il s'ensuit que si $S \setminus P = S[A := \top]$ est satisfiable, disons $\rho \models S[A := \top]$, alors $\rho[A := 1] \models S$, donc S est satisfiable. □

On remarque finalement que l'on peut aussi supprimer de S toute *tautologie*, c'est-à-dire toute clause de la forme $C \vee +A \vee -A$: si S est satisfiable, S privé de ses tautologies l'est aussi, et réciproquement.

On en déduit la procédure DPLL, écrite dans un style fonctionnel :

DPLL (S) =
  si $S = \emptyset$ **alors retourner** vrai ;
  sinon, si $\square \in S$ **alors retourner** faux ;
  sinon, si S contient une tautologie C **alors retourner** DPLL $(S \setminus \{C\})$;
  sinon, si S contient un littéral $+A$ (resp., $-A$)
    alors retourner DPLL $(S[A := \top])$ (resp., DPLL $(S[A := \bot])$) ;
  sinon, si $+A$ (resp., $-A$) est pur dans S
    alors retourner DPLL $(S[A := \top])$ (resp., DPLL $(S[A := \bot])$) ;
  sinon choisir $A \in \mathrm{FV}(S)$;
    si DPLL $(S[A := \top])$ **alors retourner** vrai ;
    sinon retourner DPLL $(S[A := \bot])$;

Les considérations précédentes montrent que DPLL (S) retourne vrai si et seulement si S est satisfiable, et faux sinon. On reconnaît les tests de terminaison ($S = \emptyset$, $\Box \in \emptyset$), l'élimination de tautologies, la résolution unitaire, l'élimination de clauses pures, puis le splitting. Ici, nous avons décidé de tester si $S[A := \top]$ était satisfiable avant de tester $S[A := \bot]$, mais on peut le faire dans l'ordre inverse. C'est une question de stratégie.

Pour tout littéral L, définissons $S[L := \top]$ et $S[L := \bot]$ par : $S[+A := \top] = S[-A := \bot] = S[A := \top]$, $S[+A := \bot] = S[-A := \top] = S[A := \bot]$. Le choix de A, ainsi que de l'ordre du test de satisfiabilité entre $S[A := \top]$ et $S[A := \bot]$, revient à choisir un littéral $L = \pm A$, et a tester ensuite $S[L := \top]$ d'abord, puis $S[L := \bot]$.

Le choix d'un tel littéral $L = \pm A$ est en général effectué à l'aide d'heuristiques. L'heuristique *MOM* choisit l'un des littéraux L qui apparaît le plus souvent parmi les clauses de longueur minimale dans S. Celle de *Jeroslow-Wang* estime la probabilité de chaque littéral L de satisfaire S, en calculant $JW(L) = \sum_C 1/2^{\#C}$, où $\#C$ est le nombre de littéraux dans C, et la somme porte sur toutes les clauses C contenant L. On choisit ensuite un littéral L qui maximise $JW(L)$. On pourra consulter Gent et Walsh [7] pour une discussion de ces heuristiques, et pour une discussion des techniques de codage de DPLL en pratique.

L'exercice suivant porte sur la classe importante des *clauses de Horn*. Une clause de Horn est par définition une clause contenant au plus un littéral positif. Une clause contenant exactement un littéral positif, c'est-à-dire de la forme $-A_1 \vee -A_2 \vee \ldots \vee -A_n \vee +A$, est appelée une *clause définie*, A est sa *tête*, et $A_1, A_2, \ldots, A_n$ est son *corps*. On la notera souvent $A \Leftarrow A_1, A_2, \ldots, A_n$, traduisant l'idée qu'il s'agit réellement d'une implication. Une clause définie de corps vide sera juste notée A ou $+A$: c'est un *fait*. Une clause ne contenant aucun littéral positif est appelée une *clause négative*, ou un *but* $-A_1 \vee -A_2 \vee \ldots \vee -A_n$. Elle sera souvent notée $\bot \Leftarrow A_1, A_2, \ldots, A_n$, et son corps est défini comme dans le cas des clauses définies. Une clause de Horn est alors soit une clause définie soit un but.

▷ Exercice 3.1

Soit S un ensemble de clauses de Horn. Considérons la règle de *résolution unitaire positive* : si S contient un fait $+A$, remplacer S par $S[A := \top]$. (C'est l'instance de la règle de résolution unitaire, restreinte aux littéraux positifs.) Montrer que, si S ne contient pas la clause vide et si la règle de résolution unitaire positive ne s'applique pas à S, alors S est satisfiable.

▷ Exercice 3.2

En déduire une modification, ou plutôt une simplification de l'algorithme de DPLL qui décide le problème suivant HORN-SAT en temps polynomial :
ENTRÉE : un ensemble fini, S, de clauses de Horn ;
QUESTION : S est-il satisfiable ?
Donc HORN-SAT $\in \mathbf{P}$.

▷ Exercice 3.3

On considère le langage 3-SAT-NON-TRIV :
ENTRÉE : un ensemble non vide S de 3-clauses, dont aucune n'est une tautologie, aucune n'est une clause unitaire (réduite à un littéral), aucune n'est vide ($\Box$), et aucune n'est pure.

QUESTION : S est-elle satisfiable ?
Montrer que 3-SAT-NON-TRIV est **NP**-complet.

3.3 Résolution

Le premier article de Davis et Putnam (1960) n'utilisait pas la règle de splitting, mais celle de *résolution*. L'idée de la résolution est d'ajouter, petit à petit, à S des clauses qui sont conséquence logique de clauses de S, jusqu'à ce que l'on dérive la clause vide $\Box$ (contradiction : l'ensemble initial de clauses était donc insatisfiable), ou bien que l'on ne puisse plus dériver de nouvelle clause.

Ce n'est pas un algorithme très efficace pour décider SAT, loin s'en faut. Cependant, quelques instances bien choisies de la règle de résolution permettent d'accélérer d'autres algorithmes comme DPLL ; c'est notamment le cas sur les *2-clauses*, c'est-à-dire les clauses contenant au plus deux littéraux. (Voir par exemple l'exercice 3.4.) La résolution est aussi l'un des mécanismes utilisés pour résoudre d'autres problèmes propositionnels, comme le calcul d'impliquants premiers. Mais surtout, la résolution dans le cas propositionnel est le prototype d'une règle de démonstration automatique en logique du premier ordre, aussi appelée résolution, et qui est, dans ce cadre, très efficace.

Voici la règle de résolution :

$$\frac{C \vee +A \qquad -A \vee C'}{C \vee C'}$$

On la lit comme suit : étant donné un ensemble courant de clauses S, si l'on peut trouver deux clauses dans S de la forme donnée au-dessus de la barre (les *prémisses* de la règle), alors ajouter la conclusion de la règle (le *résolvant*) à S. On itère jusqu'à dériver la clause vide $\Box$, ou bien jusqu'à ce qu'on ne puisse plus dériver de résolvant qui ne soit pas déjà dans l'ensemble de clauses.

Notons $\longrightarrow$ la relation entre ensembles de clauses représentant ce processus : $S \longrightarrow S'$ si et seulement si $S' = S \cup \{C \vee C'\}$, où S contient deux clauses de la forme $C \vee +A$ et $-A \vee C'$. La résolution est d'abord correcte :

Lemme 3.6 (Correction) *Soit $S \longrightarrow S'$. Alors S' est* conséquence logique *de S : pour tout environnement ρ, si $\rho \models S$ alors $\rho \models S'$.*

En particulier, si $S \longrightarrow^* S'$ et S' contient la clause vide $\Box$, alors S est insatisfiable.

Démonstration. Il suffit de montrer que si ρ satisfait à la fois $C \vee +A$ et $-A \vee C'$, alors il satisfait $C \vee C'$. En effet, si $\rho(A) = 0$, le fait que $\rho \models C \vee +A$ implique $\rho \models C$, donc $\rho \models C \vee C'$. Le cas symétrique $\rho(A) = 1$ implique $\rho \models C'$, donc aussi $\rho \models C \vee C'$.

On en déduit par récurrence sur le nombre d'étapes $\longrightarrow$ pour passer de S à S' tel que $S \longrightarrow^* S'$ que S' est encore conséquence logique de S. Comme aucun environnement ne satisfait $\Box$, si $\Box \in S'$, alors S' est insatisfiable, donc S aussi. $\Box$

La règle de résolution est en fait aussi complète : si S est insatisfiable, on peut en dériver la clause vide par un nombre fini d'instances de la règle de résolution. La démonstration fait appel à la notion d'arbre sémantique, déjà vue à la section 1.2 et à la section 1.3.

Théorème 3.7 (Complétude) *Soit S un ensemble insatisfiable de clauses propositionnelles. Alors il existe un ensemble de clauses S' tel que $S \longrightarrow^* S'$ et S' contient la clause vide $\square$.*

Démonstration. Si S est insatisfiable, fixons une énumération $A_0, A_1, \ldots, A_{n-1}$ de ses variables libres, et construisons son arbre sémantique T. En détail, soit T_0 l'arbre de tous les environnements, T l'arbre T_0 élagué aux nœuds d'échec. Montrons que l'on peut dériver la clause vide de S, par récurrence sur le nombre de nœuds de l'arbre sémantique T.

Si T n'a qu'un nœud, c'est-à-dire si sa racine est déjà un nœud d'échec, alors l'interprétation partielle vide rend fausse une clause C de S, donc tous les littéraux de C. On a donc $C = \square$, et le théorème est démontré, avec $S' = S$.

Sinon, T contient un nœud ϱ qui n'est pas d'échec mais dont les deux successeurs immédiats sont des nœuds d'échec. On appelle un tel nœud un *nœud d'inférence*. Par exemple, sur la figure 2, les nœuds 4, 6 et 7 sont des nœuds d'inférence. Il existe au moins un nœud d'inférence, car tout nœud de T qui n'est pas un nœud d'échec et qui est minimal (le plus bas possible dans T) est un nœud d'inférence.

Notons $\varrho[A_i := 1]$ est $\varrho[A_i := 0]$ les deux successeurs immédiats de ϱ. Ils sont obtenus à partir de ϱ en posant A_i égal à 1, resp. 0, où i est l'entier tel que le domaine de ϱ soit $\{A_0, A_1, \ldots, A_{i-1}\}$. Il existe une clause de S telle que $\varrho[A_i := 0]$ soit un nœud d'échec pour cette clause. En particulier, $\varrho[A_i := 0]$ rend faux tous les littéraux de cette clause. Par la définition des nœuds d'échec, son prédécesseur immédiat ϱ n'est pas un nœud d'échec pour cette clause, donc elle est de la forme $C \vee +A_i$ (et ϱ rend faux tous les littéraux de C). De même, $\varrho[A_i := 1]$ est un nœud d'échec pour une clause qui est nécessairement de la forme $C' \vee -A_i$. On peut alors appliquer la règle de résolution sur $C \vee +A_i$ et $C' \vee -A_i$ (et ϱ rend faux tous les littéraux de C').

Soit S_1 l'ensemble S union le résolvant $C \vee C'$. On note que ϱ rend faux tous les littéraux de C, ainsi que de C', donc de $C \vee C'$. L'arbre sémantique T_1 obtenu à partir de T_0 en élaguant aux nœuds d'échec de S_1 contient donc strictement moins de nœuds que T (les nœuds $\varrho[A_i := 0]$ et $\varrho[A_i := 1]$, notamment, n'y sont plus). Par hypothèse de récurrence, on a $S_1 \longrightarrow^* S'$ avec $\square \in S'$, d'où le résultat. $\square$

On applique la règle de résolution en *saturant* l'ensemble de clauses S, c'est-à-dire en ajoutant les résolvants de couples de clauses de S, ainsi que de clauses précédemment produites par la règle de résolution. Une réalisation classique de la résolution fonctionne en accumulant dans un ensemble Sat ("saturation") des clauses dont on a déjà calculé tous les résolvants, et l'on gère S comme une file, contenant toutes les clauses dont on n'a pas encore calculé les résolvants avec les clauses de Sat (ou de S) :

$Sat := \emptyset$;
tant que $S \neq \emptyset$
    choisir $C \in S$, $S := S \setminus \{C\}$;
    si $C = \square$ **alors retourner** faux ; (* insatisfiable *)
    si C est une tautologie **alors** ; (* passer à la clause suivante. *)
    sinon, si $C \in Sat$ **alors** ; (* idem *)

sinon pour tout résolvant C_1 entre C et une clause de $Sat \cup \{C\}$
$\quad S := S \cup \{C_1\}$;
$Sat := Sat \cup \{C\}$;

Notons que l'on élimine les tautologies, qui ne sont pas rajoutées à Sat : ceci préserve la complétude, car aucun nœud d'échec ne peut être un nœud d'échec pour une tautologie, laquelle est par définition vraie dans tout environnement. On élimine aussi les clauses C sont déjà dans Sat.

▷ Exercice 3.4

Montrer que le problème 2-SAT :
ENTRÉE : une liste finie, S, de 2-clauses ;
QUESTION : S est-elle satisfiable ?
est dans $\mathbf{P}$. On rappelle qu'une 2-clause est une clause contenant au plus deux littéraux.

▷ Exercice 3.5

Soit S un ensemble de clauses, avec $\mathrm{FV}(S) = \{A_0, A_1, \ldots, A_{n-1}\}$. Un *renommage* R de S est un sous-ensemble de $\mathrm{FV}(S)$. L'*application* $R[S]$ du renommage R à un littéral, une clause, ou un ensemble de clauses S est défini comme l'objet obtenu en changeant le signe de toutes les occurrences des atomes de R, et en laissant inchangé le signe des autres. Par exemple, si $R = \{A_1\}$ et $S = \{+A_1 \vee +A_2, -A_1 \vee -A_0\}$, on a $R[S] = \{-A_1 \vee +A_2, +A_1 \vee -A_0\}$. Disons qu'un ensemble de clauses S est *Horn-renommable* si et seulement s'il existe un renommage R tel que $R[S]$ soit un ensemble de clauses de Horn. Montrer que le langage HORN-REN :
ENTRÉE : un ensemble de clauses S ;
QUESTION : S est-il Horn-renommable ?
est réductible en temps polynomial à 2-SAT. En déduire que HORN-REN $\in \mathbf{P}$.

▷ Exercice 3.6

On considère le langage HORN-REN-SAT :
ENTRÉE : un ensemble de clauses Horn-renommable S ;
QUESTION : S est-elle satisfiable ?
Montrons, en utilisant une variante adéquate de DPLL, que HORN-REN-SAT $\in \mathbf{P}$. Attention : on rappelle que l'on doit décider non seulement si S est satisfiable, mais aussi si S est de la forme requise en ENTRÉE.

▷ Exercice 3.7

On considère la règle de *résolution ordonnée*. Étant donnée une énumération $A_0, A_1, \ldots, A_{n-1}$ des variables libres de S, disons que A_i est *maximal* dans une clause $C \vee \pm A_i$ si et seulement si les atomes A_j de C sont tous tels que $j < i$. La résolution ordonnée est la règle de résolution ordinaire :

$$\frac{C \vee +A \qquad -A \vee C'}{C \vee C'}$$

mais contrainte par le fait que A soit maximal dans les deux prémisses. Montrer que la résolution ordonnée est encore correcte et complète.

▷ **Exercice 3.8**

On a vu à l'exercice 2.3 que 3-SAT-3-OCC était **NP**-complet. Montrer que le problème analogue SAT-2-OCC est, lui, dans **P** :

ENTRÉE : une liste finie, S, de clauses, où chaque variable propositionnelle apparaît au plus 2 fois ;

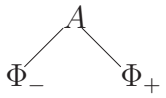
QUESTION : S est-elle satisfiable ?

3.4 Diagrammes de décision binaire (BDD)

Une autre idée qui fonctionne pour prouver des formules propositionnelles et qui vient d'idées sémantiques est celle des *diagrammes de décision binaire*, ou *BDD*. Le créateur des BDD tels que nous les connaissons aujourd'hui est Randall E. Bryant en 1986. Cependant, les BDD sont juste des arbres de décision avec quelques astuces bien connues en plus, et les arbres de décision remontent à George Boole (1854), si pas plus tôt. Les idées sont très simples, mais les réalisations informatiques sont usuellement plus complexes qu'avec les méthodes précédentes.

En gros, les astuces qui font que les BDD fonctionnent sont : d'abord, au lieu de représenter les arbres de décision comme des arbres en mémoire (où il y a un unique chemin de la racine à n'importe quel nœud), nous les représentons comme des *graphes orientés acycliques* ou *DAG*, autrement dit nous partageons tous les sous-arbres identiques. Ensuite, nous utilisons la règle de simplification suivante : si un sous-arbre a deux fils identiques, alors remplacer le sous-arbre par ce fils ; essentiellement, ce sous-arbre signifie "si A est vrai, alors utilise le fils de droite ; si A est faux, utilise le fils de gauche" : comme les fils de gauche et de droite coincident, il n'y a pas lieu d'effectuer une sélection fondée sur la valeur de A. Enfin, nous ordonnons les variables dans un ordre total donné $<$, et exigeons que, si nous descendons dans le BDD sur n'importe quel chemin, nous rencontrons les variables en ordre croissant. Cette dernière propriété assurera que les BDD sont des *représentants canoniques* de formules à équivalence logique près, c'est-à-dire que si Φ et Φ' sont deux formules équivalentes, alors leurs BDD construits sur le même ordre sont identiques.

À la base, les BDD sont soit **1** (vrai), **0** (faux) ou "si A alors Φ_+ sinon Φ_-", où Φ_+ et Φ_- sont des BDD distincts. Nous notons cette dernière forme $A \longrightarrow \Phi_+; \Phi_-$, ou sous forme d'arbre :



Cette représentation simple nous permet de construire des BDD morceau par morceau. Par exemple, pour construire le BDD de $\Phi \wedge \Phi'$, il suffit de combiner les BDD de Φ et de Φ'. Si le BDD de Φ est "si A alors Φ_+ sinon Φ_-", et celui de Φ' est "si A alors Φ'_+ sinon Φ'_-", alors le BDD de $\Phi \wedge \Phi'$ est simplement "si A alors $\Phi_+ \wedge \Phi'_+$ sinon $\Phi_- \wedge \Phi'_-$", où les deux conjonctions $\Phi_+ \wedge \Phi'_+$ et $\Phi_- \wedge \Phi'_-$ sont calculées récursivement. Le même principe s'applique aux disjonctions, implications, négations, et ainsi de suite, et est appelé le principe de décomposition de Shannon (voir plus bas).

On peut donc calculer les BDD de formules de bas en haut : par exemple, pour calculer le BDD de $(A \wedge B) \vee C$, nous calculons celui de A, à savoir $A \longrightarrow \mathbf{1}; \mathbf{0}$, celui de B, à savoir $B \longrightarrow \mathbf{1}; \mathbf{0}$, puis nous calculons leur conjonction, puis la disjonction de ce dernier avec le BDD de C. Le fait que les BDD sont des formes canoniques nous assure que la formule de départ est valide si et seulement si son BDD est exactement $\mathbf{1}$.

Définissons maintenant les BDD plus formellement. Soit $\mathbf{1}$ (vrai) et $\mathbf{0}$ (faux) deux nouveaux symboles (à ne pas confondre avec les valeurs de vérité 1 et 0, bien que ce soit ce qu'elles représentent.)

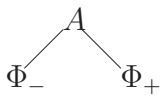
Disons que deux formules F et G sont *logiquement équivalentes* si et seulement si elles sont satisfaites par exactement les mêmes environnements, c'est-à-dire si et seulement si $\models F \Leftrightarrow G$. L'idée de base des BDD est donnée par le lemme suivant, appelé *principe de décomposition de Shannon*. On note $F[A := G]$ la formule obtenue à partir de F en remplaçant A par G.

Lemme 3.8 (Shannon) *Pour toute formule propositionnelle F, et toute variable propositionnelle A, F est logiquement équivalente à $(A \Rightarrow F[A := \top]) \wedge (\neg A \Rightarrow F[A := \bot])$. De plus, A n'est libre ni dans $F[A := \top]$ ni dans $F[A := \bot]$.*

Démonstration. On démontre d'abord que $\rho \models F[A := \top]$ si et seulement si $\rho[A := 1] \models F$, et que $\rho \models F[A := \bot]$ si et seulement si $\rho[A := 0] \models F$. Ceci est une récurrence simple sur la structure de F. Sachant ceci, si $\rho \models F$, et si $\rho(A) = 1$ alors $\rho[A := 1] \models F$ donc $\rho \models F[A := \top]$, et si $\rho(A) = 0$ alors $\rho[A := 0] \models F$ donc $\rho \models F[A := \bot]$. Dans tous les cas, $\rho \models (A \Rightarrow F[A := \top]) \wedge (\neg A \Rightarrow F[A := \bot])$. Réciproquement, si $\rho \models (A \Rightarrow F[A := \top]) \wedge (\neg A \Rightarrow F[A := \bot])$, on considère deux cas. Si $\rho(A) = 1$, on en déduit que $\rho \models F[A := \top]$, donc $\rho[A := 1] \models F$, c'est-à-dire $\rho \models F$. Si $\rho(A) = 0$, un raisonnement similaire montre que $\rho \models F$ de nouveau. □

Définition 3.9 (Graphes de Shannon) *Un* graphe de Shannon *est une formule construite sur les deux constantes $\mathbf{1}$, $\mathbf{0}$ au moyen du seul connecteur ternaire $\_ \longrightarrow \_; \_$. Plus précisément, l'ensemble des graphes de Shannon est le plus petit ensemble tel que $\mathbf{1}$ et $\mathbf{0}$ sont des graphes de Shannon, et tel que, si A est une variable, et Φ_+, Φ_- sont deux graphes de Shannon, alors $A \longrightarrow \Phi_+; \Phi_-$ ("si A alors Φ_+, sinon Φ_-") est un graphe de Shannon.*

$A \longrightarrow \Phi_+; \Phi_-$ sera aussi représenté graphiquement par :

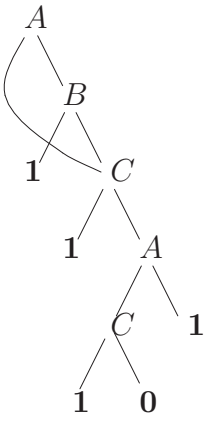


La sémantique des graphes de Shannon est donnée par :

$$\mathcal{B}[\![\mathbf{1}]\!]\rho = 1 \quad \mathcal{B}[\![\mathbf{0}]\!]\rho = 0 \quad \mathcal{B}[\![A \longrightarrow \Phi_+; \Phi_-]\!]\rho = \begin{cases} \mathcal{B}[\![\Phi_+]\!]\rho & \text{si } \rho(A) = 1 \\ \mathcal{B}[\![\Phi_-]\!]\rho & \text{si } \rho(A) = 0 \end{cases}$$

(Remarquer que la branche "A vrai" est à droite, alors que la branche "A faux" est à gauche, ce qui est l'ordre inverse de Φ_+ et Φ_-, mais correspond à la convention que nous avons prise pour les arbres sémantiques, comme à la figure 2.)

Nous représentons alors les formules construites avec $\wedge$, $\vee$, $\neg$, $\Rightarrow$, etc. par des graphes de Shannon logiquement équivalents. Par exemple, la formule $(((A \Rightarrow B) \wedge C) \Rightarrow A) \vee \neg C$ peut être représentée par le graphe de Shannon suivant :



où nous avons dessiné plusieurs nœuds **1** au lieu d'un dans l'intérêt de la lisibilité. (Vérifier que ceci est effectivement équivalent à la formule de départ, en énumérant toutes les affectations possibles sur A, B, C.) Ce n'est pas la seule représentation de la formule sous forme de graphe de Shannon, cependant.

Les graphes (ou arbres) de Shannon ont la propriété suivante, qui rend le calcul des opérations logiques aisé :

Théorème 3.10 (Orthogonalité) *Soit* $\Phi = A \longrightarrow \Phi_+; \Phi_-$, $\Phi' = A \longrightarrow \Phi'_+; \Phi'_-$.

La négation de Φ*, que nous notons* $\neg\Phi$*, est* $A \longrightarrow \neg\Phi_+; \neg\Phi_-$*. Plus formellement, ce dernier graphe est satisfait exactement par les affectations qui ne satisfont pas* Φ*.*

Pour tout connecteur binaire $\circ$ *(*$\wedge$*,* $\vee$*,* $\Rightarrow$*,* $\Leftrightarrow$*, respectivement),* $\Phi \circ \Phi' = A \longrightarrow (\Phi_+ \circ \Phi'_+); (\Phi_- \circ \Phi'_-)$ *; plus formellement, ce dernier est satisfait exactement par les affectations qui satisfont à la fois* Φ *et* Φ'*, resp.* Φ *ou* Φ'*, resp.* Φ' *ou* $\neg\Phi$*, resp. à la fois* Φ *et* Φ' *ou ni* Φ *ni* Φ'*.*

Démonstration. Immédiat. □

Nous pouvons faire mieux que les graphes de Shannon, et définir les BDD :

Définition 3.11 (BDD) *Soit* Φ *un graphe de Shannon, et* $<$ *un ordre strict total des variables de* $\mathrm{FV}(\Phi)$*.*

Nous disons que Φ *est un* diagramme de décision binaire*, ou* BDD*, ordonné par* $<$ *si et seulement si :*

- *(Réduit)* Φ *ne contient aucun sous-graphe de la forme* $A \longrightarrow \Phi'; \Phi'$ *(avec deux fils identiques) ;*
- *(Ordonné) tous les sous-graphes de* Φ *de la forme* $A \longrightarrow \Phi_+; \Phi_-$ *sont tels que* A *est strictement inférieur (pour* $<$*) à toutes les variables dans* $\mathrm{FV}(\Phi_+) \cup \mathrm{FV}(\Phi_-)$*.*

Le BDD de $(((A \Rightarrow B) \wedge C) \Rightarrow A) \vee \neg C$, avec $A < B < C$, est par exemple :

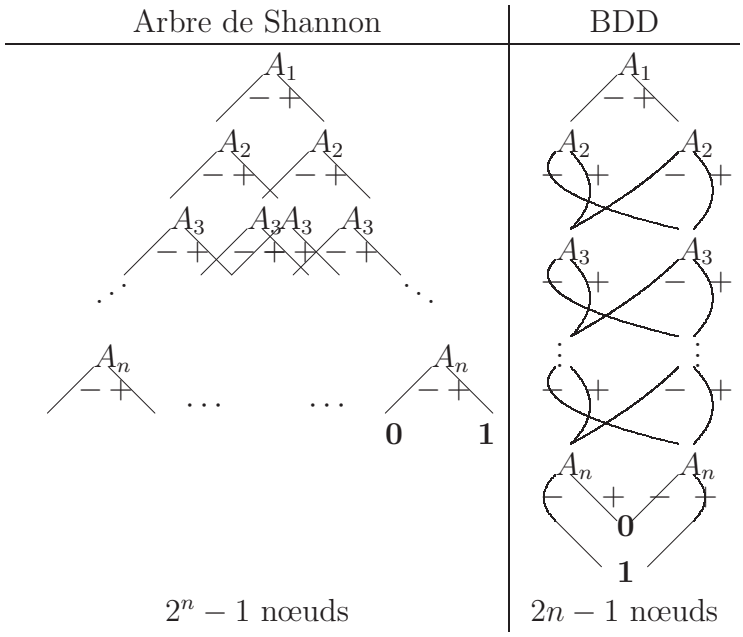
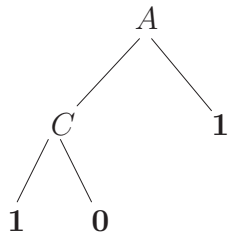


FIG. 7 – La taille comparée des BDD et des graphes de Shannon



À la figure 7 on a dessiné l'arbre de Shannon et le BDD de la formule F_n définie par récurrence sur n par : $F_1 = A_1$, $F_n = F_{n-1} \Leftrightarrow A_n$, où $A_1 < A_2 < \ldots < A_n$. On y constate qu'un BDD peut être exponentiellement plus compact qu'un graphe de Shannon.

On peut voir un BDD Φ comme un automate qui décide si une affectation ρ donnée satisfait Φ. En effet, partons de la racine du BDD et descendons comme suit : lorsque nous sommes à un nœud étiqueté A (nous considérons un sous-graphe $A \longrightarrow \Phi_+; \Phi_-$), prenons la branche de droite (Φ_+ dans l'exemple) si $\rho(A) = 1$, et la branche de gauche (Φ_-) si $\rho(A) = 0$. Ce processus termine lorsque nous arrivons à une feuille : si cette feuille est **1**, alors ρ satisfait Φ, alors que si elle est **0**, ρ ne satisfait pas Φ. Tout ceci est vrai parce qu'essentiellement, les BDD sont des représentations compactes d'arbres de décision.

Les BDD sont des formes canoniques des formules modulo équivalence logique (ce que les graphes de Shannon n'étaient pas) :

Théorème 3.12 (Canonicité) *Soient Φ et Φ' deux BDD construits sur le même ordre $<$. Alors Φ et Φ' sont logiquement équivalents si et seulement s'ils sont égaux.*

Démonstration. S'ils sont égaux, leur équivalence logique est claire.

Réciproquement, supposons Φ et Φ' logiquement équivalents. Nous montrons qu'ils sont égaux par récurrence sur le cardinal de $\mathrm{FV}(\Phi) \cup \mathrm{FV}(\Phi')$.

Si $n = 0$, alors Φ et Φ' sont dans $\{\mathbf{1}, \mathbf{0}\}$; comme Φ et Φ' sont logiquement équivalents, ils sont soit tous les deux $\mathbf{1}$ soit tous les deux $\mathbf{0}$.

Si $n \geq 1$, soit A la plus petite variable de $\mathrm{FV}(\Phi) \cup \mathrm{FV}(\Phi')$ pour l'ordre $<$ (autrement dit, la variable la plus haute). Sans perte de généralité, supposons que A est libre dans Φ. Comme A est la plus petite variable dans $\mathrm{FV}(\Phi)$, nécessairement $\Phi = A \longrightarrow \Phi_+; \Phi_-$ pour deux BDD Φ_+ et Φ_-.

Si A n'est pas libre dans Φ', alors Φ' est logiquement équivalent à Φ_+ et Φ_-. En effet, si $\rho \models \Phi'$, alors $\rho[A := 1] \models \Phi'$; comme Φ' est logiquement équivalent à Φ, $\rho[A := 1] \models \Phi$, donc $\rho[A := 1] \models \Phi_+$, donc $\rho \models \Phi_+$, puisque A n'est pas libre dans Φ_+. Réciproquement, si $\rho \models \Phi_+$, alors $\rho[A := 1] \models \Phi_+$, donc $\rho[A := 1] \models \Phi$, et par équivalence logique $\rho[A := 1] \models \Phi'$; comme A n'est pas libre dans Φ', $\rho \models \Phi'$. Donc Φ' est logiquement équivalent à Φ_+ ; par un argument similaire, Φ' est logiquement équivalent à Φ_-. Par hypothèse de récurrence (les cardinaux de $\mathrm{FV}(\Phi_+) \cup \mathrm{FV}(\Phi')$ et de $\mathrm{FV}(\Phi_-) \cup \mathrm{FV}(\Phi')$ sont en effet strictement inférieurs à n), Φ' égale à la fois Φ_+ et Φ_-. Mais c'est impossible, parce que Φ est réduit ($\Phi_+ \neq \Phi_-$).

Donc A est libre aussi dans Φ', et en fait $\Phi' = A \longrightarrow \Phi'_+; \Phi'_-$ pour deux BDD Φ'_+ et Φ'_-. Par des arguments similaires à ceux ci-dessus, Φ_+ est logiquement équivalent à Φ'_+ et Φ_- à Φ'_-, donc par hypothèse de récurrence $\Phi_+ = \Phi'_+$ et $\Phi_- = \Phi'_-$. Donc $\Phi = \Phi'$. □

En plus, les BDD de formules sont usuellement compacts. Si une formule est valide ou insatisfiable, son BDD est $\mathbf{1}$ ou $\mathbf{0}$, qui est évidemment très compact. (Mais les BDD de ses sous-formules, que nous devons construire pour calculer le BDD de la formule toute entière, peuvent être bien plus gros.) Même lorsqu'ils sont invalides et satisfiables, les BDD restent petits dans un nombre importants de cas pratiques. Il y a des exceptions, et dans le pire des cas les BDD peuvent avoir une taille exponentielle en le nombre de variables libres dans la formule de départ. (Pour être précis, si n est ce nombre, le nombre de nœuds dans un BDD ne dépasse pas $2^n/n$.)

La méthode usuelle (et la mieux connue) pour construire un BDD pour une formule donnée Φ est d'utiliser le théorème 3.10 pour le construire récursivement à partir des BDD de ses sous-formules.

Nous réalisons d'abord une fonction qui alloue et partage des triplets $A \longrightarrow \Phi'; \Phi''$ comme des enregistrements ("records") en mémoire avec trois champs pour A, Φ' et Φ'' respectivement. Pour ce faire, nous utilisons une table de hachage globale ("hash-table", cf. [8] ; c'est le type `Hashtbl.t` de OCaml) qui mémorise tous les triplets construits antérieurement. Les tables de hachage sont faites de sorte que, étant donnés A, Φ', Φ'', soit un triplet $A \longrightarrow \Phi'; \Phi''$ est déjà dans la table et on peut retourner son adresse rapidement, ou bien on annonce son absence rapidement. Un algorithme simple de table de hachage qui fonctionne bien en pratique consiste à fixer un entier N suffisamment grand (et premier pour de meilleurs résultats), et d'allouer un tableau global à N entrées, vides au départ. Ces entrées contiendront des listes chaînées de triplets déjà construits.

Pour découvrir si $A \longrightarrow \Phi'; \Phi''$ est dans la table, nous calculons d'abord une *fonction de*

hachage $h(A, \Phi', \Phi'')$ sur A, Φ', Φ'', qui retourne un entier entre 0 et $N-1$ (i.e., un index dans la table). Comme tous les BDD identiques sont partagés, ils sont décrits de façon unique par leur adresse en mémoire : un bon choix pour $h(A, \Phi', \Phi'')$ est la somme des adresses de A, Φ' et Φ'', modulo N. L'invariant de la table est que si $A \longrightarrow \Phi'; \Phi''$ est dans la table, il est dans la liste à l'entrée numéro i de la table, où $i = h(A, \Phi', \Phi'')$.

Pour trouver si $A \longrightarrow \Phi'; \Phi''$ est déjà dans la table, alors, calculer $i = h(A, \Phi', \Phi'')$; parcourir la liste dans l'entrée i, et comparer chaque élément avec l'enregistrement de composantes A, Φ' et Φ''. Si nous rencontrons le triplet attendu, nous retournons son adresse. Sinon, nous annonçons son absence. De même, pour allouer et partager $A \longrightarrow \Phi'; \Phi''$, nous le recherchons dans la table, et le retournons si nous l'avons trouvé ; sinon, nous allouons un nouvel enregistrement contenant A, Φ', Φ'', le rajoutons en tête de la liste à l'entrée numéro i, et retournons son adresse.

Le temps moyen pour retrouver ou construire et partager un triplet $A \longrightarrow \Phi'; \Phi''$ est de l'ordre de n/N, où n est le nombre de triplets antérieurement alloués. Si n n'est pas plus large que N dans de trop grandes proportions, ceci est quasi-instantané. (Ce calcul suppose que chaque opération d'accès à une case mémoire est en temps constant. Ce n'est pas le cas dans une machine de Turing à k bandes, mais bien sûr ici la notion de table de hachage est intéressante dans le cas de machines à accès direct à la mémoire, plus proches de nos ordinateurs usuels.)

Pour construire des BDD, nous définissons d'abord la fonction `BDDmake` comme suit : si $\Phi' = \Phi''$ (noter que nous comparons les BDD par adresse et non récursivement par contenu, puisqu'ils sont partagés), alors $\texttt{BDDmake}(A, \Phi', \Phi'')$ retourne Φ' ; sinon, $\texttt{BDDmake}(A, \Phi', \Phi'')$ alloue et partage $A \longrightarrow \Phi'; \Phi''$ et retourne son adresse.

La négation $\texttt{BDDneg}(\Phi)$ d'un BDD Φ est définie par récursion structurelle comme suit : $\texttt{BDDneg}(\mathbf{1}) = \mathbf{0}, \texttt{BDDneg}(\mathbf{0}) = \mathbf{1}, \texttt{BDDneg}(A \longrightarrow \Phi_+; \Phi_-) = \texttt{BDDmake}(A, \texttt{BDDneg}(\Phi_+), \texttt{BDDneg}(\Phi_-))$. Si $\circ$ est un opérateur binaire quelconque, nous définissons l'opération correspondante comme suit. Prenons l'exemple de la disjonction, réalisée par la fonction `BDDor` :

- $\texttt{BDDor}(\mathbf{1}, \Phi) = \mathbf{1}$, $\texttt{BDDor}(\mathbf{0}, \Phi) = \Phi$;
- $\texttt{BDDor}(\Phi, \mathbf{1}) = \mathbf{1}$, $\texttt{BDDor}(\Phi, \mathbf{0}) = \Phi$;
- si $\Phi = A \longrightarrow \Phi_+; \Phi_-$, et $\Phi' = A' \longrightarrow \Phi'_+; \Phi'_-$, alors :
  - si $A < A'$, alors
    $\texttt{BDDor}(\Phi, \Phi') = \texttt{BDDmake}(A, \texttt{BDDor}(\Phi_+, \Phi'), \texttt{BDDor}(\Phi_-, \Phi'))$;
  - si $A > A'$, alors
    $\texttt{BDDor}(\Phi, \Phi') = \texttt{BDDmake}(A', \texttt{BDDor}(\Phi, \Phi'_+), \texttt{BDDor}(\Phi, \Phi'_-))$;
  - si $A = A'$, alors
    $\texttt{BDDor}(\Phi, \Phi') = \texttt{BDDmake}(A, \texttt{BDDor}(\Phi_+, \Phi'_+), \texttt{BDDor}(\Phi_-, \Phi'_-))$.

Le seul problème si nous codons ces fonctions naïvement, est qu'elles tournent en temps environ proportionnel au nombre de *chemins* dans le BDD, et non son nombre de *nœuds*. C'est un point important, parce qu'à cause du partage, les BDD peuvent avoir moins de nœuds qu'ils n'ont de chemins, et ce dans des proportions astronomiques : le nombre des chemins peut atteindre une exponentielle en le nombre de nœuds, comme le montre la figure 7, où le BDD de droite (comme le graphe de Shannon de gauche) a 2^{n-1} chemins mais seulement

$2n - 1$ nœuds.

Nous codons donc `BDDneg` et `BDDor` en utilisant des *mémo-fonctions* : une mémo-fonction est une fonction f associée à une (autre) table de hachage T qui enregistre tous les résultats précédemment calculés par f. Pour être plus précis, T envoie des arguments de f vers les résultats correspondants de f. La mémo-fonction consulte d'abord la table de hachage T, et recherche une entrée correspondant à ses arguments. Si elle en trouve une, elle retourne le résultat enregistré. Sinon, elle appelle la fonction ordinaire f, enregistre le couple (argument, résultat) dans la table T et retourne le résultat. Cette astuce fait tourner `BDDneg` et `BDDor` en temps quasi-linéaire et quasi-quadratique respectivement sur une machine à accès direct à la mémoire.

▷ Exercice 3.9

Il y a au moins deux réalisations possibles de la fonction `BDDand`. L'une est dans le style de la définition donnée plus haut pour `BDDor`, et on demande de l'écrire explicitement (en s'économisant le code nécessaire pour en faire une mémo-fonction). L'autre consiste à poser $\texttt{BDDand}(\Phi_1, \Phi_2) = \texttt{BDDneg}(\texttt{BDDor}(\texttt{BDDneg}(\Phi_1), \texttt{BDDneg}(\Phi_2)))$. Sachant que toutes ces fonctions sont censées être des mémo-fonctions, discuter des avantages et des inconvénients de chaque solution, relativement à leur efficacité.

▷ Exercice 3.10

Montrer que le nombre de nœuds de $\texttt{BDDor}(\Phi_1, \Phi_2)$ est au plus le produit des nombres de nœuds de Φ et de Φ_2. Montrer que le nombre de nœuds de $\texttt{BDDneg}(\Phi_1)$ est linéaire en celui de Φ_1. En déduire que les opérations `BDDor`, `BDDneg`, `BDDand`, sont toutes en temps polynomial (sur une machine à accès direct, et en supposant pour simplifier que les recherches dans les diverses tables de hachage sont en temps constant — ce qui est en fait légèrement abusif).

Voici comment l'on peut tester la validité d'une formule à l'aide de BDD :

▷ Exercice 3.11

On convertit toute formule propositionnelle F en un BDD $BDD(F)$ comme indiqué plus haut, par $BDD(A) = A \longrightarrow \mathbf{1}; \mathbf{0}$, $BDD(\top) = \mathbf{1}$, $BDD(\bot) = \mathbf{0}$, $BDD(F_1 \vee F_2) = \texttt{BDDor}(BDD(F_1), BDD(F_2))$, $BDD(\neg F_1) = \texttt{BDDneg}(BDD(F_1))$, $BDD(F_1 \wedge F_2) = \texttt{BDDand}(BDD(F_1), BDD(F_2))$ (où `BDDand` est défini de l'une des façons possibles, à l'exercice 3.9), $BDD(F_1 \Rightarrow F_2) = \texttt{BDDor}(\texttt{BDDneg}(BDD(F_1)), BDD(F_2))$.

Montrer que $\models F$ si et seulement si $BDD(F)$ égale $\mathbf{1}$ (au sens de l'égalité des adresses). Comment peut-on tester la satisfiabilité de F à l'aide de $BDD(F)$?

▷ Exercice 3.12

Par l'exercice 3.10, toutes les opérations `BDDor`, `BDDneg`, `BDDand` sont en temps polynomial. (On admettra que ceci est effectivement aussi valable sur une machine de Turing.) Par l'exercice 3.11, on peut tester la satisfiabilité en faisant un calcul simple sur $BDD(F)$, et certainement en temps polynomial. Or FORM-SAT est **NP**-complet, par l'exercice 2.2. Peut-on en déduire $\mathbf{P} = \mathbf{NP}$. Sinon, où est l'erreur ?

▷ Exercice 3.13

Donner un algorithme **BDDexist** dans le style de **BDDor**, tel que $\mathcal{B}[\![\texttt{BDDexist}(A, \Phi)]\!]\,\rho = 1$ si et seulement si $\mathcal{B}[\![\Phi]\!]\,\rho[A := 1] = 1$ ou $\mathcal{B}[\![\Phi]\!]\,\rho[A := 0] = 1$. On peut penser à $\texttt{BDDexist}(A, \Phi)$ comme à une quantification existentielle $\exists A \cdot \Phi$, laquelle est informellement équivalente à $\Phi[A := \mathbf{1}] \vee \Phi[A := \mathbf{0}]$.

Les BDD sont en réalité souvent bien moins efficaces que DPLL par exemple pour résoudre SAT. En revanche, ils permettent de résoudre des problèmes bien plus compliqués. Le problème QPF suivant est en fait **PSPACE**-complet. **PSPACE**, la classe des langages décidables en temps fini et espace polynomial, est une classe dont on pense qu'elle est beaucoup plus grande que **NP** (on a **NP** $\subseteq$ **PSPACE**, mais l'on ne sait pas si **NP** = **PSPACE**)... l'étude de **PSPACE** fera l'objet du début du cours de complexité avancée (MPRI, M1).

▷ **Exercice 3.14**
Le langage des *formules propositionnelles quantifiées* est celui des formules propositionnelles, augmenté de deux opérateurs de quantification sur les variables propositionnelles. On définit :

F	::=	A	formule atomique, atome, variable propositionnelle
		$\top$	vrai
		$\bot$	faux
		$F \wedge F$	conjonction
		$F \vee F$	disjonction
		$\neg F$	négation
		$F \Rightarrow F$	implication
		$\exists A \cdot F$	quantification existentielle
		$\forall A \cdot F$	quantification universelle

On étend la fonction de sémantique par : $\mathcal{C}[\![\exists A \cdot F]\!]\rho = 1$ si et seulement si $[\![F]\!]\rho[A := 1] = 1$ ou $[\![F]\!]\rho[A := 0] = 1$, et $\mathcal{C}[\![\exists A \cdot F]\!]\rho = 1$ si et seulement si $[\![F]\!]\rho[A := 1] = 1$ et $[\![F]\!]\rho[A := 0] = 1$. Étendre la fonction BDD de l'exercice 3.11 à ces cas, et en déduire un algorithme à base de BDD décidant de la validité et de la satisfiabilité de formules propositionnelles quantifiées.

4 Quelques autres problèmes NP-complets

Il existe de très nombreux langages **NP**-complets. Nous en listons quelques-uns parmi les plus importants. La plupart des problèmes que nous mentionnerons ici sont des problèmes de graphes, orientés ou non.

Un *graphe non orienté* G est la donnée (V, E) d'un ensemble fini de *sommets* V, et d'un ensemble E d'*arêtes* $\{u, v\} \subseteq V$ de cardinal 2 (en particulier $u \neq v$). On dit que l'arête $\{u, v\}$ est *incidente* aux sommets u et v.

Un *graphe orienté* G est la donnée (V, E) d'un ensemble fini de sommets V, et d'un ensemble E d'*arcs* $(u, v) \in V \times V$, notés $u \to v$.

Il existe plusieurs représentations classiques des graphes, orientés ou non. La représentation par *matrice d'adjacence* numérote les sommets de 1 à n, et code E par une matrice de booléens $(e_{ij})_{1 \leq i,j \leq n}$, avec $e_{ij} = 1$ si et seulement s'il existe un arc (resp., une arête) (i, j) (resp., $\{i, j\}$) de i vers j. (Si le graphe est non orienté, on demande de plus que $e_{ij} = e_{ji}$ et $e_{ii} = 0$.) La représentation par *liste d'adjacence* code E comme un tableau, où $E[i]$ est la liste des sommets tels que $(i, j) \in E$ (resp., $\{i, j\} \in E$). Il est facile de voir que l'on peut passer d'une représentation à l'autre en temps polynomial.

4.1 INDEPENDENT SET, NODE COVER, CLIQUE

Un *ensemble indépendant* dans un graphe non orienté $G = (V, E)$ est un ensemble $C \subseteq V$ de sommets dont aucun n'est relié à aucun autre par une arête de G, c'est-à-dire tel que $u, v \in C$ implique $\{u, v\} \notin E$. Un problème classique en optimisation est de chercher un ensemble indépendant de cardinal maximal. Il n'existe aucune solution en temps polynomiale connue à ce problème. En effet, sinon, on pourrait décider, pour tout $k \in \mathbb{N}$, s'il existe un ensemble indépendant de cardinal au moins k. Or ce problème est **NP**-complet, donc pas en temps polynomial, sauf si $\mathbf{P} = \mathbf{NP}$:

Proposition 4.1 (INDEPENDENT SET) *Le langage INDEPENDENT SET :*
ENTRÉE : un graphe non orienté $G = (V, E)$, un entier $m \in \mathbb{N}$ écrit en unaire ou en binaire (peu importe) ;
QUESTION : G a-t-il un ensemble indépendant de cardinal au moins m ?
est **NP***-complet.*

Démonstration. D'abord, INDEPENDENT SET est dans **NP** : il suffit de deviner une partie C de V, et de vérifier qu'elle est de cardinal au moins m et qu'elle forme un ensemble indépendant.

Réciproquement, on réduit 3-SAT-NON-TRIV (exercice 3.3). On part donc d'un ensemble de 3-clauses S qui ne contient pas la clause vide, ni aucune tautologie ni aucune clause unitaire. Pour chaque clause $L_1 \vee L_2 \vee L_3$, on fabrique trois sommets frais formant un triangle. De même pour les clauses de taille 2 $L_1 \vee L_2$, on fabrique deux sommets frais reliés par une arête. (On appellera ceci encore un triangle... dégénéré.) Pour toute clause C contenant un littéral $+A$ et toute clause C' contenant le littéral opposé $-A$, on relie les sommets correspondants. Aucun ensemble indépendant ne peut contenir plus d'un sommet par triangle, donc, s'il y a m clauses dans S, aucun ensemble indépendant n'est de cardinal strictement supérieur à m. S'il existe un ensemble indépendant I de cardinal m, il contient exactement un sommet de chaque clique. Si un sommet étiqueté $+A$ est dans I, aucun sommet de I ne peut être étiqueté $-A$, et réciproquement : ceci fournit directement une affectation satisfaisant S. Réciproquement, si ρ est une affectation satisfaisant S, on forme un ensemble indépendant en sélectionnant un littéral vrai dans chaque clause. □

Corollaire 4.2 *Il n'existe aucun algorithme qui calcule en temps polynomial un ensemble indépendant de cardinal maximal d'un graphe non orienté donné en entrée, à moins que* $\mathbf{P} = \mathbf{NP}$.

Corollaire 4.3 (NODE COVER) *Un* recouvrement *C d'un graphe non orienté $G = (V, E)$ est un ensemble $C \subseteq V$ de sommets tel que toute arête de E est incidente à C, c'est-à-dire à au moins un élément de C. Le langage NODE COVER :*
ENTRÉE : un graphe non orienté $G = (V, E)$, un entier $m \in \mathbb{N}$ écrit en unaire ou en binaire (peu importe) ;
QUESTION : G a-t-il un recouvrement de cardinal au plus m ?
est **NP***-complet.*

Démonstration. C est un recouvrement de G (de cardinal au plus m) si et seulement si $V \setminus C$ est un ensemble indépendant de G (de cardinal au moins le cardinal de V moins m). □

Corollaire 4.4 (CLIQUE) *Une* clique C *d'un graphe non orienté* $G = (V, E)$ *est un sous-ensemble* $C \subseteq V$ *induisant un sous-graphe complet de* G*, c'est-à-dire tel que pour tous* $u, v \in C$ *avec* $u \neq v$*, on a* $\{u, v\} \in E$*. Le problème CLIQUE :*
ENTRÉE : un graphe non orienté $G = (V, E)$*, un entier* $m \in \mathbb{N}$ *écrit en unaire ou en binaire (peu importe) ;*
QUESTION : G *a-t-il une clique de cardinal au moins* m *?*
est **NP***-complet.*

Démonstration. Soit $\overline{G}$ le graphe *complémentaire* de G, c'est-à-dire $\overline{G} = (V, \overline{E})$, où $\overline{E} = \{\{u, v\} \mid u, v \in V, u \neq v, \{u, v\} \notin E\}$. Alors C est une clique de G si et seulement si C est un ensemble indépendant de $\overline{G}$. Comme le calcul de $\overline{G}$ se fait en temps polynomial, INDEPENDENT SET $\preceq_{\mathbf{P}}$ CLIQUE. Or CLIQUE $\in$ **NP** : il suffit de deviner la clique, de vérifier qu'elle est de cardinal au moins m, et que c'est bien une clique. □

4.2 Chemins et circuits hamiltoniens, eulériens

Un *chemin* dans un graphe non orienté $G = (V, E)$ est une suite de sommets $u_0, u_1, \ldots, u_n$ avec $\{u_{i-1}, u_i\} \in E$ pour tout i, $1 \leq i \leq n$. Un tel chemin est un *cycle* si et seulement si $u_n = u_0$, c'est-à-dire s'il revient à son point de départ. C'est un *cycle simple* si et seulement si pour tous i, j, $1 \leq i < j \leq n$, on a $u_i \neq u_j$, autrement dit le circuit ne passe pas deux fois par le même sommet (à part pour son sommet de départ et d'arrivée).

Un chemin est *hamiltonien* s'il passe par chaque sommet de G *exactement* une fois, autrement dit si tous les u_i, $0 \leq i \leq n$, sont distincts deux à deux d'une part, et tout sommet de V est un u_i pour un certain i.

La notion de *cycle hamiltonien* est similaire, à ceci près qu'on ne demande bien sûr pas que le sommet de départ n'apparaisse qu'une fois. On dit donc qu'un cycle est hamiltonien si et seulement si c'est un cycle simple qui passe par tous les sommets du graphe.

L'essentiel de cette section consiste à démontrer le théorème suivant. La démonstration est non triviale.

Théorème 4.5 (HAMILTONIAN PATH) *Le langage HAMILTONIAN PATH :*
ENTRÉE : un graphe $G = (V, E)$ *non orienté ;*
QUESTION : existe-t-il un chemin hamiltonien π *dans* G *?*
est **NP***-complet.*

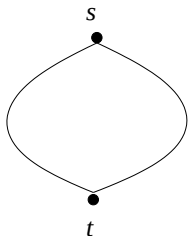
Démonstration. HAMILTONIAN PATH est clairement dans **NP** : deviner une suite de sommets de V, vérifier qu'elle définit un chemin dans G, qu'aucun sommet n'y apparaît deux fois, que tous les sommets apparaissent.

Pour la direction réciproque, on va réduire une variante de 3-SAT à HAMILTONIAN PATH. Il est nécessaire de recoder :

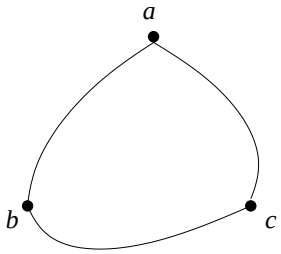
1. Les variables booléennes A, et le fait qu'elles sont soit vraies soit fausses ;

2. Les 3-clauses C, et le fait qu'au moins un littéral dans C est vrai ;
3. Le fait que les valeurs des occurrences des littéraux $\pm A$ dans les clauses C (décrites au point 2 ci-dessus) sont bien les mêmes que celles dictées par les choix faits au point 1 ci-dessus (condition de *cohérence*).

