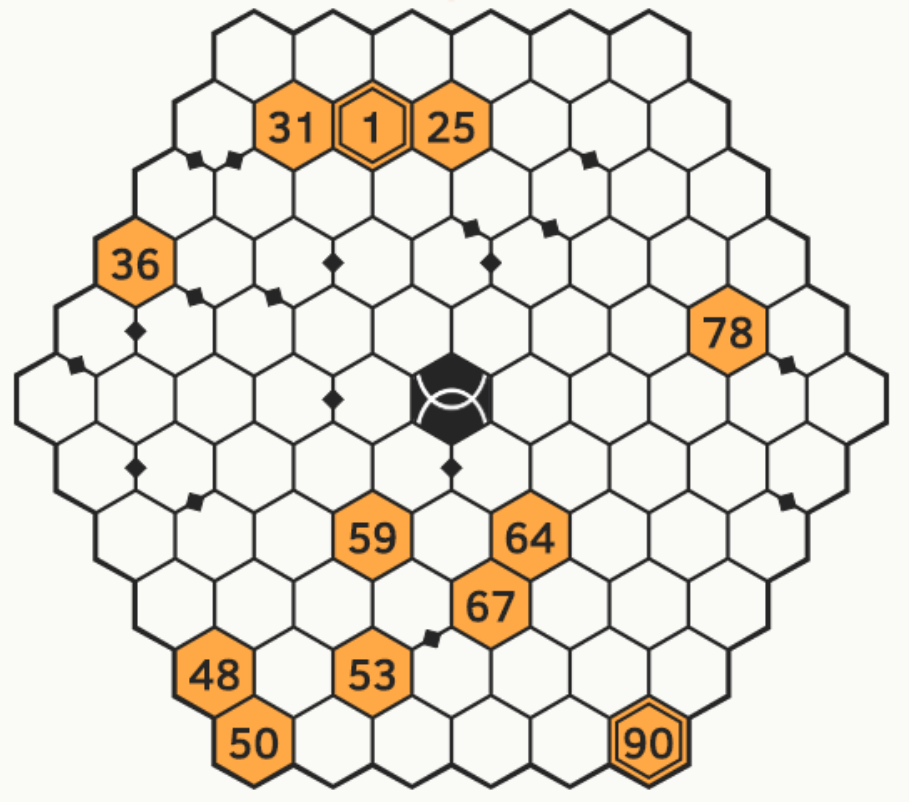
VERRIERE Maxime
IMBERT Matteo
POUILLAT Matteo
ROBICHON Nicolas

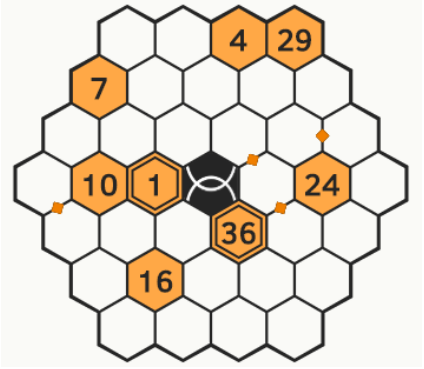
Projet INF402 : Le Rikudo



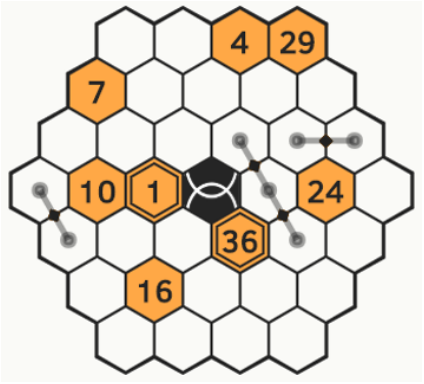
I) