Master 1 Informatique

Année 2018-2019

Smullyan et le Coq

Auteur :
M. Maël Dumas

Encadrant :
M. David Delahaye

Table des matières

Introduction

Raymond Smullyan était un logicien et mathématicien, qui a notamment contribué à la logique du premier ordre et à la théorie de la calculabilité. Il est également connu pour avoir écrit des livres d'énigmes logiques et mathématiques, qui permettent de montrer au grand public des concepts avancés en logique. On lui doit notamment le paradoxe du buveur, tiré de son livre « What Is the Name of This Book ? », dont la preuve n'est pas triviale et est utilisée dans de nombreux cours de logique pour montrer la difficulté de faire certaines preuves en logique classique.

Le livre de Smullyan qui nous a intéressé dans le cadre de ce projet est le livre d'énigmes « Le livre qui rend fou ! » [1], en particulier les énigmes de la princesse et du tigre. L'objectif de ce projet a été de formaliser ces énigmes et de les résoudre en apportant les preuves nécessaires (à savoir qu'une solution est bien une solution). Pour ce faire, nous avons utilisé l'outil d'aide à la preuve Coq, qui permet de spécifier des problèmes mathématiques et de faire les preuves correspondantes.

Ensuite, dans un second temps, nous avons créé un générateur d'énigmes inspiré des énigmes de la princesse et du tigre.

Le code du générateur ainsi que les preuve en coq des énigmes sont sur le dépot Git du projet : https://github.com/maeldumas/smulcoq

1. La princesse et le tigre

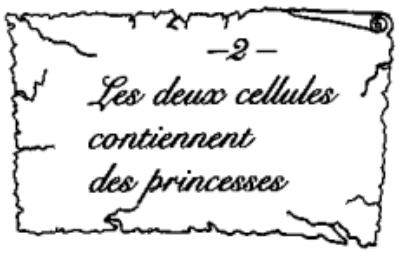
1.1 Présentation des énigmes

Dans l'histoire de la princesse ou du tigre un prisonnier doit choisir entre deux cellules, dont l'une cache une princesse, l'autre cache un tigre. S'il choisit la princesse, il doit l'épouser, mais s'il tombe sur le tigre, il est dévoré.
Un roi d'une contrée lointaine ayant entendu parler de cette histoire, décida de faire la même chose avec ses prisonniers. Mais il ne voulait pas que le résultat soit uniquement dû au hasard, ce ne serait pas drôle. Il décida donc d'afficher des inscriptions sur les portes des deux cellules. Les prisonniers ayant l'esprit logique pour trouver la solution à l'énigme seront graciés et pourront partir avec la princesse.

Le premier jour, le roi organisa trois épreuves. Il expliqua aux prisonniers que chacune des deux cellules contenait un tigre ou bien une princesse. Il peut y avoir deux princesses, deux tigres, ou une princesse et un tigre. L'énigme proposée au premier prisonnier est la suivante :

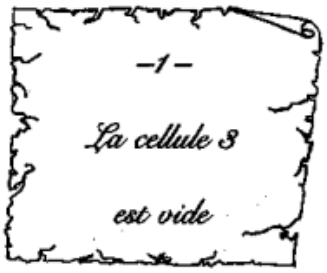
Le but est donc de trouver dans laquelle de ces cellules se trouve une princesse, mais en l'état il est difficile de déduire où se trouve une princesse. Le roi donne donc une information supplémentaire « Une des affiches dit la vérité, l'autre ment ». À partir de là, le prisonnier peut déduire où se trouve la princesse et avoir la vie sauve.
S'en suivent deux nouvelles énigmes aux règles similaires pour les prisonniers suivants lors du premier jour.

Puis le second jour, les énigmes de la veille ayant été résolues sans trop de difficultés, le roi décida de changer un peu les règles. Au lieu de dire si les affiches sont vraies ou non, il pose la règle suivante : si la première cellule contient une princesse, alors son affiche dit vrai, sinon elle ment et si la seconde cellule contient une princesse, alors son affiche ment, sinon elle dit vrai.
La quatrième énigme est :



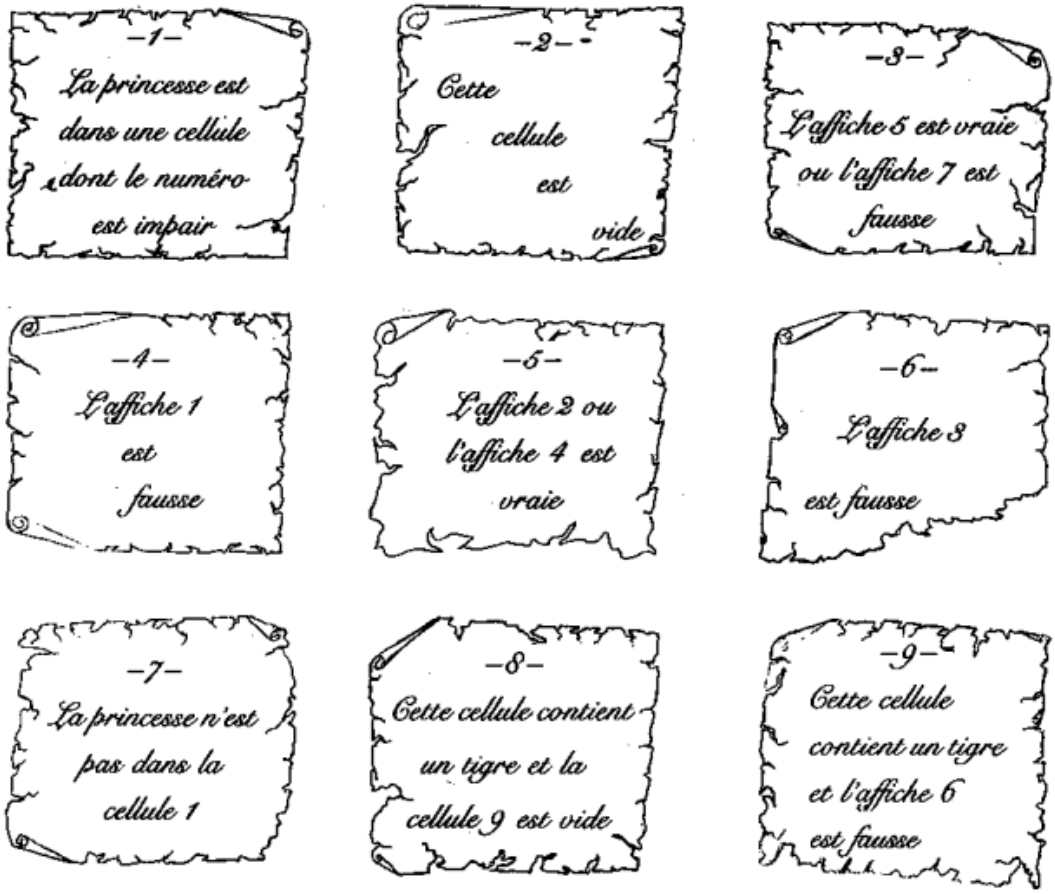
Le même jour, le roi propose quatre autres énigmes avec des règles similaires aux prisonniers suivants, elles furent toutes résolues sans trop de difficultés encore une fois.