Il est intuitivement facile de réaliser un codage du point 1. Il suffit de fabriquer des sous-graphes avec deux sommets s et t, et tels que tout chemin hamiltonien aille de s à t en passant par deux chemins possibles, et seulement deux. Intuitivement, il nous faudra un *gadget de choix* qui ait grosso modo la forme suivante :

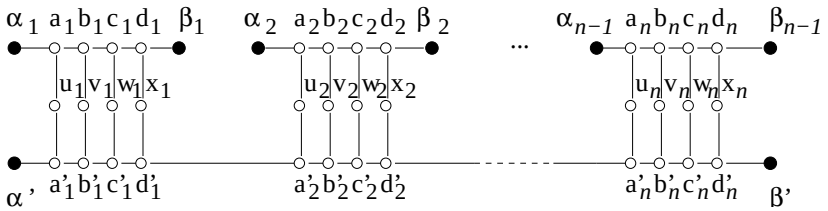


Pour coder le point 2, on va utiliser un gadget très utile (sous différentes formes) : le *triangle*. Il s'agit d'un sous-graphe avec trois sommets distingués a, b, c, et trois chemins distingués, intuitivement :



Les trois chemins représenteront les trois littéraux de la clause. Le chemin hamiltonien recherché passera par ab si la première variable de la clause est vraie, par bc si la seconde variables de la clause est vraie, etc. En réalité, dans notre cas, il sera plus facile de coder la contrainte 2 en demandant de passer par ab si la première variable est fausse, par bc si la deuxième est fausse, par ca si la troisième est fausse. En effet, un chemin hamiltonien ne pourra pas emprunter les trois sous-chemins ab, bc, ca, sinon il passera deux fois par le même sommet. Et ceci, via notre codage, signifiera qu'un des littéraux doit être vrai.

Reste le plus difficile : la contrainte de cohérence 3. Pour ceci, considérons le *gadget de cohérence* suivant, où $n \geq 1$:

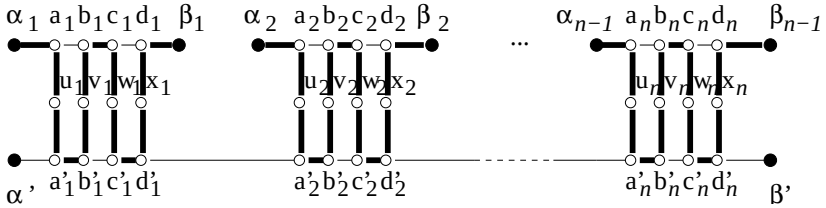


Nous noterons aussi α_0 le sommet α, et α_n le sommet β. On dira que ce gadget de cohérence est d'*arité n*.

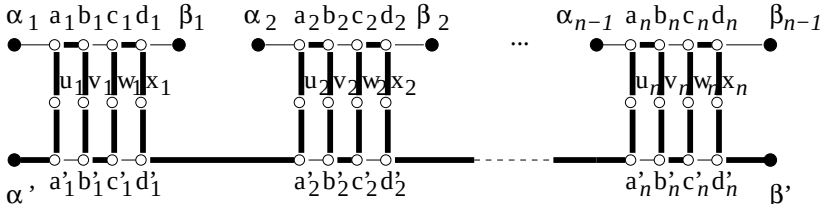
Observons les propriétés suivantes. Disons que ce gadget est *plongé* dans un graphe G si et seulement si on retrouve ce gadget comme sous-graphe, et que les seules arêtes incidentes aux sommets en creux (a_i, b_i, c_i, d_i, a'_i, b'_i, c'_i, d'_i, u_i, v_i, w_i, x_i, pour $1 \leq i \leq n$) sont celles montrées ci-dessus. Les sommets en plein ($\alpha = \alpha_0$, α_1, α_2, ..., α_{n-1}, $\alpha_n = \beta$, α' et β') peuvent être connectés à d'autres sommets de G. Les sommets du gadget de cohérence sont distincts deux à deux.

On observe alors :

Fait A. *Si le gadget de cohérence est plongé dans G, et π est un chemin hamiltonien de G dont aucune des deux extrémités n'est un sommet en creux, alors π passe par le gadget de cohérence de l'une des deux façons illustrées ci-dessous.*

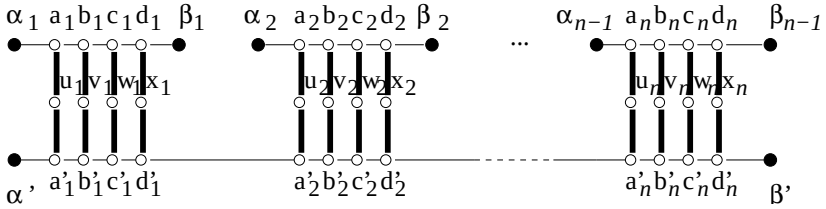


ou :



Démonstration. L'argument essentiel est que π passe par tout sommet en creux, et que comme ces sommets en creux ne sont pas des extrémités, le chemin π doit exhiber exactement deux arêtes incidentes à chaque sommet en creux. (Plus de deux contredirait le fait que le chemin ne passe pas deux fois par le même sommet. Moins de deux impliquerait que le sommet soit une extrémité.)

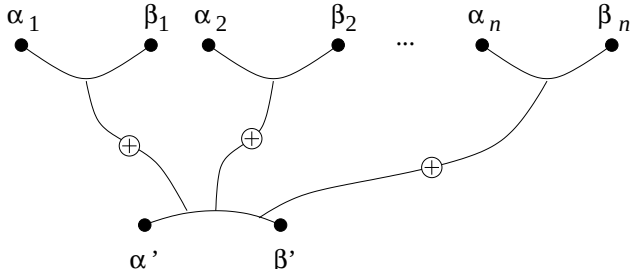
En considérant les sommets u_i, v_i, w_i, x_i pour tout i, $1 \leq i \leq n$, on voit donc que π passe au moins par les arêtes montrées en gras ci-dessous :



Considérons maintenant deux cas :

– π passe par l'arête b_1c_1 (laquelle existe car $n \geq 1$). Alors π ne passe pas par $b'_1c'_1$, sinon $b_1c_1c'_1b'_1$ formerait un cycle, contredisant l'hamiltonicité de π. Comme b'_1 est en creux, π passe par $a'_1b'_1$. Alors π ne peut pas passer par a_1b_1, sinon il créerait un cycle, donc il passe par $\alpha_1 a_1$. Ceci règle ce qui se passe à gauche de b_1c_1.
À droite de b_1c_1, comme c'_1 est en creux, π passe par $c'_1d'_1$. Donc il ne passe pas par c_1d_1, donc il passe par $d_1\beta_1$. Notons qu'il ne passe pas non plus par $d'_1a'_2$, sinon il y aurait trois arêtes incidentes à d'_1. Donc π passe par $a'_2b'_2$, et par le même raisonnement que plus haut, par $\alpha_2 a_2$, b_2c_2, $c'_2d'_2$, $d_2\beta_2$.
De proche en proche, c'est-à-dire par une récurrence sur i suivant les arguments ci-dessus, on montre ainsi que π passe par $a'_ib'_i$, puis $\alpha_i a_i$, b_ic_i, $c'_id'_i$, $d_i\beta_i$, pour tout i, $1 \leq i \leq n$. On est donc dans la première configuration énoncée dans le fait.
– π ne passe pas par l'arête b_1c_1. En considérant b_1, π doit passer par a_1b_1, donc par $\alpha' a'_1$ en regardant à gauche. De l'autre côté, π doit passer par $b'_1c'_1$, c_1d_1, $d'_1a'_2$.
Maintenant, considérons le sommet b'_2 : π ne peut pas passer par $a'_2b'_2$ sinon il y aurait trois arêtes incidentes à a'_2 dans π, donc π passe par $b'_2c'_2$. On en déduit que π ne passe pas par b_2c_2, donc passe par a_2b_2 ; puis que π passe par c_2d_2, puis par $d'_2a'_3$. De proche en proche comme dans le premier cas, on en déduit que l'on est dans la deuxième configuration énoncée dans le fait. □

Le gadget de cohérence code une sorte de ou exclusif : pour tout chemin hamiltonien π, soit π traverse le gadget de α' à β', et ne rentre pas dans le gadget via aucun des sommets α_i, β_i ; soit π traverse de α_1 à β_1, de α_2 à β_2, …, de α_n à β_n, mais pas de α' à β'. On abrégera dans la suite le gadget de cohérence sous forme de la notation :



Réduisons maintenant une instance de 3-SAT-NON-TRIV à HAMILTONIAN PATH. Soit donc S un ensemble de 3-clauses ne contenant pas la clause vide ni aucune clause unitaire, et sans clause pure. Toutes les clauses contiennent donc 2 ou 3 littéraux, et toute variable apparaît au moins une fois avec chaque signe, + ou −.

Partant de S, on fabrique le graphe G comme suit. On illustre la construction sur l'ensemble des clauses

$$A_1 \vee A_2 \vee A_3$$
$$\neg A_1 \vee \neg A_2 \vee \neg A_3$$
$$\neg A_1 \vee \neg A_2 \vee A_3$$

en figure 8.

Le graphe G a :

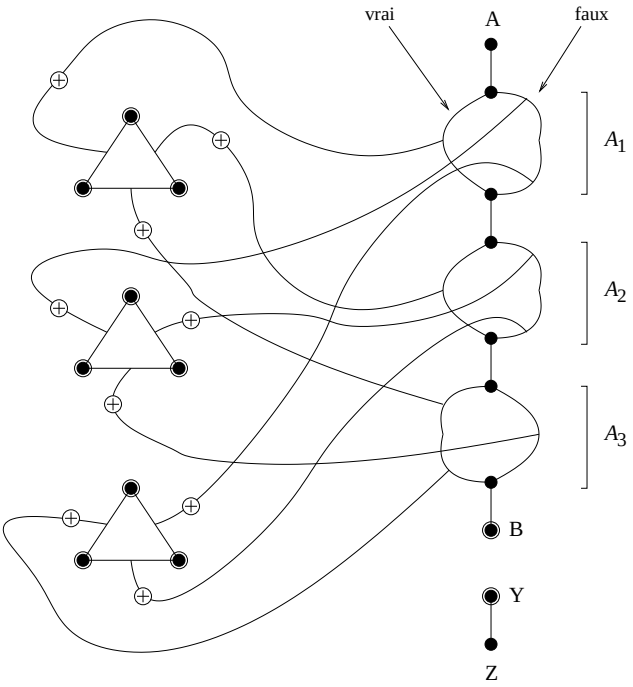
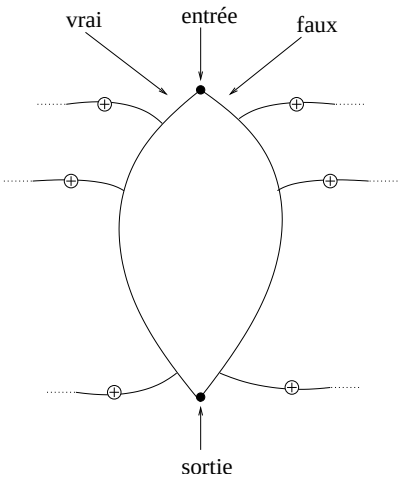


FIG. 8 – Codage des clauses $+A_1 \vee +A_2 \vee +A_3$, $-A_1 \vee -A_2 \vee -A_3$, $-A_1 \vee -A_2 \vee +A_3$

1. pour commencer, n gadgets de choix, un par variable A_i, $1 \leq i \leq n$. Le gadget de choix utilisé est plus compliqué que ce que notre description initiale pourrait laisser penser, pour des raisons qui seront éclaircies plus loin. Il est toujours de la forme :



où les chemins entre deux sommets pleins sont des côtés α'–β' de gadgets de cohérence. On notera que, par le fait A, tout chemin hamiltonien π dans le graphe que nous allons construire, tel que les extrémités de π ne sont pas sur des sommets en creux du gadget de choix ci-dessus (les sommets en creux étant ceux des gadgets de cohérence utilisés pour le construire), alors π passe de l'entrée à la sortie du gadget de choix ci-dessus

soit en passant du côté gauche ("vrai") soit du côté droit ("faux").

Dans le gadget de choix ci-dessus, correspondant à la variable A_i, le gadget de cohérence du côté vrai est d'arité n_i^+, où n_i^+ est le nombre d'occurrences de $+A_i$ dans S (c'est-à-dire, ici, le nombre de clauses où $+A_i$ apparaît). De manière symétrique, le gadget de cohérence du côté faux est d'arité n_i^-, où n_i^- est le nombre d'occurrences de $-A_i$ dans S. Comme le suggère l'exemple de la figure 8, cette construction permet de relier le côté A_i vrai du gadget de choix ci-dessus aux n_i^+ occurrences de $+A_i$ dans S, codées sous formes de côtés de triangles, et de relier le côté A_i faux du gadget de choix aux n_i^- occurrences de $-A_i$ dans S, elles aussi codées sous formes de côtés de triangles.

Les différents gadgets de choix sont alignés les uns au-dessus des autres, reliés l'un après l'autre par des arêtes simples (voir figure 8, partie droite). Le gadget de choix de A_1 est relié de la même façon à un sommet A (en haut), et celui de la dernière variable A_n est relié à un sommet B (en bas).

2. Ensuite, m triangles, un par clause de S (à gauche de la figure 8). Chaque côté du triangle correspond à un littéral de S. Ceci convient pour coder des clauses à exactement 3 littéraux. Pour des clauses à moins de 3 littéraux, on pourrait imaginer d'autres gadgets, mais ce n'est pas la peine. Par exemple, on peut voir la clause $A_1 \vee A_2$ comme étant donnée par les trois occurrences de littéraux A_1, A_2, et A_2 de nouveau ; autrement dit, on codera éventuellement certaines occurrences de littéraux sous forme de plusieurs arêtes d'un même triangle (et l'on calculera n_i^+ et n_i^- en conséquence).

   Chaque occurrence de littéral dans chaque clause est connectée via un gadget de cohérence à un gadget de choix. Ceci signifie que chaque triangle est en réalité formé de trois sommets pleins, reliés par 3 chemins α_i–β_i d'un gadget de cohérence relié à l'un des côtés d'un gadget de choix : si $+A_i$ apparaît dans la clause C, alors le gadget de cohérence relie le côté correspondant à A_i dans le triangle codant C au côté "vrai" du gadget de choix de la variable A_i ; si $-A_i$ apparaît dans C, alors le gadget de cohérence relie le côté $-A_i$ du triangle au côté "faux" du gadget de choix de la variable A_i.

   Tous les sommets impliqués sont distincts deux à deux.

3. Finalement, on a une arête entre deux sommets supplémentaires Y et Z (en bas à droite dans la figure 8), et des arêtes entre toutes les paires de sommets parmi B, Y, et les sommets pleins des triangles. Dans la figure 8, ces sommets sont représentés sous forme de sommets pleins entourés d'un petit cercle. Nous appellerons ces sommets les sommets *distingués*. Les arêtes les reliant seront les *arêtes distinguées*.

Ceci termine la description de la réduction de 3-SAT-NON-TRIV vers HAMILTONIAN PATH. Nous devons ensuite montrer que c'est bien une réduction.

Fait B. *S'il existe un chemin hamiltonien π à travers G, alors l'affectation ρ de valeurs de vérité qui attribue à chaque variable A_i la valeur vraie si π passe par le côté gauche ("vrai") du gadget de choix de A_i, et faux sinon, satisfait S.*

Démonstration. Les deux extrémités de π sont nécessairement les deux sommets de G de degré 1 (c'est-à-dire dont une seule arête part), c'est-à-dire les sommets A et Z. On peut donc voir π comme un chemin reliant A à Z. Le fait A s'applique puisque les extrémités de

π sont A et Z, qui ne sont pas des sommets en creux de gadgets de cohérence. Le chemin hamiltonien π passe donc de A au sommet juste en-dessous, puis de là par l'un des deux côtés du gadget de choix pour A_1, puis par l'un des deux côtés du gadget de choix pour A_2, et ainsi de suite jusqu'à A_n. Le chemin π arrive ensuite au sommet B. Ce faisant, π a défini sans ambiguïté l'affectation ρ.

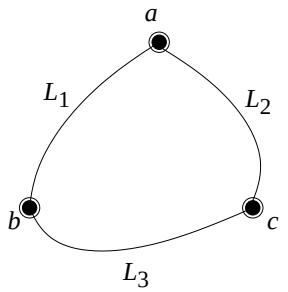
Le chemin π traverse ensuite les triangles dans un certain ordre, qui a peu d'importance, jusqu'à aboutir sur le sommet Y, et ensuite au sommet Z. Montrons que ρ satisfait chaque clause C de S. Considérons le triangle correspondant à C. Il est impossible que π passe par les 3 côtés de ce triangle, sinon π passerait deux fois par un même sommet. Considérons un des côtés par lesquels π ne passe pas, disons un littéral $-A_i$ (l'argument est symétrique pour $+A_i$). Par le fait A, π doit donc passer par l'autre côté du gadget de cohérence : c'est le côté "faux" du gadget de choix de la variable A_i. Donc ρ donne la valeur faux à A_i, donc $-A_i$ est vrai, et donc la clause C est bien vraie. □

Fait C. *Si ρ est une affectation de valeurs de vérité qui rend S vraie, alors il existe un chemin hamiltonien π à travers G.*

Démonstration. Le chemin π démarre au sommet A, descend le long des gadgets de choix pour chacune des variables $A_1, \ldots, A_n$, en passant à gauche si ρ rend A_i vraie, à droite si ρ rend A_i fausse. On arrive ainsi au sommet B.

Soient $C_1, \ldots, C_m$ les clauses de S. Comme ρ rend vraie chaque clause C_i, au moins un littéral par clause est vrai. Nous allons nous arranger pour que π passe exactement par les côtés de chaque triangle qui, vus en tant que littéraux, sont faux dans ρ. (On y est obligé, de toute façon, vu le fait A et le fait que les côtés de chaque triangle sont connectés par des gadgets de cohérence aux gadgets de choix.) De la sorte, π n'aura à passer que par au plus 2 côtés de chaque triangle.

Supposons pour simplifier qu'exactement deux des littéraux de C_1 soient faux, disons L_1 et L_2 dans le triangle suivant :



On fait arriver π sur le sommet b par une arête distinguée, π poursuit ensuite en parcourant le côté L_1, arrive sur le sommet a, passe sur le côté L_2, et arrive sur c.

On continue la construction en passant de c à la clause C_2, puis à $C_3, \ldots$, jusqu'à arriver au sommet Y, et l'on termine sur Z.

Si dans la clause C_1, ou n'importe laquelle des suivantes, seulement un littéral est faux, disons L_1, il est nécessaire que π passe toujours par tous les sommets pleins du triangle : π arrive sur b, parcourt le côté L_1 et arrive sur a comme avant, mais franchit une arête distinguée pour sauter directement à c.

De même, si aucun littéral de C_1 n'est faux, π passe de b à a puis à c en franchissant deux arêtes distinguées.

Lorsque tous les triangles de toutes les clauses ont été parcourus, π va vers Y via une arête distinguée, puis vers Z. Il est maintenant clair, en utilisant le fait A, que π passe exactement une fois par chaque sommet de G. □

Montrons donc le théorème 4.5. Par les faits B et C, S est satisfiable si et seulement si G a un chemin hamiltonien. La réduction, de plus, est trivialement faisable en temps polynomial. □

▷ **Exercice 4.1**

Déduire du théorème 4.5 que le langage HAMILTONIAN CYCLE :
ENTRÉE : un graphe $G = (V, E)$ non orienté ;
QUESTION : existe-t-il un cycle hamiltonien π dans G ?
est **NP**-complet.

▷ **Exercice 4.2**

Montrer que le langage DIRECTED HAMILTONIAN PATH :
ENTRÉE : un graphe $G = (V, E)$ orienté ;
QUESTION : existe-t-il un chemin hamiltonien π dans G ?
est lui aussi **NP**-complet. (Un *chemin* dans un graphe orienté $G = (V, E)$ est une suite de sommets $u_0, u_1, \ldots, u_n$ avec $(u_{i-1}, u_i) \in E$ pour tout i, $1 \leq i \leq n$. Il est *hamiltonien* s'il passe par chaque sommet de G *exactement* une fois, autrement dit si tous les u_i, $0 \leq i \leq n$, sont distincts deux à deux d'une part, et tout sommet de V est un u_i pour un certain i.)

▷ **Exercice 4.3**

Montrer que HAMILTONIAN CIRCUIT :
ENTRÉE : un graphe $G = (V, E)$ orienté ;
QUESTION : existe-t-il un circuit hamiltonien π dans G ?
est **NP**-complet. (Un chemin $u_0, u_1, \ldots, u_n$ est un *circuit* si et seulement si $u_n = u_0$. C'est un *circuit simple* si et seulement si pour tous i, j, $1 \leq i < j \leq n$, on a $u_i \neq u_j$, et un *circuit hamiltonien* si et seulement s'il est simple et passe par tous les sommets du graphe.)

▷ **Exercice 4.4**

Le problème du voyageur de commerce, TSP (ou TRAVELING SALESMAN PROBLEM), est :
ENTRÉE : une matrice $D = (d_{ij})_{1 \leq i,j \leq n}$ de valeurs $d_{ij} = d_{ji} \in \mathbb{N}$, avec $d_{ii} = 0$ pour tout i (où d_{ij} est écrit en binaire), et un *budget* $k \in \mathbb{N}$ écrit en binaire.
QUESTION : existe-t-il une permutation π de $\{1, \ldots, n\}$ dont le *coût* $c(\pi) = \sum_{i=1}^{n} d_{\pi(i),\pi(i \bmod n+1)}$ soit d'au plus k ?
L'intuition est que l'on dispose d'une carte indiquant des villes numérotées de 1 à n, et l'on sait que la distance de i à j vaut d_{ij}. La question est de trouver un trajet passant une fois et une seule par chaque ville pour finir par retourner à la ville de départ qui minimise la distance totale parcourue.

Montrer que TSP est **NP**-complet.

La morale des exercices qui suivent montrent qu'il se peut que deux langages aient des définitions très similaires, et que pourtant l'un soit **NP**-complet et l'autre dans **P**.

▷ **Exercice 4.5**

Un cycle dans un graphe non orienté est *eulérien* s'il passe exactement une fois par chaque arête (et non plus par chaque sommet, comme pour les cycles hamiltoniens). On considère le langage EULER CYCLE :
ENTRÉE : un graphe non orienté $G = (V, E)$;
QUESTION : G a-t-il un cycle eulérien ?
Montrer que EULER CYCLE $\in \mathbf{P}$.

On pourra procéder comme suit. Pour chaque chemin π dans G, notons $|\pi|_e$ le nombre de fois où π passe par l'arête e, $in_v(\pi)$ le nombre de fois où π arrive sur le sommet v, et $ex_v(\pi)$ le nombre de fois où π part de v. Le *degré* deg_v d'un sommet v dans G est le nombre d'arêtes e telles que $v \in e$. Montrer que :

1. Si π est un cycle, alors pour tout sommet v de G, $in_v(\pi) = ex_v(\pi)$.
2. Si G a un cycle eulérien, alors G vérifie la *loi de Kirchoff* : deg_v est pair pour tout sommet v de G.
3. Disons que G est *connexe* si et seulement si, pour tous sommets v, v' de G, il existe un chemin de v à v'. Un *sommet isolé* de G est un sommet de degré nul. Montrer que si G a un cycle eulérien, et si G n'a pas de sommet isolé, alors G est connexe.
4. Soit G un graphe vérifiant la loi de Kirchoff. Si π est un chemin dans G tel que $|\pi|_e \leq 1$ pour toute arête e de G, et qui est de longueur maximale parmi ceux vérifiant cette propriété, alors π est un cycle.
5. Soit G un graphe connexe vérifiant la loi de Kirchoff. Si π est un chemin comme au point 4, et si de plus $|\pi|_e = 0$ pour une arête e, alors il existe un sommet v par lequel passe π, et un cycle π' de v à v passant par e, et qui ne passe que par des arêtes par lesquelles π ne passe pas.
6. Soit G un graphe connexe vérifiant la loi de Kirchoff. Si π est un chemin comme au point 4, alors π est un cycle eulérien.

5 Logique propositionnelle intuitionniste

La logique (propositionnelle) intuitionniste est un raffinement de la logique classique, que l'on peut définir de différentes façons. L'une des plus simples est d'en décrire un système de déduction naturelle : le système **NJ** de déduction naturelle pour la *logique propositionnelle intuitionniste* est obtenue en enlevant la règle $(\neg\neg E)$ du système **NK**. On avait déjà remarqué que la règle $(\neg\neg E)$ était d'une forme différente des autres, qui (à part l'axiome) introduisaient ou éliminaient *un* connecteur logique.

5.1 Intuitionnisme, réalisabilité et déduction naturelle, le système NJ

Tout jugement dérivable en **NJ** est aussi dérivable en **NK**, et il existe des jugements dérivables en **NK** mais pas en **NJ**. Pour le démontrer, et en même temps justifier l'intérêt de la logique intuitionniste, nous en définissons sa sémantique dite par *réalisabilité*, due à S.C. Kleene en 1945. La logique intuitionniste elle-même a été définie par L.E.J. Brouwer dès les années 1910. Notre "réalisabilité" est en fait une variante de celle de Kleene, qui mentionne explicitement la vérité au sens de la logique classique ; la notion de réalisabilité a de nombreuses variantes.

On fixe ici une injection $\langle \_, \_ \rangle : \mathbb{N}^2 \to \mathbb{N}$ calculable, par exemple celle que nous avons utilisée lors de la démonstration de la proposition 2.11. On fixe aussi une injection de $\mathbb{N} \uplus \mathbb{N}$ dans $\mathbb{N}$; ceci revient à fixer deux fonctions ι_1 et ι_2 de $\mathbb{N}$ dans $\mathbb{N}$ telles que pour tout entier

$$\frac{}{\Gamma, F \vdash F}\,(Ax) \qquad \frac{}{\Gamma \vdash \top}\,(\top I) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash G}\,(\bot E)$$

$$\frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \wedge F_2}\,(\wedge I) \qquad \frac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_1}\,(\wedge E_1) \qquad \frac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_2}\,(\wedge E_2)$$

$$\frac{\Gamma \vdash F_1}{\Gamma \vdash F_1 \vee F_2}\,(\vee I_1) \qquad \frac{\Gamma \vdash F_2}{\Gamma \vdash F_1 \vee F_2}\,(\vee I_2) \qquad \frac{\Gamma \vdash F_1 \vee F_2 \quad \Gamma, F_1 \vdash G \quad \Gamma, F_2 \vdash G}{\Gamma \vdash G}\,(\vee E)$$

$$\frac{\Gamma, F \vdash \bot}{\Gamma \vdash \neg F}\,(\neg I) \qquad \frac{\Gamma \vdash \neg F \quad \Gamma \vdash F}{\Gamma \vdash G}\,(\neg E)$$

$$\frac{\Gamma, F_1 \vdash F_2}{\Gamma \vdash F_1 \Rightarrow F_2}\,(\Rightarrow I) \qquad \frac{\Gamma \vdash F_1 \Rightarrow F_2 \quad \Gamma \vdash F_1}{\Gamma \vdash F_2}\,(\Rightarrow E)$$

FIG. 9 – Le système de déduction naturelle **NJ** pour la logique intuitionniste

n, il existe au plus un indice $i \in \{1, 2\}$ et un entier m tels que $n = \iota_i(m)$. On peut choisir $\iota_1(m) = 2m$, $\iota_2(m) = 2m + 1$. On fixe aussi un codage $< \mathcal{M} >$ de machines de Turing I/O $\mathcal{M}$, et l'on considère que ces machines calculent des fonctions partielles récursives, de $\mathbb{N}$ vers $\mathbb{N}$.

Définition 5.1 (Réalisabilité) *On définit la relation $\rho, n \Vdash F$ ("l'entier n* réalise *la formule F dans l'environnement ρ") par récurrence structurelle sur F :*

$$\begin{array}{lll}
\rho, n \Vdash A & ssi & \rho(A) = 1 \\
\rho, n \Vdash \top & & \textit{toujours} \\
\rho, n \Vdash \bot & & \textit{jamais} \\
\rho, n \Vdash F_1 \wedge F_2 & ssi & \rho, n_1 \Vdash F_1 \textit{ et } \rho, n_2 \Vdash F_2, \textit{ où } n = \langle n_1, n_2 \rangle \\
\rho, n \Vdash F_1 \vee F_2 & ssi & \left\{ \begin{array}{l} n = \iota_1(m) \textit{ et } \rho, m \Vdash F_1 \\ \textit{ou} \quad n = \iota_2(m) \textit{ et } \rho, m \Vdash F_2 \end{array} \right. \textit{pour un certain } m \in \mathbb{N} \\
\rho, n \Vdash \neg F & ssi & \rho \models \neg F
\end{array}$$

$$\begin{array}{lll}
\rho, n \Vdash F_1 \Rightarrow F_2 & ssi & \rho \models F_1 \Rightarrow F_2 \textit{ et} \\
& & n = < \mathcal{M} > \textit{pour une machine I/O } \mathcal{M} \textit{ telle que} \\
& & \textit{pour tout } m \textit{ tel que } \rho, m \Vdash F_1, \textit{ } \mathcal{M} \textit{ termine sur l'entrée } m \textit{ et} \\
& & \textit{on a } \rho, \mathcal{M}(m) \Vdash F_2
\end{array}$$

Le cas le plus intéressant est celui de l'implication. Il exprime que l'entier n réalise $F_1 \Rightarrow F_2$ si et seulement si, non seulement $F_1 \Rightarrow F_2$ est vraie, mais encore n est le code d'une fonction partielle récursive qui transforme tout réaliseur de F_1 en un réaliseur de F_2. On note ici $\mathcal{M}(m)$ la valeur calculée par $\mathcal{M}$ sur l'entrée m, autrement dit on identifie la machine I/O

$$\frac{\Gamma, F_1, F_2, \Delta \vdash F}{\Gamma, F_2, F_1, \Delta \vdash F}\,(Ech) \quad \frac{\Gamma, F_1, F_1 \vdash F}{\Gamma, F_1 \vdash F}\,(Contr) \quad \frac{\Gamma \vdash F}{\Gamma, F_1 \vdash F}\,(Aff)$$

FIG. 10 – Échange, contraction, affaiblissement

$\mathcal{M}$ avec la fonction partielle récursive qu'elle calcule. La sémantique d'un réaliseur d'une implication est donc une fonction. De même, un réaliseur d'une conjonction est un couple de réaliseurs, et un réaliseur d'une disjonction est un réaliseur de l'une ou l'autre des formules en disjonction.

Nous avons dit plus haut que la logique intuitionniste était un raffinement de la logique classique. Ceci est matérialisé par :

Lemme 5.2 *Si $\rho, n \Vdash F$ alors $\rho \models F$. Si $\rho, n \vdash F$ est dérivable en* **NJ***, alors il l'est aussi en* **NK***.*

Démonstration. La deuxième partie est évidente. La première est une récurrence facile sur la structure de F, qui est particulièrement évidente lorsque F et atomique, une négation ou une implication. □

▷ **Exercice 5.1**

Montrer que $\rho, n \Vdash \neg F$ si et seulement si $\rho, n \Vdash F \Rightarrow \bot$. Montrer de même que $\Gamma \vdash \neg F$ est dérivable en **NJ** si et seulement si $\Gamma \vdash F \Rightarrow \bot$ est dérivable en **NJ**. Ce dernier résultat nous permet de dire que $\neg F$ et $F \Rightarrow \bot$ sont *intuitionnistiquement équivalentes*.

Pour montrer que **NJ** est correct pour la sémantique de réalisabilité, nous devons définir ce qu'est un jugement valide pour la réalisabilité. Intuitivement, un jugement $F_1, \ldots, F_k \vdash F$ est réalisable s'il existe une machine de Turing qui envoie tout k-uplet de réaliseurs pour F_1, $\ldots$, F_k respectivement, vers un réaliseur de F. Cependant, ceci n'a de sens que si $F_1, \ldots, F_k$ est une *liste*, pas un ensemble de formules.

Appelons *jugement rigide* (l'appellation n'est pas standard) toute expression de la forme $F_1, \ldots, F_k \vdash F$, où $F_1, \ldots, F_k$ est une liste de formules. On peut en particulier voir apparaître une même formule plusieurs fois à gauche du signe thèse $\vdash$. L'ordre des formules est lui aussi important.

Tout jugement rigide donne lieu à un jugement, en remplaçant $F_1, \ldots, F_k$ par l'ensemble $\{F_1, \ldots, F_k\}$. On peut définir un système de déduction naturelle analogue à **NJ**, mais qui travaille sur des jugements rigides : le système $\mathbf{NJ}_{\text{rig}}$ a pour règles celles du système **NJ** (mais où les jugements sont maintenant rigides), plus les règles d'échange, de contraction, et d'affaiblissement de la figure 10.

Il est clair que tout jugement rigide dérivable en $\mathbf{NJ}_{\text{rig}}$ donne lieu à un jugement (ordinaire) dérivable en **NJ** : toute dérivation de $\mathbf{NJ}_{\text{rig}}$ fournit une dérivation en **NJ** du jugement correspondant, en effaçant simplement les règles de la figure 10. Réciproquement, tout jugement $\Gamma \vdash F$ s'écrit sous forme d'un jugement rigide $F_1, \ldots, F_k$, en choisissant d'énumérer les formules de Γ dans un ordre donné (mais en fait quelconque). Toute dérivation en **NJ** de

$\Gamma \vdash F$ se traduit en une dérivation en $\mathbf{NJ}_{\text{rig}}$ de $F_1, \ldots, F_k \vdash F$. Toutes les règles sauf (Ax) restent inchangées. Pour (Ax), il s'agit de trouver une dérivation en $\mathbf{NJ}_{\text{rig}}$ d'un jugement de la forme $\Delta \vdash F$, où Δ est une liste de formules contenant au moins une copie de F. Par une succession d'instance de l'échange (Ech), on en ramène une à droite de la liste, puis on applique (Ax).

Disons qu'un jugement rigide $F_1, \ldots, F_k \vdash F$ est *réalisable* si et seulement s'il existe une machine I/O $\mathcal{M}$ (le *réaliseur*) qui envoie tout k-uplet de réaliseurs de F_1, …, F_k vers un réaliseur de F. Plus précisément, pour tout environnement ρ, pour tous entiers $n_1, \ldots, n_k$ tels que $\rho, n_1 \Vdash F_1, \ldots, \rho, n_k \Vdash F_k$, si la machine $\mathcal{M}$ termine sur le k-uplet $\langle n_1, \ldots, n_k \rangle$, alors elle retourne un entier n tel que $\rho, n \Vdash F$. (On note $\langle n_1, \ldots, n_k \rangle$ l'entier qui vaut 0 si $k = 0$, et $\langle \langle n_1, \ldots, n_{k-1} \rangle, n_k \rangle$ sinon.)

Proposition 5.3 (Correction) $\mathbf{NJ}_{\text{rig}}$ *(donc* $\mathbf{NJ}$*) est correct pour la sémantique de réalisabilité : tout jugement rigide* $\Gamma \vdash F$ *dérivable en* $\mathbf{NJ}_{\text{rig}}$ *est réalisable.*

Démonstration. Par récurrence sur une dérivation donnée π de $\Gamma \vdash F$ en $\mathbf{NJ}_{\text{rig}}$, on construit une machine I/O $\mathcal{M}_\pi$ adéquate. Si la dernière règle est (Ech) :

$$\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma, F_1, F_2, \Delta \vdash F\end{array}}{\Gamma, F_2, F_1, \Delta \vdash F}\,(Ech)$$

alors par récurrence on a une machine $\mathcal{M}_{\pi_1}$, et si $\Delta = F_3, \ldots, F_k$, on produit la machine $\mathcal{M}_\pi$ qui prend en entrée un entier n, calcule $m, n_2, n_1, n_3, n_4, \ldots, n_k$ tels que $n = \langle m, n_2, n_1, n_3, n_4, \ldots, n_k \rangle$, puis appelle $\mathcal{M}_{\pi_1}$ sur l'entrée $\langle m, n_1, n_2, n_3, n_4, \ldots, n_k \rangle$. Les autres règles sont traitées de façon similaire. Pour $(Contr)$, la machine $\mathcal{M}_\pi$ sur l'entrée $\langle m, n_1 \rangle$ appelle $\mathcal{M}_{\pi_1}$ sur $\langle m, n_1, n_1 \rangle$. Pour (Aff), $\mathcal{M}_\pi$ sur l'entrée $\langle m, n_1 \rangle$ appelle $\mathcal{M}_{\pi_1}$ sur m.