Vint ensuite le troisième jour. Le roi, dépité d'avoir vu toutes ses énigmes de la veille résolues, décida de les compliquer encore une fois, et ajouta une troisième cellule. Il décida de ne mettre qu'une princesse et les deux autres cellules contiendraient un tigre. La neuvième et la dixième énigmes étaient similaires aux précédentes et furent encore une fois résolues.
Pour la onzième énigme, il décida de compliquer encore la chose, une des cellules qui contenait un tigre est désormais vide. Il donna en plus la règle suivante : la cellule qui contient la princesse dit la vérité, celle qui contient le tigre ment et celle qui est vide peut dire la vérité ou mentir. La onzième énigme a toujours pour objectif de trouver la cellule qui contient la princesse et les affiches sont les suivantes :



Cette énigme fut, une nouvelle fois, résolue.

Finalement, pour le quatrième et dernier jour, le roi, face à l'échec de ses énigmes précédentes, décida de sortir les grands moyens pour le dernier prisonnier. Il y avait désormais 9 cellules, et une seule princesse était cachée parmi elles, les autres étaient vides ou contenaient un tigre. Le roi reprit la même règle que pour la onzième énigme : la cellule qui contient la princesse dit la vérité, celle qui contient le tigre ment et celle qui est vide peut dire la vérité ou mentir. Les affiches sont les suivantes :



L'énigme étant insoluble, le prisonnier demanda au roi si la cellule 8 était vide ou non. Le roi, qui avait des remords lui répondit sincèrement, et le prisonnier put déduire où se trouvait la princesse.

La totalité des énoncés des énigmes du tigre et de la princesse peuvent être trouvés dans « Le livre qui rend fou ! ». Elles ont toutes été traitées lors de ce projet, mais ce rapport ne va s'attarder que sur la première, la quatrième, la onzième et la douzième.

1.2 Première résolution des énigmes

Nous définissons qu'une énigme de la princesse et du tigre est bien formée si à partir de ses hypothèses, il est possible de décider si une cellule contient une princesse ou non. Nous avons d'abord supposé que c'était le cas pour les énigmes posées par Raymond Smullyan, puis nous avons pu vérifier que c'était bien le cas.

Nous avons d'abord cherché les solutions en raisonnant "intuitivement". Par exemple pour la première énigme, on sait qu'une des deux affiches dit la vérité, l'autre ment. Il y a donc une princesse dans une cellule et un tigre dans l'autre, sinon les deux affiches mentiraient. Si la princesse est dans la cellule 1, alors, le tigre est dans la cellule 2 et les deux affiches disent la vérité, ce qui est exclu. Donc la princesse est dans la cellule 2 et le tigre dans la cellule 1, ainsi l'affiche 1 ment et l'affiche 2 dit la vérité.

Pour la quatrième énigme, on sait que si la cellule 1 contient une princesse alors l'affiche 1 dit la vérité, donc que les deux cellules contiennent une princesse. Mais si la cellule 2 contient

une princesse, alors son affiche ment, donc on sait qu'au moins une des deux cellules ne contient pas de princesse. On en déduit que la cellule 1 ne contient pas de princesse, elle contient donc un tigre. Une des cellules contient un tigre. Si la cellule 2 contient elle aussi un tigre, alors son affiche dirait la vérité, ce qui est exclu car les deux cellules ne contiendraient pas de tigre. En conséquence, la cellule 2 contient une princesse, l'affiche 2 ment, il doit y avoir au moins une cellule qui ne contient pas de princesse, ce qui est vérifié par la cellule 1 qui contient un tigre.

Dans la onzième énigme il y a trois cellules, une qui contient une princesse, une qui contient un tigre et une vide. On sait aussi que la cellule qui contient la princesse dit la vérité, celle qui contient un tigre ment et celle qui est vide peut dire la vérité comme mentir. Si la cellule 3 contient la princesse alors l'affiche 3 dit la vérité et cette cellule est vide, ce qui est exclu car elle contient la princesse. Si la cellule 2 contient la princesse, son affiche dit la vérité et la cellule 1 contient le tigre, la cellule 3 est donc vide, mais dans ce cas là l'affiche 1 dit la vérité, ce qui est exclu vu qu'elle contient le tigre. La princesse se trouve donc dans la cellule 1, l'affiche 1 dit la vérité donc la cellule 3 est vide, et la cellule 2 contient le tigre, et son affiche est bien fausse car la cellule 1 ne contient pas de tigre.

On peut observer que dans la onzième énigme, si il n'y a pas l'hypothèse qu'une cellule contient une princesse, une un tigre et une est vide, on ne peut pas trouver la solution. En effet, il est toujours possible d'avoir toutes les cellules vides, et dans ce cas là toutes les affiches vérifient bien la règle « si la cellule est vide alors son affiche peut dire la vérité ou mentir ». Il est donc impossible de déduire la présence d'une princesse dans une cellule sans cette hypothèse de répartition.

En suivant des raisonnements similaires, on a pu trouver les solutions aux neuf autres énigmes, c'est-à-dire savoir quelles cellules contiennent ou non une princesse. Mais pour prouver que nos solutions sont bien des solutions, nous devons formaliser les énigmes pour pouvoir les résoudre formellement.

1.3 Modélisation des énigmes

Afin de modéliser les énigmes, notre choix s'est porté sur la logique propositionnelle. En effet, de par la simplicité d'exprimer ce type d'énigme avec elle, elle est apparue comme étant la plus adaptée à notre problème. Il aurait été possible de modéliser les énigmes en logique du premier ordre, mais cela n'apportait pas suffisamment de généralisation pour que ce soit plus avantageux que la logique propositionnelle. Nous utilisons les notations suivantes :

- Variables propositionnelles :
  - PC_i : La cellule i contient une princesse.
  - TC_i : La cellule i contient un tigre.
  - VC_i : La cellule i est vide.
- A_i est l'affirmation de l'affiche i.
- HY est l'hypothèse du roi.
- HN est l'hypothèse exclusive, c'est-à-dire, qu'une cellule contient soit un tigre soit une princesse, mais pas les deux (ou qu'une cellule, si elle ne contient ni l'un ni l'autre est vide dans le cas des énigmes 11 et 12). HN s'exprime comme suit pour une énigme à n cellules :
  - sans cellule vide : $\bigwedge_{i=1}^{n} (PC_i \wedge \neg TC_i) \vee (\neg PC_i \wedge TC_i)$
  - avec cellule vide :
    $\bigwedge_{i=1}^{n} (PC_i \wedge \neg TC_i \wedge \neg VC_i) \vee (\neg PC_i \wedge TC_i \wedge \neg VC_i) \vee (\neg PC_i \wedge \neg TC_i \wedge VC_i)$

— HD est l'hypothèse de répartition, présente quand le roi précise le contenu des cellules (par exemple il y une princesse et deux tigres).

Remarque : Il aurait été possible, dans le cas où il n'y a pas de cellule vide, d'exprimer tout en fonction de PC_i, les TC_i devenant $\neg PC_i$. Dans le cas où il y a des cellules vides, il aurait été possible de remplacer VC_i par $(\neg PC_i \wedge \neg TC_i)$. Les résultats obtenus auraient été les mêmes avec ces choix.

Avec ces notations, la première énigme :
— L'affiche 1 : Il y a une princesse dans cette cellule et un tigre dans l'autre.
— L'affiche 2 : Il y a une princesse dans une cellule et un tigre dans une cellule.
— Hypothèse du roi : Il y a une affiche qui dit la vérité et l'autre ment.
Se modélise comme suit :

$$A_1 = PC_1 \wedge TC_2 \qquad A_2 = (PC_1 \wedge TC_2) \vee (PC_2 \wedge TC_1)$$
$$HY = (A_1 \wedge \neg A_2) \vee (\neg A_1 \wedge A_2)$$

On cherche à prouver que la solution (ici il n'y a pas de princesse dans la cellule 1 et il y en a une en cellule 2) est conséquence logique de l'hypothèse du roi et de l'hypothèse exclusive. Cela se formalise par :

$$HY, HN \models \neg PC_1 \wedge PC_2$$

Pour la quatrième énigme, l'hypothèse du roi dit que si la première cellule contient une princesse, alors son affiche dit vrai, sinon elle ment et si la seconde cellule contient une princesse, alors son affiche ment, sinon elle dit vrai. se modélise par :

$$HY = (PC_1 \rightarrow A_1) \wedge (TC_1 \rightarrow \neg A_1) \wedge (PC_2 \rightarrow \neg A_2) \wedge (TC_2 \rightarrow A_2)$$

Pour la onzième énigme l'hypothèse de répartition dit qu'il y a une princesse, un tigre et une cellule vide. L'hypothèse du roi dit qu'une cellule qui contient une princesse dit la vérité, une qui contient un tigre ment et une qui est vide peut dire la vérité ou mentir. On a donc :

$$HD = \bigvee_{i,j,k \in \{1,2,3\}, i \neq j \neq k} (PC_i \wedge TC_j \wedge VC_k)$$
$$HY = \bigwedge_{i=1}^{3} ((PC_i \rightarrow A_i) \wedge (TC_i \rightarrow \neg A_i) \wedge (VC_i \rightarrow (A_i \vee \neg A_i)))$$

La douzième énigme a la même hypothèse du roi que la onzième, mais avec neuf cellules au lieu de trois. L'hypothèse de répartition dit qu'il y a une seule princesse :

$$HD = \bigwedge_{i=1}^{9} (PC_i \wedge (\bigwedge_{j=1, j \neq i}^{9} \neg PC_j))$$

Il y a en plus la réponse à la question du prisonnier au roi à savoir, la cellule 8 est elle vide ou non. On peut voir facilement que la cellule 8 ne peut pas être vide sinon le prisonnier n'aurait pas réussi à trouver la solution, ainsi on doit montrer :

$$HY, HN, HD, \neg PC_8 \models \neg PC_1 \wedge \neg PC_2 \wedge \neg PC_3 \wedge \neg PC_4 \wedge \neg PC_5 \wedge \neg PC_6 \wedge PC_7 \wedge \neg PC_8 \wedge \neg PC_9$$

2. Le Coq

2.1 Présentation de Coq

Coq [2] est un assistant de preuve développé par l'équipe Inria πr^2. D'après le manuel de référence de Coq, il est conçu pour développer des preuves mathématiques et particulièrement pour écrire des spécifications formelles, des programmes et de vérifier que ces programmes sont corrects et respectent leur spécifications. Il met à disposition un langage de spécification nommé Gallina. Les termes de Gallina peuvent représenter des programmes mais aussi des propriétés de ces programmes et des preuves de ces propriétés. Utilisant l'isomorphisme de Curry-Howard, les programmes, les propriétés et les preuves sont formalisés dans le même langage, le calcul des constructions inductives qui est un λ-calcul richement typé. Toutes les décisions logiques en Coq sont des décisions de type. Le coeur du système Coq consiste en des algorithmes de vérification de type qui vérifient la validité des preuves, en d'autres mots, Coq vérifie qu'un programme est conforme à ses spécifications.

2.2 La modélisation dans Coq des énigmes

Pour la modélisation des énigmes avec Coq, nous avons fait la correspondance suivante avec la modélisation en logique propositionnelle :

— On déclare des variables propositionnelles XC_i avec "`Parameters` XCi : `Prop`.", Prop correspond au type des propositions dans Coq.
— On définit les affiches et l'hypothèse du roi avec : "`Definition` *X* : `Prop` := ..."
— On fait correspondre les symboles logiques à leur équivalent Coq (voir annexe).

Par exemple, l'énigme 1 se modélise comme suit (la correspondance des symboles logique et :

`Parameters` *PC1 PC2 TC1 TC2* : `Prop`.

`Definition` *A1* : `Prop` := *PC1*/*TC2*.
`Definition` *A2* : `Prop` := (*PC1* /*TC2*) $\bigvee$ (*PC2* /*TC1*).
`Definition` *HY1* : `Prop` := (*A1* /\~*A2*) $\bigvee$ (*A2* /\~*A1*).
`Definition` *HN2* : `Prop` := (*PC1* $\bigvee$ *TC1*) /\~(*PC1* /*TC1*) /\(*PC2* $\bigvee$ *TC2*) /\~(*PC2* /*TC2*).

Pour prouver les solutions que nous avons trouvées pour les énigmes, nous allons prouver que $H \rightarrow C$ est valide, où H sont les hypothèses et C les solutions. Dans le cas de l'énigme 1, cela revient à prouver le résultat suivant :

`Lemma` *S1* : *HY1* /*HN2* $\rightarrow$ ~*PC1* /*PC2*.