Passons aux règles logiques, c'est-à-dire celles de la figure 9. L'axiome (Ax) déduit $\Gamma, F \vdash F$, et est réalisé par la machine qui prend $\langle m, n \rangle$ en entrée et retourne n. Notons que cette machine réalise le jugement rigide $\Gamma, F \vdash F$: si $\Gamma = F_1, \ldots, F_k$, pour tout environnement ρ tel que $\rho, n_1 \Vdash F_1, \ldots, \rho, n_k \Vdash F_k$ et $\rho, n \Vdash F$, la machine appliquée à $\langle n_1, \ldots, n_k, n \rangle$ retourne n, et $\rho, n \Vdash F$ par hypothèse.

Nous traitons des règles de l'implication, les autres étant évidentes ou similaires. Pour $(\Rightarrow I)$, on a par hypothèse de récurrence une machine I/O $\mathcal{M}_{\pi_1}$ qui réalise le jugement rigide $\Gamma, F_1 \vdash F_2$. Il existe alors une machine I/O, qui sera notre machine $\mathcal{M}_\pi$, qui envoie tout entier m vers le code $< \lambda n_1 \cdot \mathcal{M}_{\pi_1} \langle m, n_1 \rangle >$ d'une machine I/O, que nous notons $\lambda n_1 \cdot \mathcal{M}_{\pi_1} \langle m, n_1 \rangle$, et qui sur l'entrée n_1, va calculer le couple $\langle m, n_1 \rangle$ et ensuite exécuter $\mathcal{M}_{\pi_1}$ sur l'entrée $\langle m, n_1 \rangle$. (Cette construction s'appelle le théorème s-m-n de Kleene — dans le cas $m = n = 1$.) Le fait que $\rho \models F_1 \Rightarrow F_2$ dès que toutes les formules de Γ sont réalisées dans l'environnement ρ est une conséquence facile du lemme 5.2.

Pour $(\Rightarrow E)$, on a par hypothèse de récurrence une machine I/O $\mathcal{M}_{\pi_1}$ qui réalise le jugement rigide $\Gamma \vdash F_1 \Rightarrow F_2$ et une autre, $\mathcal{M}_{\pi_2}$, qui réalise le jugement rigide $\Gamma \vdash F_1$. La machine $\mathcal{M}_\pi$ prend en entrée un entier n codant les réaliseurs des formules de Γ, calcule

$m = \mathcal{M}_{\pi_1}(n)$, $p = \mathcal{M}_{\pi_2}(n)$, et si ces deux calculs terminent, simule la machine $\mathcal{M}$ dont le code est m sur l'entrée p. En notation, $\mathcal{M}_\pi(m) = \mathcal{M}(\mathcal{M}_{\pi_2}(m))$, où $\mathcal{M} = \mathcal{M}_{\pi_1}(m)$. □

On note que, dans tout environnement ρ, si n réalise une disjonction $F_1 \vee F_2$, alors F_1 ou bien F_2 est réalisable. C'est la définition ! On en déduit :

Lemme 5.4 *Soit A une formule atomique. Le jugement $\vdash A \vee \neg A$ n'est pas dérivable en* **NJ**.

Démonstration. S'il l'était, il serait réalisable par la proposition 5.3. Il existerait donc une machine I/O qui envoie 0 (l'unique réaliseur du côté gauche du signe thèse) sur un entier n tel que pour tout environnement ρ, on a $\rho, n \Vdash A \vee \neg A$. Par définition de la réalisabilité des disjonctions, n est donc de la forme $\iota_1(m)$ ou $\iota_2(m)$. Le point crucial est que n est de cette forme, *indépendamment de* ρ : n est juste un entier donné ! Si $n = \iota_1(m)$, on en déduit que $\rho, m \Vdash A$ pour tout environnement ρ, ce qui est impossible lorsque $\rho(A) = 0$. Si $n = \iota_2(m)$, on a $\rho, m \Vdash \neg A$ pour tout environnement ρ, mais ceci implique $\rho \models \neg A$ par le lemme 5.2 ; c'est impossible lorsque $\rho(A) = 1$. □

Ceci distingue donc la logique intuitionniste de la logique classique. En effet (exercice 1.6), $\vdash F \vee \neg F$ est toujours dérivable en **NK**. On en déduit aussi que la règle $(\neg\neg E)$ n'est pas déductible du reste, **NJ**, des règles de **NK** :

Lemme 5.5 *La règle $(\neg\neg E)$ n'est pas admissible en* **NJ**. *Il existe une formule propositionnelle F telle que $\vdash \neg\neg F$ est démontrable en* **NJ** *mais pas $\vdash F$.*

Démonstration. On note que, par l'exercice 1.5, on peut démontrer $\neg(F_1 \vee F_2) \vdash \neg F_1$ et $\neg(F_1 \vee F_2) \vdash \neg F_2$ en **NJ**. En posant $F_1 = A$, $F_2 = \neg A$, on en déduit :

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ \neg(A \vee \neg A) \vdash \neg\neg A & \neg(A \vee \neg A) \vdash \neg A\end{array}}{\neg(A \vee \neg A) \vdash \bot}\,(\neg I)}{\vdash \neg\neg(A \vee \neg A)}\,(\neg I)$$

en **NJ**. Posons $F = A \vee \neg A$: le jugement $\vdash \neg\neg F$ est dérivable en **NJ**, mais pas le jugement $\vdash F$, par le lemme 5.4. □

▷ **Exercice 5.2**

Donner une dérivation de $\Gamma \vdash F \Rightarrow \neg\neg F$ en **NJ**. En déduire une de $\Gamma \vdash \neg\neg\neg F \Rightarrow \neg F$. En conséquence, montrer que la règle $(\neg\neg E)$ est admissible en **NJ** pour les formules *niées*, c'est-à-dire de la forme $\neg F$.

▷ **Exercice 5.3**

Montrer que $\Gamma \vdash \neg(F_1 \vee F_2) \Rightarrow \neg F_1 \wedge \neg F_2$ est dérivable en **NJ**.

$$\frac{}{\Gamma, A \vdash A}\,(Ax_{Atom}) \qquad \frac{\Gamma \vdash F \quad \Gamma', F \vdash G}{\Gamma, \Gamma' \vdash G}\,(Cut)$$

$$\frac{}{\Gamma \vdash \top}\,(\vdash \top) \qquad \frac{}{\Gamma, \bot \vdash G}\,(\bot \vdash)$$

$$\frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \wedge F_2}\,(\vdash \wedge) \qquad \frac{\Gamma, F_1, F_2 \vdash G}{\Gamma, F_1 \wedge F_2 \vdash G}\,(\wedge \vdash)$$

$$\frac{\Gamma \vdash F_1}{\Gamma \vdash F_1 \vee F_2}\,(\vdash \vee_1) \quad \frac{\Gamma \vdash F_2}{\Gamma \vdash F_1 \vee F_2}\,(\vdash \vee_2) \quad \frac{\Gamma, F_1 \vdash G \quad \Gamma, F_2 \vdash G}{\Gamma, F_1 \vee F_2 \vdash G}\,(\vee \vdash)$$

$$\frac{\Gamma, F \vdash \bot}{\Gamma \vdash \neg F}\,(\vdash \neg) \qquad \frac{\Gamma \vdash F}{\Gamma, \neg F \vdash G}\,(\neg \vdash)$$

$$\frac{\Gamma, F_1 \vdash F_2}{\Gamma \vdash F_1 \Rightarrow F_2}\,(\vdash \Rightarrow) \qquad \frac{\Gamma, F_2 \vdash G \quad \Gamma \vdash F_1}{\Gamma, F_1 \Rightarrow F_2 \vdash G}\,(\Rightarrow \vdash)$$

FIG. 11 – Le système de calcul des séquents **LJ** pour la logique propositionnelle intuitionniste

5.2 Calcul des séquents, le système LJ

On a vu que le calcul des séquents **LK** était plus pratique, en terme d'automatisation de la recherche de dérivations en tout cas, que le système de déduction naturelle **NK**. Il en est, en quelque sorte, de même, pour la logique intuitionniste. Il existe de nombreux calculs de séquents pour la logique intuitionniste, et certains, particulièrement optimisés pour la recherche de démonstrations, sont le système **LJT** de Dyckhoff [4] et surtout le système **SLJ** de Larchey-Wendling [5].

Le calcul des séquents **LJ** que nous présentons est l'un des premiers qui aient été considérés, et est décrit à la figure 11. On peut le voir en première approche comme une restriction de **LK** au cas de *séquents intuitionnistes*, qui sont des séquents avec exactement une formule à droite du signe thèse. (C'est-à-dire ce que nous appelions des jugements, pour les systèmes de déduction naturelle.) Certaines règles changent nécessairement, cependant. Par exemple, la règle $(\vdash \vee)$ ne peut pas être écrite de sorte à n'utiliser que des séquents intuitionnistes : même en forçant Δ à être vide, la prémisse en serait $\Gamma \vdash F_1, F_2$. On remplace cette règle par les deux règles $(\vdash \vee_1)$ et $(\vdash \vee_2)$ de la figure 11.

On définit de même le système $\mathbf{LJ}_{\text{rig}}$, formé à l'aide de jugements *rigides*. Ses règles sont celles de la figure 11, plus les règles d'échange, de contraction, et de contraction de la figure 10.

Proposition 5.6 *Tout jugement dérivable en* **LJ** *est dérivable en* **NJ**. *Il existe un algorithme*

en temps polynomial qui transforme toute dérivation en **LJ** *en une dérivation du même jugement en* **NJ**.

Démonstration. Par récurrence structurelle sur la dérivation π donnée. La règle (Ax_{Atom}) est un cas particulier de la règle (Ax) de **NJ**, et les règles droites (de la forme $(\vdash op)$, où op est un opérateur) sont exactement les règles d'introduction des connecteurs correspondants en **NJ**. Ces règles ne posent donc aucune problème particulier. Si π se termine par (Cut) (en prenant les notations de la figure 11), on produit la dérivation :

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, \Gamma', F \vdash G\end{array}}{\Gamma, \Gamma' \vdash F \Rightarrow G}\,(\Rightarrow I) \qquad \begin{array}{c}\vdots\\ \Gamma, \Gamma' \vdash F\end{array}}{\Gamma, \Gamma' \vdash G}\,(\Rightarrow E)$$

où les dérivations omises ($\vdots$) sont obtenues par hypothèse de récurrence et affaiblissement. (Il est facile de voir que l'affaiblissement est admissible en **NJ**.) Si π se termine par $(\bot \vdash)$, on produit :

$$\dfrac{\dfrac{}{\Gamma, \bot \vdash \bot}\,(Ax)}{\Gamma, \bot \vdash G}\,(\neg E)$$

Si π se termine par $(\wedge \vdash)$, on a par hypothèse une dérivation en **NJ** de $\Gamma, F_1, F_2 \vdash G$, donc une de $\Gamma, F_1 \wedge F_2, F_1, F_2 \vdash G$ par affaiblissement, d'où l'on déduit :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, F_1 \wedge F_2, F_1, F_2 \vdash G\end{array}}{\Gamma, F_1 \wedge F_2, F_1 \vdash F_2 \Rightarrow G}\,(\Rightarrow I) \quad \dfrac{\dfrac{}{\Gamma, F_1 \wedge F_2, F_1 \vdash F_1 \wedge F_2}\,(Ax)}{\Gamma, F_1 \wedge F_2, F_1 \vdash F_2}\,(\wedge E_2)}{\Gamma, F_1 \wedge F_2, F_1 \vdash G}\,(\Rightarrow E) \qquad \dfrac{\dfrac{}{\Gamma, F_1 \wedge F_2 \vdash F_1 \wedge F_2}\,(Ax)}{\Gamma, F_1 \wedge F_2 \vdash F_1}\,(\wedge E_1)}{\Gamma, F_1 \wedge F_2 \vdash G}\,(\Rightarrow E)$$

Si π se termine par $(\vee \vdash)$, on a par hypothèse deux dérivations en **NJ**, une de $\Gamma, F_1 \vdash G$ et une de $\Gamma, F_2 \vdash G$, d'où :

$$\dfrac{\dfrac{}{\Gamma, F_1 \vee F_2 \vdash F_1 \vee F_2}\,(Ax) \qquad \begin{array}{c}\vdots\\ \Gamma, F_1 \vdash G\end{array} \quad \begin{array}{c}\vdots\\ \Gamma, F_2 \vdash G\end{array}}{\Gamma, F_1 \vee F_2 \vdash G}\,(\vee E)$$

Si π se termine par $(\neg \vdash)$, on a par hypothèse une dérivation de $\Gamma \vdash F$ en **NJ**, d'où, en utilisant l'affaiblissement :

$$\dfrac{\dfrac{}{\Gamma, \neg F \vdash \neg F}\,(Ax) \qquad \begin{array}{c}\vdots\\ \Gamma, \neg F \vdash F\end{array}}{\Gamma, \neg F \vdash G}\,(\neg E)$$

Si π se termine, enfin, par $(\Rightarrow\vdash)$, on a par hypothèse de récurrence deux dérivations en **NJ** de $\Gamma, F_2 \vdash G$ et de $\Gamma \vdash F_1$, d'où, en utilisant l'affaiblissement sur chacune :

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, F_1 \Rightarrow F_2, F_2 \vdash G}}{\Gamma, F_1 \Rightarrow F_2 \vdash F_2 \Rightarrow G}(\Rightarrow I) \quad \dfrac{\dfrac{}{\Gamma, F_1 \Rightarrow F_2 \vdash F_1 \Rightarrow F_2}(Ax) \quad \dfrac{\vdots}{\Gamma, F_1 \Rightarrow F_2 \vdash F_1}}{\Gamma, F_1 \Rightarrow F_2 \vdash F_2}(\Rightarrow E)}{\Gamma, F_1 \Rightarrow F_2 \vdash G}(\Rightarrow E)$$

Finalement, il est facile de voir que cette traduction s'effectue en temps polynomial, en particulier parce qu'aucune dérivation n'est dupliquée. □

Proposition 5.7 *Tout jugement dérivable en* **NJ** *est dérivable en* **LJ**. *Il existe un algorithme en temps polynomial qui transforme toute dérivation en* **NJ** *en une dérivation du même jugement en* **LJ**.

Démonstration. Par récurrence structurelle sur la dérivation donnée π en **NJ**. Les règles d'introduction sont juste les règles droites. La règle (Ax) sur une formule axiome F se traduit, par récurrence structurelle sur F, comme à l'exercice 1.9. On examine le cas où F est une implication $F_1 \Rightarrow F_2$, à titre d'exemple :

$$\dfrac{\dfrac{\dfrac{\vdots \quad \vdots}{\Gamma, F_2, F_1 \vdash F_2 \quad \Gamma, F_1 \vdash F_1}(\Rightarrow\vdash)}{\Gamma, F_1 \Rightarrow F_2, F_1 \vdash F_2}(\vdash\Rightarrow)}{\Gamma, F_1 \Rightarrow F_2 \vdash F_1 \Rightarrow F_2}$$

où les deux dérivations manquantes ($\vdots$) sont obtenues par hypothèse de récurrence. Dans la suite, nous utiliserons donc (Ax) comme si c'était une règle de **LJ**.

Traitons des règles d'élimination. D'abord, $(\bot E)$:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \bot} \quad \dfrac{}{\bot \vdash G}(\bot\vdash)}{\Gamma \vdash G}(Cut)$$

Puis $(\wedge E_1)$:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash F_1 \wedge F_2} \quad \dfrac{\dfrac{}{F_1, F_2 \vdash F_1}(Ax)}{F_1 \wedge F_2 \vdash F_1}(\wedge\vdash)}{\Gamma \vdash F_1}(Cut)$$

On traite de même du cas $(\wedge E_2)$. Pour $(\vee E)$, on produit :

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash F_1 \vee F_2} \quad \dfrac{\dfrac{\vdots \quad \vdots}{\Gamma, F_1 \vdash G \quad \Gamma, F_2 \vdash G}}{\Gamma, F_1 \vee F_2 \vdash G}(\vee\vdash)}{\Gamma \vdash G}(Cut)$$

Pour $(\neg E)$,

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash F} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash \neg F} \qquad \dfrac{\dfrac{}{F \vdash F}\,(Ax)}{\neg F, F \vdash G}\,(\neg\vdash)}{\Gamma, F \vdash G}\,(Cut)}{\Gamma \vdash G}\,(Cut)$$

Finalement, pour $(\Rightarrow E)$,

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash F_1} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash F_1 \Rightarrow F_2} \qquad \dfrac{\dfrac{}{F_2, F_1 \Rightarrow F_2}\,(Ax) \quad \dfrac{}{F_1 \vdash F_1}\,(Ax)}{F_1 \Rightarrow F_2, F_1 \Rightarrow F_2}\,(\Rightarrow\vdash)}{\Gamma, F_1 \vdash F_2}\,(Cut)}{\Gamma \vdash F_2}\,(Cut)$$

Ceci définit un algorithme en temps polynomial, notamment parce que nous ne dupliquons jamais aucune dérivation. □

Donc **LJ** et **NJ** dérivent exactement les mêmes jugements. Sur le même principe qu'à la proposition 1.10, on peut démontrer que les coupures s'éliminent. En l'état de nos connaissances, seule la méthode syntaxique de la proposition 1.10 s'applique.