2.3 Une résolution détaillée de la première énigme

Nous allons détailler la résolution en Coq de la première énigme uniquement, les suivantes sont résolues grâce à des outils de Coq automatisant les preuves. Pour prouver dans Coq notre

notre solution, après avoir modéliser en Coq l'énigme comme décrit dans la section précédente, on utilise des tactiques qui permettent d'appliquer des règles de déduction sur le but que l'on doit prouver. On commence par déclarer toutes les données de notre énigme et le résultat à prouver, ce dernier sera notre but :

```
Coq < Lemma S1 : HY1 /\ HN2 -> ~PC1 /\ PC2.
1 subgoal

  ============================
  HY1 /\ HN2 -> ~ PC1 /\ PC2
```

On va ensuite déplier les définitions des hypothèses et des affiches et on introduit nos hypothèses :

```
S1 < unfold HY1, HN2, A1, A2.
S1 < intro.
1 subgoal

  H : ((PC1 /\ TC2) /\ ~ (PC1 /\ TC2 \/ PC2 /\ TC1) \/
       (PC1 /\ TC2 \/ PC2 /\ TC1) /\ ~ (PC1 /\ TC2)) /\
      (PC1 \/ TC1) /\ ~ (PC1 /\ TC1) /\ (PC2 \/ TC2) /\ ~ (PC2 /\ TC2)
  ============================
  ~ PC1 /\ PC2
```

Pour travailler sur l'hypothèse *H* on l'élimine avec les règles d'élimination correspondantes : si on a $A \wedge B$, on obtient les hypothèses A et B, si on a $A \vee B$ on obtient un sous-but avec A comme hypothèse et un autre avec B comme hypothèse. En appliquant une règle d'élimination sur *H* on obtient :

```
S1 < destruct H.
S1 < intro.
1 subgoal

  H : (PC1 /\ TC2) /\ ~ (PC1 /\ TC2 \/ PC2 /\ TC1) \/
      (PC1 /\ TC2 \/ PC2 /\ TC1) /\ ~ (PC1 /\ TC2)
  H0 : (PC1 \/ TC1) /\ ~ (PC1 /\ TC1) /\ (PC2 \/ TC2) /\ ~ (PC2 /\ TC2)
  ============================
  ~ PC1 /\ PC2
```

De la même manière on continue avec *H0*, et comme *H* est maintenant de la forme $A \vee B$ on a deux sous-but à prouver :

```
S1 < destruct H0; destruct H1; destruct H2; destruct H; destruct H.
2 subgoals

  H : PC1 /\ TC2
  H4 : ~ (PC1 /\ TC2 \/ PC2 /\ TC1)
  H0 : PC1 \/ TC1
  H1 : ~ (PC1 /\ TC1)
  H2 : PC2 \/ TC2
  H3 : ~ (PC2 /\ TC2)
  ============================
  ~ PC1 /\ PC2

subgoal 2 is:
 ~ PC1 /\ PC2
```

Dans le premier sous-but on observe qu'il y a une contradiction dans les hypothèses entre *H* et *H4*. Pour le montrer on utilise la règle d'élimination suivante sur *H4* : si on a l'hypothèse $\neg A$, alors on remplace le but par A. Le nouveau but est *PC1 /\TC2 \/ PC2 /\TC1*, on peut voir que la partie gauche de cette disjonction est l'hypothèse *H*, ce qui permet de conclure la preuve de ce sous-but :

```
S1 < destruct H4; left; assumption.
1 subgoal

  H : PC1 /\ TC2 \/ PC2 /\ TC1
  H4 : ~ (PC1 /\ TC2)
  H0 : PC1 \/ TC1
  H1 : ~ (PC1 /\ TC1)
  H2 : PC2 \/ TC2
  H3 : ~ (PC2 /\ TC2)
  ============================
  ~ PC1 /\ PC2
```

Il reste le second sous-but à prouver, on a deux nouveaux sous-buts en éliminant *H* :

```
S1 < destruct H.
2 subgoals

  H : PC1 /\ TC2
  H4 : ~ (PC1 /\ TC2)
  H0 : PC1 \/ TC1
  H1 : ~ (PC1 /\ TC1)
  H2 : PC2 \/ TC2
  H3 : ~ (PC2 /\ TC2)
  ============================
  ~ PC1 /\ PC2

subgoal 2 is:
 ~ PC1 /\ PC2
```

Pour ce sous-but, on voit que *H* et *H4* sont en contradiction, il reste donc un sous-but à prouver :

```
S1 < contradiction.
1 subgoal

  H : PC2 /\ TC1
  H4 : ~ (PC1 /\ TC2)
  H0 : PC1 \/ TC1
  H1 : ~ (PC1 /\ TC1)
  H2 : PC2 \/ TC2
  H3 : ~ (PC2 /\ TC2)
  ============================
  ~ PC1 /\ PC2
```

Ici on casse *H* et on prouve séparément *~PC1* et *PC2* :

```
S1 < destruct H; split.
2 subgoals

  H : PC2
  H5 : TC1
  H4 : ~ (PC1 /\ TC2)
  H0 : PC1 \/ TC1
  H1 : ~ (PC1 /\ TC1)
  H2 : PC2 \/ TC2
  H3 : ~ (PC2 /\ TC2)
  ============================
  ~ PC1

subgoal 2 is:
 PC2
```

Pour *~PC1*, vu que $\neg A$ est définit par $A \rightarrow \bot$, on peut introduire *PC1* en hypothèse. On élimine l'hypothèse *H1* : *~(PC1 /\TC2)* et on a *TC1* et *PC1* en hypothèse, ce qui nous permet de conclure

la preuve de ce but :

```
S1 < intro; destruct H1; split; assumption.
1 subgoal

  H : PC2
  H5 : TC1
  H4 : ~ (PC1 /\ TC2)
  H0 : PC1 \/ TC1
  H1 : ~ (PC1 /\ TC1)
  H2 : PC2 \/ TC2
  H3 : ~ (PC2 /\ TC2)
  ============================
  PC2

S1 < assumption.
No more subgoals.
```

PC2 étant dans les hypothèse, on peut finir la preuve du second sous-but, ce qui conclut par la même occasion notre preuve. Notre solution à l'énigme était donc bonne.

3. Résolution automatique des énigmes

3.1 La tactique tauto

Pour prouver nos solutions automatiquement, nous avons utilisé la tactique Coq tauto. C'est une procédure de décision pour le calcul propositionnel intuitionniste basé sur l'algorithme de calcul des séquents de Roy Dyckhoff [3]. Elle répond à n'importe quelle instance de proposition intuitionniste tautologique. L'implantation actuelle de tauto a été réalisée par David Delahaye.

Cette tactique pourra donc prouver toutes nos énigmes si elles sont bien formulées, c'est-à-dire que pour chaque énigme, il est possible de dire, pour chaque cellule si, oui ou non une princesse est dedans. La tactique fonctionne sur les onze premières énigmes, mais à partir de l'énigme 9, quand on a trois cellules, on peut constater un petit délai avant la validation de la preuve avec tauto. Pour l'énigme 12 la tactique tauto ne permet pas de prouver les solutions en un temps raisonnable (en plus de 2h de calcul sur nos ordinateurs personnels, la tactique n'avait toujours pas fourni de résultats, et avait une utilisation mémoire de 6 Go). La tactique tauto est une méthode de recherche de preuve en logique intuitionniste, c'est ce qui explique sa difficulté à prouver nos solutions quand le nombre de cellules augmente. En effet, plus il y aura d'affiches, plus l'arbre de preuve à explorer sera grand.

En conséquence nous avons dû choisir une autre méthode pour prouver nos solutions. La modélisation étant en logique propositionnelle, le choix du solveur SAT s'est imposé rapidement. Mais nous souhaitions toujours faire la preuve dans Coq.

3.2 Le plugin SMTcoq

SMTCoq [4] est un plugin Coq qui permet de vérifier les témoins (traces) de preuve de solveurs SAT et SMT externes, le solveur SAT utilisé par SMTCoq est ZChaff. SMTCoq met à disposition une nouvelle tactique, zchaff. Cette tactique va prendre un but, le transformer en forme normale conjonctive, appeler ZChaff sur celle-ci et ensuite vérifier la validité du témoin de preuve produit par ZChaff. Cette tactique permet donc de prouver les solutions des énigmes dans Coq. On notera néanmoins que les solveurs SAT utilisent la logique classique, alors que Coq utilise la logique intuitionniste.

La tactique zchaff doit travailler sur un terme de type bool, SMTCoq propose des tactiques permettant de transformer un terme de type Prop en type bool et inversement, mais nous n'avons pas réussi à les faire fonctionner. Nous avons donc fait une version de la modélisation Coq des énigmes utilisant le type bool au lieu de Prop (modification du type des paramètres et des définitions, et utilisation des connecteurs logiques correspondant au type bool).

Via SMTCoq nous avons pu améliorer le temps de calcul pour prouver nos solutions, en particulier, le temps pour la preuve de la douzième énigme a été réduit à moins d'une seconde.

4. Génération et résolution d'énigmes

4.1 Premières idées

Dans cette partie notre but était de générer des énigmes similaires à celles du tigre et la princesse. Nous avons décidé de considérer des énigmes avec les paramètres et caractéristiques suivants :

- N cellules avec une affiche associée à chacune. Une affiche peut décrire le contenu de sa cellule ou celle des autres, ou décrire la vérité d'autres affiches (mais elles ne doivent pas former de cycle, par exemple on exclut les situations où l'affiche 1 dit que l'affiche 2 est fausse, et inversement).
- Une hypothèse sur la vérité des affiches, l'hypothèse du roi. Il y a deux types d'hypothèses du roi que l'ont peut retrouver dans les énigmes de Smullyan :
  - Soit un certain nombre d'affiches disent la vérité.
  - Soit l'affiche dit ou non la vérité en fonction de son contenu.
- L'hypothèse exclusive.
- Optionnel : une hypothèse de répartition.

Il y a deux approches principales auxquelles nous avons pensé : partir d'une solution et construire une énigme qui a cette solution, ou générer une énigme aléatoirement et vérifier que cette énigme est bien formée. Dans un premier temps nous nous sommes intéressés à la seconde approche, en raison de la facilité de produire des énigmes qui auraient du sens et un intérêt pour un humain. La difficulté sera d'avoir des énigmes bien formées qui correspondent à nos caractéristiques (et dont on pourra définir le nombre de princesses).

4.2 La génération d'énigme aléatoire

L'objectif ici est d'avoir un générateur qui puisse proposer des énigmes bien formées compréhensibles et intéressantes pour un humain, et qui puisse proposer un maximum de variété parmi elles.

L'idée va être de choisir une hypothèse du roi et de générer aléatoirement N affiches selon des motifs prédéfinis. On va ensuite tester successivement pour chacune des cellules si on peut déduire des hypothèses la présence ou bien l'absence d'une princesse, alors on aura une énigme bien formée. Si dans l'une des cellules on ne peut déduire ni l'absence ni la présence d'une princesse, ou les deux à la fois, alors l'énigme ne sera pas bien formée, et on génère de nouvelles affiches.

Pour avoir des énigmes plus intéressantes, nous cherchons à avoir le moins de princesses possible dans nos énigmes, ceci impliquant qu'il y a moins de chance de trouver une cellule en contenant une princesse au hasard. Ainsi dans une énigme bien formée, si on peut déduire la présence d'un nombre de princesse dépassant une borne fixée, alors on recommence le processus et on génère de nouvelles affiches.

Utiliser un solveur SAT est la solution qui nous est apparue la plus adaptée et la plus rapide pour tester si les énigmes sont bien formées. Pour en utiliser un il est nécessaire de mettre sous forme normale conjonctive notre formule. L'outil utilisé pour faire cela est Limboole [5] qui permet de vérifier la validité de formules logiques de forme quelconque (mais nécessitant d'être bien parenthésée et utilisant des connecteurs logiques qui sont décrits dans l'annexe). Limboole utilise la transformation de Tseitin [6] pour transformer une formule en une forme normale conjonctive dans le format DIMACS [7] qui est un format standard utilisé par beaucoup de solveurs SAT. Il utilise ensuite un solveur SAT (PicoSAT ou Lingeling) pour vérifier la validité des formules.

4.2.1 Structure du générateur

Le générateur n'a pas été développé en Coq car le but n'était pas de le certifier, il n'était donc pas nécessaire d'utiliser un langage aussi contraignant (Coq requiert de faire des fonctions qui sont totales et qui terminent). Ainsi le générateur a été développé en C++ et toutes les formules logiques ont été traitées sous forme de chaînes de caractères. La gestion de l'aléatoire à été faite avec la fonction $rand()$ de la libraire $< cstdlib >$. Nous avons créé une classe $enigme$ pour représenter les énigmes, elle permet de :

- Générer des affiches selon certains paramètres (nombre de cellules, nombre minimum et maximum de princesses, possibilité d'avoir des cellules vides ou non, affiches pouvant mentionner d'autres affiches ou non)
- Vérifier la validité de l'énigme via Limboole.
- Écrire un fichier Coq qui définit l'énigme, sa solution et peut faire sa preuve avec SMTCoq.
- Écrire un fichier texte qui correspond à une interprétation en langage naturel de l'énigme.
- Prendre un fichier d'énigme en entrée et trouver sa solution. Le fichier doit respecter notre format décrit plus bas. Il est aussi possible d'écrire un fichier qui correspond à l'énigme dans ce format.