Théorème 5.8 (Élimination des coupures) *Il existe une machine de Turing qui, sur toute dérivation π d'un séquent en* **LJ***, termine et calcule une dérivation du même séquent en* $\mathbf{LJ}_{\mathrm{cf}}$*, c'est-à-dire du système* **LJ** *sans la coupure* (Cut)*.*

Démonstration. Les règles de transformation sont essentiellement les mêmes que pour **LK**, à la différence des cas de la disjonction, de la négation et de l'implication dans le cas où les formules principales des prémisses de la coupure considérée sont toutes les deux la formule de coupure. Ces cas changent peu par rapport à la figure 5, et nous les donnons à la figure 12. L'argument de terminaison est identique à celui de la proposition 1.10 ; la seule nouveauté est que nous devons vérifier, dans le cas de la négation, que la taille de $\bot$ (sur laquelle nous effectuons une coupure supplémentaire à droite de $\rightsquigarrow$, dans la troisième transformation de la figure 12) est strictement plus petite que celle de $\neg F$. □

On en déduit un résultat relativement étonnant, qui est typique de la logique intuitionniste, et clairement faux de la logique classique :

Proposition 5.9 *La disjonction intuitionniste est* constructive *: si $\vdash F \vee G$ est dérivable en* **NJ** *(resp.,* **LJ***), alors $\vdash F$ ou $\vdash G$ est dérivable en* **NJ** *(resp.,* **LJ***).*

Démonstration. Que ce soit en **NJ** ou en **LJ** revient au même. Par le théorème 5.8, si $\vdash F \vee G$ est dérivable en **LJ**, alors il l'est en $\mathbf{LJ}_{\mathrm{cf}}$. Mais les seules possibilités pour la dernière règle de la dérivation sont $(\vdash \vee_1)$ et $(\vdash \vee_2)$. □

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_3\\ \Gamma \vdash F_1\end{array}}{\Gamma \vdash F_1 \vee F_2}(\vdash \vee_1) \quad \dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma', F_1 \vdash G\end{array} \quad \begin{array}{c}\vdots\,\pi_2\\ \Gamma', F_2 \vdash G\end{array}}{\Gamma', F_1 \vee F_2 \vdash G}(\vee \vdash)}{\Gamma, \Gamma' \vdash G}(Cut) \quad \rightsquigarrow \quad \dfrac{\begin{array}{c}\vdots\,\pi_3\\ \Gamma \vdash F_1\end{array} \quad \begin{array}{c}\vdots\,\pi_1\\ \Gamma', F_1 \vdash G\end{array}}{\Gamma, \Gamma' \vdash G}(Cut)$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_3\\ \Gamma \vdash F_2\end{array}}{\Gamma \vdash F_1 \vee F_2}(\vdash \vee_2) \quad \dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma', F_1 \vdash G\end{array} \quad \begin{array}{c}\vdots\,\pi_2\\ \Gamma', F_2 \vdash G\end{array}}{\Gamma', F_1 \vee F_2 \vdash G}(\vee \vdash)}{\Gamma, \Gamma' \vdash G}(Cut) \quad \rightsquigarrow \quad \dfrac{\begin{array}{c}\vdots\,\pi_3\\ \Gamma \vdash F_2\end{array} \quad \begin{array}{c}\vdots\,\pi_2\\ \Gamma', F_2 \vdash G\end{array}}{\Gamma, \Gamma' \vdash G}(Cut)$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma, F \vdash \bot\end{array}}{\Gamma \vdash \neg F}(\vdash \neg) \quad \dfrac{\begin{array}{c}\vdots\,\pi_2\\ \Gamma' \vdash F\end{array}}{\Gamma', \neg F \vdash G}(\neg \vdash)}{\Gamma, \Gamma' \vdash G}(Cut) \quad \rightsquigarrow \quad \dfrac{\begin{array}{c}\vdots\,\pi_2\\ \Gamma' \vdash F\end{array} \quad \dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma, F \vdash \bot\end{array} \quad \dfrac{}{\bot \vdash G}(\bot \vdash)}{\Gamma, F \vdash G}(Cut)}{\Gamma, \Gamma' \vdash G}(Cut)$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_3\\ \Gamma, F_1 \vdash F_2\end{array}}{\Gamma \vdash F_1 \Rightarrow F_2}(\vdash \Rightarrow) \quad \dfrac{\begin{array}{c}\vdots\,\pi_2\\ \Gamma', F_2 \vdash G\end{array} \quad \begin{array}{c}\vdots\,\pi_1\\ \Gamma' \vdash F_1\end{array}}{\Gamma', F_1 \Rightarrow F_2 \vdash G}(\Rightarrow \vdash)}{\Gamma, \Gamma' \vdash G}(Cut) \quad \rightsquigarrow \quad \dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma' \vdash F_1\end{array} \quad \dfrac{\begin{array}{c}\vdots\,\pi_2\\ \Gamma', F_2 \vdash G\end{array} \quad \begin{array}{c}\vdots\,\pi_3\\ \Gamma, F_1 \vdash F_2\end{array}}{\Gamma, F_1, \Gamma' \vdash G}(Cut)}{\Gamma, \Gamma' \vdash G}(Cut)$$

FIG. 12 – Les principaux cas dans l'élimination des coupures en **LJ**

Ce résultat est faux en logique classique, c'est-à-dire en **NK** ou en **LK**, notamment parce que l'on peut toujours y déduire $\vdash F \vee \neg F$ (exercice 1.6), mais en général ni $\vdash F$ ni $\vdash \neg F$. Par exemple, si F est une formule atomique A, si A ni $\neg A$ n'est valide, donc ni $\vdash A$ ni $\vdash \neg A$ n'est dérivable par le lemme 1.5.

On en déduit le raffinement suivant du lemme 5.4.

Corollaire 5.10 *Si* $\vdash F$ *et* $\vdash \neg F$ *ne sont pas dérivables en* **NJ** *(resp.,* **LJ***), alors* $\vdash F \vee \neg F$ *non plus.*

▷ **Exercice 5.4**
On cherche à généraliser la proposition 5.9. Les formules de Harrop sont définies par la grammaire :

$$H \quad ::= \quad A \mid F \Rightarrow H \mid H \wedge H$$

où A parcourt les formules atomiques, et F les formules arbitraires. Démontrer le *théorème de Harrop* : si Γ est un ensemble fini de formules de Harrop et $\Gamma \vdash F \vee G$ est démontrable en **LJ**, alors $\Gamma \vdash F$ ou $\Gamma \vdash G$ l'est déjà. Montrer que ceci échoue si Γ n'est pas un ensemble de formules de Harrop.

L'exercice qui suit montre que l'on ne peut pas définir la disjonction intuitionniste en termes de négation et de conjonction.

▷ **Exercice 5.5**
Montrer que $\Gamma \vdash \neg(F_1 \wedge F_2) \Rightarrow \neg F_1 \vee \neg F_2$ n'est pas en général dérivable en **LJ**. (On considérera le cas où Γ est vide, et F_1 et F_2 sont atomiques. On fera bien attention que, contrairement à **LK**, les règles de **LJ** ne sont pas nécessairement inversibles.) Notons que l'implication réciproque est dérivable, par l'exercice 5.3.

▷ **Exercice 5.6**
Montrer de même que $\Gamma \vdash (F_1 \Rightarrow F_2) \Rightarrow (\neg F_1 \vee F_2)$ n'est pas en général dérivable en **LJ**, alors que $\Gamma \vdash (\neg F_1 \vee F_2) \Rightarrow (F_1 \Rightarrow F_2)$ l'est.

Ceci étant, la logique intuitionniste est beaucoup plus proche de la logique classique que ce dernier résultat ne laisse paraître :

▷ **Exercice 5.7**
On définit la traduction suivante de l'espace des formules vers lui-même, due à Gödel :

$$\begin{array}{rclrcl} A^* & = & \neg\neg A & & & \\ \top^* & = & \top & \bot^* & = & \bot \\ (F_1 \wedge F_2)^* & = & F_1^* \wedge F_2^* & (F_1 \vee F_2)^* & = & \neg(\neg F_1^* \wedge \neg F_2^*) \\ (\neg F)^* & = & \neg F^* & (F_1 \Rightarrow F_2)^* & = & F_1^* \Rightarrow F_2^* \end{array}$$

On notera que F est *classiquement équivalente* à F^*, c'est-à-dire que les environnements ρ qui satisfont F sont exactement les mêmes que ceux qui satisfont F^*. Démontrer que $\Delta \vdash \neg\neg F^* \Rightarrow F^*$ est dérivable en **NJ** pour toute formule propositionnelle F. En déduire que F est valide si et seulement si F^* est *intuitionnistement* démontrable : plus précisément, que le jugement $\Gamma \vdash F$ est valide si et seulement si $\Gamma^* \vdash F^*$ est dérivable en **NJ**, où Γ^* est l'ensemble des formules G^*, lorsque G parcourt Γ.

La traduction de Gödel fonctionne encore pour la logique du premier ordre, l'arithmétique et d'autres théories. L'exercice suivant fournit un résultat qui, lui, n'est valable qu'en logique propositionnelle.

▷ **Exercice 5.8**

Montrer que, si $\Gamma \vdash \Delta$ est dérivable en $\mathbf{LK}_{\mathrm{cf}}$, alors $\neg\neg\Gamma, \neg\Delta \vdash \bot$ est dérivable en **LJ**, où l'on note $\neg\Gamma$ l'ensemble des formules $\neg G$, $G \in \Gamma$. En déduire le *théorème de Glivenko* : F est (classiquement) valide si et seulement si $\vdash \neg\neg F$ est dérivable en **LJ** (resp., **NJ**).

5.3 Décider les formules intuitionnistes propositionnelles

L'élimination des coupures nous permet aussi de démontrer la décidabilité de la logique intuitionniste :

Théorème 5.11 *Le problème INT-PROOF :*
ENTRÉE : un séquent intuitionniste propositionnel $\Gamma \vdash F$ *;*
QUESTION : est-il dérivable en **NJ** *(resp.,* **LJ***) ?*
est décidable.

Démonstration. Par le théorème 5.8, il suffit de chercher une dérivation π en $\mathbf{LJ}_{\mathrm{cf}}$. On remarque la propriété importante *de la sous-formule* : toute formule apparaissant dans un séquent intuitionniste dans la dérivation π est une sous-formule d'une formule de $\Gamma \vdash F$.

Il n'existe qu'un nombre fini (polynomial) de sous-formules de $\Gamma \vdash F$, et donc qu'un nombre fini (exponentiel) de séquents intuitionnistes formés de sous-formules de $\Gamma \vdash F$. On construit alors l'ensemble fini E de tous ces séquents intuitionnistes, et l'on marque tous ceux qui sont dérivables en $\mathbf{LJ}_{\mathrm{cf}}$: d'abord, tous ceux qui sont instances de (Ax_{Atom}), de $(\vdash \top)$ ou de $(\bot \vdash)$. Ensuite, pour tout jugement marqué qui est prémisse d'une règle à une prémisse dont la conclusion est dans E, on marque la conclusion ; pour tout couple de prémisses des règles binaires dont la conclusion est dans E, marquer la conclusion ; ceci tant qu'il reste des séquents intuitionnistes marquables et non marqués. Ceci termine car E est fini. Finalement, on accepte si et seulement si $\Gamma \vdash F$ lui-même est marqué. □

La procédure de la démonstration du théorème 5.11 peut sembler très inefficace. Elle est cependant pratiquable, à condition de ne pas lister les éléments de E, mais de fabriquer au vol les éléments de E que l'on marque, et d'utiliser un calcul des séquents légèrement différent, où l'axiome est de la forme $A \vdash A$ et non $\Gamma, A \vdash A, \Delta$ par exemple, et la règle $(\vdash \wedge)$ (parmi d'autres) est remplacée par une règle qui déduit $\Gamma, \Gamma' \vdash F_1 \wedge F_2, \Delta, \Delta'$ à partir de $\Gamma \vdash F_1, \Delta$ et $\Gamma' \vdash F_2, \Delta'$. C'est la *méthode inverse*, due à Maslov (1964) — inverse, car elle cherche une démonstration sans coupure en partant du haut, plutôt que du bas comme dans une méthode de tableaux.

Il n'y a rien de spécifique à l'intuitionnisme qui nous force à utiliser la méthode inverse, et elle fonctionne tout aussi bien pour la logique classique. De façon symétrique, on pourrait aussi définir des tableaux intuitionnistes, mais c'est un peu plus compliqué qu'en logique classique. D'abord, certaines règles, comme $(\vdash \vee_1)$, $(\vdash \vee_2)$, $(\Rightarrow\vdash)$, $(\neg \vdash)$, ne sont pas *inversibles*. Il est donc possible qu'il faille revenir sur un choix de règle fait. C'est clair pour $(\vdash \vee_1)$ et $(\vdash \vee_2)$. Pour la règle d'implication gauche, c'est plus subtil. Pour démontrer $A \Rightarrow B, A, C \Rightarrow D \vdash B$, si l'on utilise $(\Rightarrow\vdash)$ sur la formule $C \Rightarrow D$ avec $\Gamma = A \Rightarrow B, A$, il restera à dériver $A \Rightarrow B, A, D \vdash B$ et $A \Rightarrow B, A \vdash C$. Mais ce dernier

jugement n'est en général pas démontrable (prendre A vrai, B vrai, C faux). Il faut donc revenir en arrière, et utiliser $(\Rightarrow\vdash)$ sur $A \Rightarrow B$. Pour éviter d'avoir à revenir en arrière sur ce genre de règles, on peut à la place (et c'est en fait obligatoire pour conserver la complétude) prendre $\Gamma = A \Rightarrow B, A, C \Rightarrow D$. Mais l'on a alors un problème de *bouclage*. Par exemple, on peut tenter de démontrer $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ comme suit :

$$\dfrac{\dfrac{\dfrac{\vdots\,(\Rightarrow\vdash)}{(A\Rightarrow B)\Rightarrow B, B\vdash A} \quad \dfrac{\dfrac{\vdots\,(\vdash\Rightarrow)}{(A\Rightarrow B)\Rightarrow B, A, B\vdash B}(\vdash\Rightarrow)}{(A\Rightarrow B)\Rightarrow B, B\vdash A\Rightarrow B}(\Rightarrow\vdash)}{(A\Rightarrow B)\Rightarrow B, B\vdash A} \quad \dfrac{\dfrac{\vdots\,(\vdash\Rightarrow)}{(A\Rightarrow B)\Rightarrow B, A\vdash B}(\vdash\Rightarrow)}{(A\Rightarrow B)\Rightarrow B\vdash A\Rightarrow B}(\Rightarrow\vdash)}{\dfrac{(A\Rightarrow B)\Rightarrow B\vdash A}{\vdash((A\Rightarrow B)\Rightarrow B)\Rightarrow A}(\vdash\Rightarrow)}$$

où l'on voit notamment que les séquents intuitionnistes de la branche manquante à gauche de la dérivation se répètent indéfiniment. (En fait, $\vdash ((A \Rightarrow B) \Rightarrow B) \Rightarrow A$ n'est pas démontrable en **LJ**.) Ceci se corrige à l'aide de *tests de bouclage* : si l'on rencontre un séquent intuitionniste à dériver qui est déjà apparu plus bas, on échoue et on revient sur un choix antérieur. Les systèmes **LJT** [4] et **SLJ** [5] évitent ce problème (et d'autres, dans le cas de **SLJ**).

Dans tous les cas, la méthode inverse montre que le problème INT-PROOF de dérivabilité des séquents intuitionnistes se décide en temps exponentiel, c'est-à-dire majoré par l'exponentielle d'un polynôme. Il se trouve que INT-PROOF est nettement plus complexe que SAT : INT-PROOF est en effet **PSPACE**-complet, comme le problème QPF de l'exercice 3.14.o

5.4 Sémantique de Kripke

La sémantique de réalisabilité n'est pas une caractérisation correcte et complète de la dérivabilité en logique intuitionniste. Par exemple, la formule $\neg\neg A \Rightarrow A$ est réalisable. La machine de Turing I/O $\mathcal{M}$ qui à n associe n lui-même est telle que $\rho, < \mathcal{M} > \Vdash \neg\neg A \Rightarrow A$. Mais $\vdash \neg\neg A \Rightarrow A$ n'est pas dérivable en $\mathbf{LJ}_{\mathrm{cf}}$. En effet, s'il en existe une dérivation, elle est de la forme :

$$\dfrac{\dfrac{\dfrac{\vdots}{[\neg\neg A]\vdash\neg A}(\neg\vdash)}{\neg\neg A\vdash A}(\vdash\neg)}{\vdash\neg\neg A\Rightarrow A}$$

où la formule $\neg\neg A$ entre crochets est optionnelle. On ne peut pas démontrer $\vdash \neg A$, et réappliquer $(\neg \vdash)$ ne sert à rien. Formellement, une dérivation de taille minimale dérive nécessairement $\neg\neg A \vdash \neg A$ par la règle $(\vdash \neg)$, à partir de $\neg\neg A, A \vdash \bot$. Ceci ne peut être dérivé que grâce à $(\neg \vdash)$, à partir de $[\neg\neg A,]A \vdash \neg A$. Par le même raisonnement, ceci ne peut être dérivé qu'à partir de $\neg\neg A, A \vdash \bot \ldots$ mais ceci est une instance de bouclage... contredisant, formellement, la minimalité de la dérivation.

Il existe en revanche plusieurs sémantiques pour lesquelles les différents systèmes de logique intuitionniste sont correctes et complètes. Une des plus connues, et des plus utiles, est la *sémantique de Kripke*. Ne dénigrons cependant pas la notion de réalisabilité, qui a été l'une des plus fructueuses en logique. Les arguments de terminaison par réductibilité du cours de logique et informatique (second semestre) sont en fait des formes de réalisabilité.