Nous avons créé deux outils pour utiliser cette classe, genigme et solvigme. Le premier sert à générer des énigmes en pouvant choisir les différents paramètres, le second permet de prendre une énigme en entrée et de trouver sa solution si elle est bien formée. La notice de ces outils est consultable en annexe.

4.2.2 Affiches et motifs

Pour générer les affiches, nous utilisons des motifs inspirés des affiches des énigmes de Smullyan, l'avantage d'utiliser des motifs va être la simplicité de les transposer en langage naturel. On peut les séparer en deux catégories, celles qui concernent uniquement ce que contiennent les cellules et celles qui concernent les autres affiches.

Pour la première catégorie, il y a six motifs :

- XC_i : La cellule i contient X.
- $\neg XC_i$: La cellule i ne contient pas X.
- $XC_i \wedge YC_j$: La cellule i contient X et la cellule j contient Y.
- $(\neg XC_i \wedge YC_j) \vee (XC_i \wedge \neg YC_j)$: La cellule i contient X et la cellule j ne contient pas Y, ou la cellule i ne contient pas X et la cellule j contient Y.
- $\bigwedge_{m=pk+q} XC_m$: Pour tout k, la cellule numérotée pk+q contient X (où p et q sont fixés).
- $\bigvee_{m=pk+q} XC_m$: Il existe un k tel la cellule numérotée pk+q contient X. (où p et q sont fixés)

Dans ces motifs, $XC_i \in \{VC_i, TC_i, VC_i\}$, $i, j \in \{1, \ldots, N\}$. À la création d'un motif, toutes les variables se voient attribuer un élément de leur domaine de façon équiprobable. Pour la génération d'affiches avec uniquement des motifs de la première catégorie, à chaque affiche est attribué aléatoirement un motif parmi les six possibles.

Pour la seconde catégorie, il y a deux motifs :
— T1 : $\neg^* A_i \diamond \neg^* XC_k$
— T2 : $(\neg^* A_i \diamond \neg^* A_j) \diamond \neg^* XC_k$
Ici $\neg^*$ désigne la présence ou non d'une négation, $\diamond$ un connecteur logique parmi $\{\wedge, \vee, \rightarrow, \leftrightarrow\}$ et k le numéro de la cellule associé à l'affiche.

Pour éviter de créer des cycles, l'organisation de nos affiches peut se voir comme une structure d'arbre binaire, que nous appelons arbre d'affiches. Pour générer les affiches avec un arbre d'affiches, nous utilisons la procédure suivante :

Données : T : l'arbre des affiches
début
  pour chaque *affiche A sur T* **faire**
    si *A est sur une feuille de T* **alors**
      A ← Motif de la catégorie 1 choisi aléatoirement
    fin
    si *A est sur un noeud interne de T et a un fils* **alors**
      A ← Motif T1
    fin
    si *A est sur un noeud interne de T et a deux fils* **alors**
      A ← Motif T2
    fin
  fin
fin

4.2.3 Vérification d'une énigme

Pour vérifier si une énigme est bien formée, notre algorithme est le suivant :

début
  $bien\_formée \leftarrow vrai; i \leftarrow 1;$
  tant que $bien\_formée$ ***et*** $i <= nombre\_cellules$ **faire**
    $p \leftarrow consequenceH(PCi);$
    $np \leftarrow consequenceH(!PCi);$
    si $p = np$ **alors**
      $bien\_formée \leftarrow faux;$
    fin
    $i \leftarrow i + 1;$
  fin
  retourner $bien\_formée$
fin

La méthode $consequenceH(C)$ renvoie vrai si $H \rightarrow C$ est valide, faux sinon, où H est l'hypothèse de l'énigme. Pour faire cette vérification cette méthode utilise Limboole. On remarque que quand l'on trouve une éngime on a trouvé la solution à l'énigme)

4.2.4 Choix pour le générateur genigme

Nous avons choisi de n'utiliser qu'une seule hypothèse du roi pour le générateur, celle de l'énigme 12. Il y aurait eu un problème d'ordre combinatoire avec une hypothèse du type « il y a tant d'affiches qui disent la vérité », en effet, si on veut qu'il y ait la moitié des affiches qui

disent la vérité, on aurait alors $\binom{N}{N/2}$ possibilités, ce qui fait que la modélisation de l'énigme en logique propositionnelle aurait été trop lourde. Il aurait été possible de trouver des variantes à l'hypothèse du roi de l'énigme 12, mais les motifs ont été choisis empiriquement pour trouver une énigme bien formée intéressante le plus rapidement possible en fonction de cette hypothèse. Changer d'hypothèse ne nous garantirait plus nécessairement d'avoir une énigme bien formée en un temps raisonnable.

Notre hypothèse du roi disant qu'une affiche d'une cellule vide peut dire la vérité ou mentir, il était nécessaire d'ajouter une hypothèse de répartition sur le nombre de princesses si il y a la possibilité d'avoir des cellules vides. Cela s'explique par les mêmes raisons qu'il est nécessaire d'avoir l'hypothèse de répartition pour résoudre la onzième énigme. Pour des raisons combinatoires, encore une fois, nous avons ajouté uniquement l'hypothèse qu'il y a exactement une princesse dans les cellules. Ainsi si on veut créer une énigme permettant d'avoir des cellules vides il ne peut y avoir qu'une princesse. Dans l'outil genigme que nous avons développé, le nombre de princesses a aussi été limité à un dans le cas où il n'y a pas de cellule vide. En effet, nous nous sommes aperçu qu'ajouter l'hypothèse de répartition permettait d'augmenter les chances d'obtenir une énigme à une princesse bien formée. De plus, le but du générateur d'énigme est de produire des énigmes avec le moins de princesses possible, c'est une ressource rare !

La probabilité d'avoir une énigme bien formée dépends du nombre de cellules, plus le nombre étant élevé, plus la probabilité est faible (voir 4.2.6 Performances du générateur). Pour cette raison, nous avons mis en place un arrêt du programme au bout de 60 secondes si aucune énigme bien formée n'a été produite. L'utilisateur peut modifier ce délai (voir annexe).

4.2.5 Exemple d'énigmes générées

Énigme à 7 cellules, une princesse, sans cellules vides et l'hypothèse du roi de l'énigme 12, générée avec la commande "./genigme 7" :

— Affiche 1 : La cellule 5 contient un tigre et la cellule 3 contient une princesse.
— Affiche 2 : La cellule 2 contient une princesse et la cellule 1 ne contient pas de princesse, ou la cellule 2 ne contient pas de princesse et la cellule 1 contient une princesse.
— Affiche 3 : La cellule 3 contient une princesse et la cellule 2 contient un tigre.
— Affiche 4 : Les cellules numérotées 3k+2 contiennent un tigre.
— Affiche 5 : La cellule 4 ne contient pas de princesse.
— Affiche 6 : La cellule 4 ne contient pas de tigre.
— Affiche 7 : La cellule 3 ne contient pas de tigre.

Énigme à 9 cellules, une princesse, avec cellules vides et l'hypothèse du roi de l'énigme 12, générée avec la commande "./genigme 9 -vide -af" :

— Affiche 1 : Les cellules numérotées 2k+0 contiennent un tigre.
— Affiche 2 : Les cellules numérotées 4k+1 contiennent un tigre.
— Affiche 3 : La cellule 6 est vide et la cellule 8 contient une princesse.
— Affiche 4 : L'affiche 1 dit la vérité et cette cellule est vide.
— Affiche 5 : L'affiche 2 ment et cette cellule ne contient pas de princesse.
— Affiche 6 : L'affiche 4 dit la vérité ou l'affiche 3 ment, et cette cellule n'est pas vide.
— Affiche 7 : L'affiche 6 ment équivaut à l'affiche 5 ment, et cette cellule n'est pas vide.
— Affiche 8 : L'affiche 7 ment et cette cellule contient un tigre.
— Affiche 9 : L'affiche 8 ment et cette cellule contient un tigre.

4.2.6 Performances du générateur

Nous avons mesuré le nombre d'énigme bien formée pour des énigmes à une princesse allant de 2 à 25 cellules dans les quatre configurations suivantes :

— NV-NT : sans cellules vides et sans arbre d'affiches.
— NV-T : sans cellules vides et avec arbre d'affiches.
— V-NT : avec cellules vides et sans arbre d'affiches.
— V-T : avec cellules vides et avec arbre d'affiches.

Pour chaque nombre de cellules et chaque configuration, 20 000 énigmes ont été générées et testées. Le premier tableau montre le nombre et le pourcentage d'énigmes bien formée pour chacune des configurations. Le second montre le temps moyen pour générer une énigme et vérifier si elle est bien formée pour chacune des configurations. Le temps a été mesuré avec la librairie C++ $std :: chrono :: system\_clock$.

Nombre de cellules	**NV-NT**	**NV-T**	**V-NT**	**V-T**
2	3671 (18.355%)	5587 (27.935%)	9218 (46.09%)	9945 (49.725%)
3	6303 (31.515%)	5968 (29.84%)	7539 (37.695%)	8326 (41.63%)
4	5435 (27.175%)	5446 (27.23%)	4656 (23.28%)	6357 (31.785%)
5	4134 (20.67%)	4472 (22.36%)	2507 (12.535%)	4318 (21.59%)
6	2927 (14.635%)	3380 (16.9%)	1190 (5.95%)	2922 (14.61%)
7	2059 (10.295%)	2729 (13.645%)	567 (2.835%)	2055 (10.275%)
8	1453 (7.265%)	2116 (10.58%)	257 (1.285%)	1292 (6.46%)
9	934 (4.67%)	1490 (7.45%)	126 (0.63%)	828 (4.14%)
10	604 (3.02%)	1143 (5.715%)	53(0.265%)	565 (2.825%)
11	341 (1.705%)	793 (3.965%)	22 (0.11%)	323 (1.615%)
12	238 (1.19%)	593 (2.965%)	8 (0.04%)	236 (1.18%)
13	186 (0.93%)	403 (2.015%)	4 (0.02%)	156 (0.78%)
14	101 (0.505%)	329 (1.645%)	1 (0.005%)	81 (0.405%)
15	68 (0.34%)	237 (1.185%)	1 (0.005%)	60 (0.3%)
16	32 (0.16%)	164 (0.82%)	0 (0%)	37 (0.185%)
17	24 (0.12%)	111 (0.555%)	0 (0%)	20 (0.1%)
18	22 (0.11%)	76 (0.38%)	0 (0%)	15 (0.075%)
19	9 (0.045%)	52 (0.26%)	0 (0%)	9 (0.045%)
20	7 (0.035%)	63 (0.315%)	0 (0%)	3 (0.015%)
21	2 (0.01%)	38 (0.19%)	0 (0%)	2 (0.01%)
22	2 (0.01%)	29 (0.145%)	0 (0%)	0 (0%)
23	1 (0.005%)	16 (0.08%)	0 (0%)	1 (0.005%)
24	5 (0.025%)	7 (0.035%)	0 (0%)	0 (0%)
25	1 (0.005%)	7 (0.035%)	0 (0%)	0 (0%)

TABLE 4.1 – Nombre (pourcentage) d'énigmes bien formées parmi les 20 000 générées pour chaque configuration.

Nombre de cellules	NV-NT	NV-T	V-NT	V-T
2	4.9434ms	5.29145ms	6.2701ms	6.3864ms
3	6.98835ms	6.85595ms	8.76765ms	8.98865ms
4	8.25385ms	8.3007ms	10.0691ms	10.9982ms
5	8.50175ms	8.8795ms	10.9142ms	11.7084ms
6	8.03665ms	8.86345ms	11.2084ms	12.6271ms
7	7.88265ms	9.1193ms	11.9898ms	13.592ms
8	7.5667ms	9.12345ms	12.651ms	13.6656ms
9	6.8996ms	8.36545ms	13.3726ms	14.4322ms
10	6.76135ms	8.36145ms	14.2541ms	15.0772ms
11	6.1649ms	7.6956ms	15.3841ms	15.9592ms
12	6.1792ms	7.75585ms	16.8256ms	16.9643ms
13	6.3396ms	7.373ms	18.1633ms	17.8277ms
14	6.1119ms	7.40445ms	19.4018ms	19.0304ms
15	6.12985ms	7.2103ms	21.059ms	20.0215ms
16	6.1262ms	7.16175ms	22.5224ms	21.5382ms
17	6.23625ms	7.0967ms	24.1537ms	23.0481ms
18	6.44825ms	7.1732ms	26.007ms	24.7104ms
19	6.59745ms	7.25375ms	27.5739ms	26.3333ms
20	6.7226ms	7.55205ms	29.219ms	28.1887ms
21	6.88415ms	7.60225ms	30.0254ms	29.0497ms
22	7.06235ms	7.76025ms	30.7598ms	29.9287ms
23	7.224ms	7.89365ms	31.4224ms	30.9522ms
24	7.47745ms	8.0875ms	31.8743ms	32.0007ms
25	7.6195ms	8.4237ms	33.1269ms	32.8866ms

TABLE 4.2 – Temps moyen en millisecondes pour générer une énigme et vérifier si elle est bien formée pour chaque configuration.