Définition 5.12 (Kripke) *On appelle* univers $\mathcal{W}$ *tout ensemble muni d'un préordre* $\leq$ *(une relation réflexive et transitive). Les éléments de* $\mathcal{W}$ *sont appelés les* mondes w. *Un* $\mathcal{W}$-environnement *est une fonction* ρ *qui à chaque variable propositionnelle associe une partie close par le haut de* $\mathcal{W}$ *— autrement dit, pour tout* $w \in \rho(A)$, *si* $w \leq w'$ *alors* $w' \in \rho(A)$. *On définit la relation* $\rho, w \models F$ *par récurrence par :*

$$\begin{array}{rcl}
\rho, w \models A & \textit{ssi} & w \in \rho(A) \\
\rho, w \models \top & & \textit{toujours} \\
\rho, w \models \bot & & \textit{jamais} \\
\rho, w \models F_1 \wedge F_2 & \textit{ssi} & \rho, w \models F_1 \textit{ et } \rho, w \models F_2 \\
\rho, w \models F_1 \vee F_2 & \textit{ssi} & \rho, w \models F_1 \textit{ ou } \rho, w \models F_2 \\
\rho, w \models \neg F & \textit{ssi} & \textit{pour aucun } w' \in \mathcal{W} \textit{ tel que } w \leq w', \textit{ on n'a } \rho, w' \models F \\
\rho, w \models F_1 \Rightarrow F_2 & \textit{ssi} & \textit{pour tout } w' \in \mathcal{W} \textit{ tel que } w \leq w', \textit{ si } w', \rho \models F_1 \textit{ alors } w', \rho \models F_2
\end{array}$$

On peut expliquer intuitivement cette sémantique comme suit. Imaginons que les mondes soient des instants, et que $w \leq w'$ signifie "w' est un futur possible de w". On peut imaginer que les formules de la logique intuitionniste ne sont pas vraies ou fausses dans l'absolu comme en logique classique, mais vraies à certains instants, fausses à d'autres. La sémantique de Kripke exprime ce qu'un scientifique idéal saurait affirmer à chaque instant. On lira alors $\rho, w \models F$: "à l'instant w, on peut affirmer que F est vraie", ou bien "en w, on *sait* que F est vraie".

Une propriété fondamentale (le lemme 5.13) sera que si $\rho, w \models F$ et $w \leq w'$ alors $\rho, w' \models F$. Ceci exprime que si l'on sait que F est vraie à l'instant w, alors on le saura toujours à l'instant w' : notre scientifique idéal n'oublie rien.

En revanche, on peut apprendre de nouveaux faits du monde, par exemple découvrir la loi des gaz parfaits. Si A est la formule (atomique) qui exprime la loi des gaz parfaits, et qu'on la découvre à l'instant w mais pas avant, alors $\rho(A)$, l'ensemble des mondes où A sera vraie, sera juste $\{w' \in \mathcal{W} \mid w \leq w'\}$.

La sémantique de $\top$, $\bot$, $\wedge$, $\vee$, est relativement inintéressante. La sémantique de la négation $\neg F$ se ramène à celle de l'implication $F \Rightarrow \bot$. Le plus intéressant, c'est la sémantique de l'implication. Elle exprime que si l'on sait à l'instant w que F_1 implique F_2, ceci ne signifie pas juste que si F_1 est vrai maintenant (en w), alors F_2 aussi : ceci signifie que si jamais on apprend F_1 plus tard, on pourra immédiatement affirmer F_2. C'est exactement ce qui se passe en science. Pour le moment, je ne sais pas si SAT est en temps polynomial. Mais je sais que si SAT est en temps polynomial alors $\mathbf{P} = \mathbf{NP}$. Si dans un futur proche ou non, on découvre que SAT est en temps polynomial, alors on saura immédiatement que $\mathbf{P} = \mathbf{NP}$.

Lemme 5.13 *Pour toute formule propositionnelle F, si $\rho, w \models F$ et $w \leq w'$, alors $\rho, w' \models F$.*

Démonstration. Par récurrence structurelle sur F. Lorsque $F = A$ est atomique, c'est parce que $\rho(A)$ est clos par le haut. Le lemme est évident lorsque F est une négation ou une implication, ou $\top$ ou $\bot$. C'est une utilisation directe de l'hypothèse de récurrence dans les autres cas. □

Proposition 5.14 (Correction) *Les systèmes* **NJ** *et* **LJ** *sont corrects pour la sémantique de Kripke : tout jugement $\Gamma \vdash F$ dérivable dans l'un ou l'autre système est* intuitionnistiquement valide*, c'est-à-dire que dans tout univers $\mathcal{W}$, pour tout $\mathcal{W}$-environnement ρ, en tout monde $w \in \mathcal{W}$, si $\rho, w \models G$ pour tout $G \in \Gamma$, alors $\rho, w \models F$.*

Démonstration. Montrons-le pour **NJ**. Puisque **NJ** et **LJ** dérivent exactement les mêmes jugements, ceci démontrera la proposition. Nous traitons uniquement des cas des règles $(\Rightarrow I)$ et $(\Rightarrow E)$, les autres étant évidentes.

Pour $(\Rightarrow I)$, supposons $\rho, w \models G$ pour tout $G \in \Gamma$. Par hypothèse de récurrence, pour tout $w' \in \mathcal{W}$, si $\rho, w' \models G$ pour tout $G \in \Gamma$, et si $\rho, w' \models F_1$ alors $\rho, w' \models F_2$. Or, lorsque $w \leq w'$, on a effectivement $\rho, w' \models G$ pour tout $G \in \Gamma$, par le lemme 5.13. Donc $\rho, w' \models F_1$ implique $\rho, w' \models F_2$. Comme w' est arbitraire tel que $w \leq w'$, ceci signifie que $\rho, w \models F_1 \Rightarrow F_2$.

Pour $(\Rightarrow E)$, supposons de nouveau $\rho, w \models G$ pour tout $G \in \Gamma$. Par hypothèse de récurrence, $\rho, w \models F_1$ d'une part ; d'autre part, $\rho, w \models F_1 \Rightarrow F_2$, ce qui, en posant $w' = w$ dans la définition de $\models$ sur les implications, implique que si $\rho, w \models F_1$ alors $\rho, w \models F_2$. Donc, effectivement, $\rho, w \models F_2$. □

Regardons la sémantique de $\neg\neg F$: on a $\rho, w \models \neg\neg F$ si et seulement si pour aucun w' avec $w \leq w'$, on n'a $\rho, w' \models \neg F$, c'est-à-dire pour tout w' avec $w \leq w'$, il existe un w'' tel que $w' \leq w''$ tel que $\rho, w'' \models F$ — F finit toujours par devenir vrai. La formule $\neg\neg F \Rightarrow F$ exprime donc que, dans tout futur w, si F finit toujours par devenir vraie après w, alors F est déjà vraie en w. Ceci n'est clairement pas le cas en général. L'univers formé des deux mondes distincts w_1, w_2 avec $w_1 \leq w_2$, où A est vrai en w_2 mais pas en w_1 ne vérifie pas $\neg\neg A \Rightarrow A$ en w_1. Ceci explique notamment pourquoi $(\neg\neg E)$ n'est pas admissible en **NJ** (lemme 5.5).

Dans la suite, notons $\rho, w \models \Gamma$ si et seulement si $\rho, w \models G$ pour tout $G \in \Gamma$, et $\rho, w \models \Gamma \vdash F$ si et seulement si $\rho, w \models \Gamma$ implique $\rho, w \models F$.

Proposition 5.15 (Complétude) **NJ**, **LJ**, *et* $\mathbf{LJ}_{\mathrm{cf}}$ *sont complets pour la sémantique de Kripke : tout jugement intuitionnistiquement valide $\Gamma \vdash F$ est dérivable en $\mathbf{LJ}_{\mathrm{cf}}$.*

Plus précisément, si $\Gamma \vdash F$ n'est pas dérivable en **LJ***, alors il existe un univers fini $\mathcal{W}$, un monde $w \in \mathcal{W}$ et un $\mathcal{W}$-environnement tels que $\rho, w \not\models \Gamma \vdash F$.*

Cette proposition établit donc non seulement la complétude, mais aussi un résultat nouveau : la propriété de *modèle fini*.

Démonstration. Fixons un jugement $\Gamma \vdash F$, et soit E l'ensemble fini des sous-formules de F ou de formules de Γ, plus $\bot$. Appelons *formule signée* un couple formé d'un signe, $+$

ou $-$, et d'une formule F de E. On note une formule signée $+F$ ou $-F$ selon le cas. Si S est un ensemble de formules signées, notons $+S$ le sous-ensemble des formules F telles que $+F \in S$, et $-S$ celui des F telles que $-F \in S$. Disons qu'un ensemble S de formules signées contenant $+\top$ et $-\bot$ est *cohérent* si et seulement si on ne peut pas dériver $\Gamma \vdash F$ en **LJ** pour aucune $\Gamma \subseteq +S$ et $F \in -S$. Un ensemble S de formules signées est *cohérent maximal* si et seulement si S est cohérent et aucun sur-ensemble strict de S n'est cohérent.

On a : (a) pour tout ensemble cohérent S, pour tout $F \in E$, $+F$ et $-F$ ne sont pas toutes les deux dans S. Sinon, comme on peut démontrer $F \vdash F$ par (Ax), S ne serait pas cohérent.

Ensuite : (b) tout ensemble cohérent S est inclus dans un ensemble cohérent maximal. C'est évident : si S n'est pas maximal, il existe un ensemble S' cohérent plus gros, et l'on démontre (b) par récurrence sur la différence entre 2^{2n} et le cardinal de S, où n est le cardinal de E. (Tout ensemble cohérent est alors de cardinal au plus 2^{2n}.)

Posons $\mathcal{W}$ l'ensemble de tous les ensembles cohérents maximaux. Un monde w est donc un ensemble cohérent maximal.

On a : (c) pour tout $F \in E$, pour tout $w \in \mathcal{W}$, $+F$ ou $-F$ appartient à w. Supposons par contradiction que ni $+F$ ni $-F$ ne soit dans w. Posons $w_+ = w \cup \{+F\}$, $w_- = w \cup \{-F\}$. Si w_- est incohérent, c'est que l'on peut dériver $\Gamma \vdash G$ pour un certain $\Gamma \subseteq +w_-$ (donc $\Gamma \subseteq +w$) et $G \in -w_-$. Si G appartenait à $-w$, w serait incohérent. Donc $G = F$. Si w_+ est aussi incohérent, c'est que l'on peut dériver $\Gamma' \vdash F'$, où $F' \in -w$, et $\Gamma' \subseteq +w \cup \{F\}$. Quitte à utiliser une instance de la règle admissible d'affaiblissement, on peut supposer que Γ' contient F. Donc on peut dériver $\Gamma', F \vdash F'$ pour un certain $\Gamma' \subseteq +w$ et $-F' \in w$. De $\Gamma \vdash F$ et $\Gamma', F \vdash F'$ on déduit $\Gamma, \Gamma' \vdash F'$ par (Cut), ce qui contredit le fait que w soit cohérent.

Puis : (d) si $+F_1, \ldots, +F_n$ sont dans un monde w, $F \in E$, et $F_1, \ldots, F_n \vdash F$ est dérivable en **LJ**, alors $F \in w$. En effet, sinon par (c) $-F$ serait dans w, contredisant la cohérence de w.

Posons $w \leq w'$ si et seulement si, pour toute formule signée de la forme $+\neg F$ dans w, on a $+F \notin w'$ (donc $-F \in w'$) et, pour toute formule signée de la forme $+(F_1 \Rightarrow F_2)$ dans w, si $+F_1 \in w'$ alors $+F_2 \in w'$. En identifiant $\neg F$ à $F \Rightarrow \bot$, la première condition se ramène à la seconde. Pour simplifier l'argument dans la suite, nous ne considérerons pas les négations.

On vérifie d'abord que : (e) si $+F$ est dans un monde w et $w \leq w'$, alors $+F \in w'$. En effet, d'abord $F \vdash \top \Rightarrow F$ est dérivable en **LJ** par $(\vdash\Rightarrow)$ et (Ax), donc par (d), $\top \Rightarrow F$ est dans w. Comme $+\top$ est dans tout monde, en particulier dans w', on a $+F \in w'$ par définition de $\leq$.

Montrons que : (f) $w \leq w$ pour tout monde w. Supposons que $+(F_1 \Rightarrow F_2)$ et $+F_1$ soient dans w. Si $+F_2$ n'était pas dans w, alors $-F_2$ serait dans w. Mais alors la dérivation suivante contredirait la cohérence de w :

$$\dfrac{\dfrac{}{F_2, F_1 \vdash F_2}\,(Ax) \qquad \dfrac{}{F_1 \vdash F_1}\,(Ax)}{F_1 \Rightarrow F_2, F_1 \vdash F_2}\,(\Rightarrow\vdash)$$

Montrons ensuite que : (g) $\leq$ est transitive. Supposons $w \leq w'$ et $w' \leq w''$, $+(F_1 \Rightarrow F_2) \in$

w, $+F_1 \in w''$, et montrons que $+F_2 \in w''$. Par (e), $+(F_1 \Rightarrow F_2) \in w'$, donc par définition de $\leq$, puisque $w' \leq w''$, $+F_2 \in w''$.

Le résultat clé est : (h) si pour tout w' tel que $w \leq w'$ et w' contient $+F_1$, w' contient aussi $+F_2$, alors w contient $+(F_1 \Rightarrow F_2)$. Considérons l'ensemble $S = \{+G \mid +G \in w\} \cup \{+F_1, -F_2\}$. Si S était cohérent, par (b) il existerait un monde w' contenant S. Pour tout $+(F_1' \Rightarrow F_2') \in w$, $+(F_1' \Rightarrow F_2')$ est dans S, donc dans w'. Si $+F_1' \in w'$, comme $w' \leq w'$ par (f), on a aussi $+F_2' \in w'$. Ceci démontre que $w \leq w'$. Par hypothèse, si w' contient $+F_1$ il contient aussi $+F_2$. Mais comme w' contient $+F_1$ et $-F_2$, ceci contredirait (a). Donc S est incohérent. Ceci implique que l'on peut démontrer $\Gamma, F_1 \vdash F_2$, avec $\Gamma \subseteq +w$. (Noter que, si en fait F_1 n'est pas présent à gauche du signe thèse, on peut l'y rajouter par affaiblissement.) Par la règle $(\vdash\Rightarrow)$, on peut donc déduire $\Gamma \vdash F_1 \Rightarrow F_2$, donc $+(F_1 \Rightarrow F_2) \in w$ par (d). (Le cas de la négation s'obtient à l'aide de la règle $(\vdash \neg)$.)

Définissons l'environnement ρ qui à toute variable A associe l'ensemble des mondes w qui contiennent $+A$. On démontre maintenant par récurrence structurelle sur la formule $F \in E$ que : (i) pour tout monde w, $+F \in w$ si et seulement si $\rho, w \models F$. C'est clair lorsque F est une variable propositionnelle A, en utilisant (a) et (c). Lorsque $F = \top$, on rappelle que tout monde contient $+\top$ et que $\rho, w \models \top$. Lorsque $F = \bot$, on rappelle que tout monde contient $-\bot$, donc pas $+\bot$, et que $\rho, w \not\models \bot$. Lorsque F est une conjonction $F_1 \wedge F_2$, si $+(F_1 \wedge F_2) \in w$ alors $+F_1$ et $+F_2$ sont aussi dans w par (d) appliqué à la dérivation de $F_1 \wedge F_2 \vdash F_1$ (resp., $F_1 \wedge F_2 \vdash F_2$) obtenue par $(\wedge \vdash)$ et (Ax), donc $\rho, w \models F_1$ et $\rho, w \models F_2$ par hypothèse de récurrence ; réciproquement, si $\rho, w \models F_1 \wedge F_2$, alors $\rho, w \models F_1$ et $\rho, w \models F_2$, donc par hypothèse de récurrence $+F_1$ et $+F_2$ sont dans w, donc aussi $+(F_1 \wedge F_2)$, en utilisant (d), $(\vdash \wedge)$ et (Ax). On procède de même lorsque F est une disjonction. Lorsque F est une implication $F_1 \Rightarrow F_2$, si $+(F_1 \Rightarrow F_2) \in w$ alors pour tout w' tel que $w \leq w'$, si $\rho, w' \models F_1$ alors $+F_1 \in w'$ par hypothèse de récurrence, donc $+F_2 \in w'$ par définition de $\leq$, donc $\rho, w' \models F_2$; comme w' est arbitraire, $\rho, w \models F_1 \Rightarrow F_2$. Réciproquement, si $\rho, w \models F_1 \Rightarrow F_2$, alors pour tout w' tel que $w \leq w'$, si $+F_1 \in w'$ on a $\rho, w' \models F_1$ par hypothèse de récurrence donc $\rho, w' \models F_2$, donc $+F_2$ par hypothèse de récurrence ; comme w' est arbitraire, par (h) on obtient $+(F_1 \Rightarrow F_2) \in w$. Le cas de la négation est similaire.

Revenons à notre jugement $\Gamma \vdash F$, supposé non dérivable en **LJ**. Posons $S = \{+G \mid G \in \Gamma\} \cup \{-F\}$. S est cohérent par hypothèse : soit donc w un monde contenant S, en utilisant (b). Par (i), $\rho, w \models G$ pour toute formule $G \in \Gamma$, et $\rho, w \not\models F$, donc $\rho, w \not\models \Gamma \vdash F$. □

On notera à la lecture de la démonstration que $(\Rightarrow\vdash)$, resp. $(\neg \vdash)$, se traduit sous forme de la réflexivité de la relation $\leq$, et que les seules règles qui provoquent un saut d'un monde w à un monde w' ($w \leq w'$) sont $(\vdash\Rightarrow)$ et $(\vdash \neg)$.

Références

[1] Sanjeev Arora and Boaz Barak. Complexity theory : A modern approach, 2007. `http://www.cs.princeton.edu/theory/complexity/`.

[2] Stephen Cook. The complexity of theorem proving procedures. In *Proc. 3rd annual ACM symposium on Theory of computing (STOC'71)*, pages 151–158, 1971.

[3] Marcello D'Agostino. Are tableaux an improvement over truth-tables ? Cut-free proofs and bivalence. *Journal of Logic, Language and Information*, 1(3), 1992.

[4] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3) :795–807, 1992.

[5] Didier Galmiche and Dominique Larchey-Wendling. Structural sharing and efficient proof-search in propositional intuitionistic logic. In *Advances in Computing Science—ASIAN'99 : 5th Asian Computing Science Conference*, pages 101–112, Phuket, Thailand, December 1999. Springer Verlag LNCS 1742.

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

[7] Ian P. Gent and Toby Walsh. The search for satisfaction. In *?*, 1999.

[8] Donald Ervin Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[9] Richard Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1) :155–171, 1975.

[10] Thomas J. Schaefer. The complexity of satisfiability problems. In *Proc. 10th annual ACM symposium on Theory of computing (STOC'71)*, pages 216–226, 1978.

[11] Helmut Schwichtenberg. Proof theory : Some applications of cut-elimination. In J. Barwise, editor, *Handbook of Mathematical Logic*, chapter D.2, pages 867–895. North-Holland Publishing Company, 1977.