Nous pouvons observer que la probabilité de produire une énigme bien formée est décroissante en fonction du nombre de cellules (à partir de 3 cellules dans le cas sans cellules vide) et tend rapidement vers 0. Le temps moyen pour générer et tester une énigme dans le cas sans cellule vide en fonction du nombre de cellules est d'abord croissant, puis décroissant et est croissant de nouveau. Cela peut s'expliquer par la probabilité de produire une énigme bien formée, en effet, le test d'une énigme bien formée est généralement plus long qu'une énigme avec la même configuration qui n'est pas bien formée. On constate aussi que le temps moyen pour une énigme avec cellule vide est croissant.

4.3 Résolution d'énigmes avec solvigme

La méthode qui nous permet de vérifier si une énigme est bien formée, le fait en trouvant la solution de l'énigme si elle existe. Il est donc possible d'utiliser cette méthode pour résoudre une énigme déjà existante. Notre solveur solvigme prend en entrée un fichier d'énigme et donne la liste des cellules qui contiennent une princesse si elle est bien formée et peut écrire le fichier coq associé. Le fichier d'énigme doit respecter le format suivant :

```
V/NV
n
(A1)
(A2)
...
(An)
(H)
```

Où V/NV désigne si l'énigme peut contenir des cellules vides(V) ou non (NV), n le nombre de cellules, Ai l'affirmation de l'affiche i, H l'hypothèse du roi et l'hypothèse de répartition si il y en a une. L'hypothèse exclusive n'est pas requise, elle peut être générée à partir du nombre de cellules. Le format demande, pour les formules logiques, d'utiliser la même syntaxe que celle utilisée par Limboole (voir annexe). L'affiche i doit être mise à la ligne $i + 2$ et si Ai décrit la valeur de vérité de Aj alors $j < i$.

Dans ce format, la première énigme se définie comme suit :

```
NV
2
(PC1 & TC2)
((PC1 & TC2) | (PC2 & TC1))
((A1 & !A2) | (A2 & !A1))
```

On remarquera que la douzième énigme ne peut pas se modéliser sans réorganiser la numérotation des cellules car l'ordre croissant n'est pas respecté. En effet, la cellule 8 décrit la valeur de vérité de la cellule 9.

5. Conclusion

5.1 Bilan

Dans le cadre de TER nous avons trouvé les solutions des énigmes de la princesse et du tigre et nous les prouvées formellement en utilisant Coq et SMTCoq. Nous avons développé un générateur d'énigmes inspirées de celles de la princesse et du tigre. Ce générateur permet de jouer sur différents paramètres pour proposer des énigmes diverses. Nous avons aussi développé un solveur d'énigmes qui permet trouver la solutions à des énigmes qui respectent une construction similaire à celles de la princesse et du tigre (il permet notamment de résoudre ces dernières).

5.2 Perspectives

Il est possible d'enrichir le générateur d'énigmes en ajoutant de nouvelles hypothèses du roi et de nouveaux motifs. Il aurait été aussi possible de réfléchir à un autre type de générateur, qui part d'une solution déjà existante pour ensuite construire l'énigme autour de cette solution. L'idée serait de créer des affiches qui n'invaliderait pas la solution initiale, de faire du bruit pour noyer la bonne solution. L'avantage de cette approche est qu'elle garantirait d'avoir une énigme bien formée, mais elle ne serait pas nécessairement intéressante (c'est à dire, où l'arbre de preuve de la solution ne dépasse pas un certain nombre de noeuds). Un autre problème de cette approche est qu'il sera probablement plus difficile d'interpréter les énigmes en langage naturel.

Bibliographie

[1] R. Smullyan, *Le livre qui rend fou*. Paris : Dunod, 1989.

[2] T. C. D. Team, "The coq proof assistant, version 8.9.0," Jan. 2019.

[3] R. Dyckhoff, "Contraction-free sequent calculi for intuitionistic logic," *The Journal of Symbolic Logic*, vol. 57, no. 3, pp. 795–807, 1992.

[4] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner, "A modular integration of sat/smt solvers to coq through proof witnesses," in *International Conference on Certified Programs and Proofs*, pp. 135–150, Springer, 2011.

[5] Limboole, version 1.1, http ://fmv.jku.at/limboole/.

[6] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of reasoning*, pp. 466–483, Springer, 1983.

[7] D. Challenge, "Satisfiability : Suggested format," *DIMACS Challenge. DIMACS*, 1993.

Annexe

Correspondance des symboles logiques

Logique	Coq Prop	Coq bool	Limboole
$\neg$ A	~A	negb A	!A
A $\wedge$ B	A /\B	A && B	A & B
A $\vee$ B	A \/ B	A || B	A | B
A $\rightarrow$ B	A -> B	A --> B	A -> B
A $\leftrightarrow$ B	A <-> B	eqb A B	A <-> B

Notice de genigme

```
usage: genigme <nbcell> [ <option> ... ]

  -h              Affiche cette aide et quitte.
  -vide           Permet d'avoir des cellules vide.
  -af             Permet d'avoir un arbre d'affiches.
                  (défaut : affiches indépendantes les unes des autres)
  -o <file>       Les fichiers produit seront : file.v, file.txt, file.ppta.
                  (défaut : file = enigme)
  -t <time>       Timeout au bout de time secondes
                  (défaut : time = 60)

Attention : le nombre de cellules doit être plus grand que 2.
```

Notice de solvigme

```
usage: solvigme <in-file> [ <option> ... ]

  -h                  Affiche cette aide et quitte.
  -coq <out-file>     Écrit le fichier coq associé à l'énigme dans un fichier out-file.v.

Le format à respecter pour le fichier in-file est le suivant :

V/NV        : V si l'énigme a des cellules vides, NV sinon.
n           : nombre de cellules.
(A1)        : La formule de l'affiche 1.
(A2)        : La formule de l'affiche 2.
...
(An)        : La formule de l'affiche n.
(H)         : L'hypothèse du roi et l'hypothèse de répartition si il y en a une.

Attention à utiliser les bons connecteurs logiques (&,|,->,<->,!),
à bien parenthéser les formules et à bien respecter l'ordre des affiches.
```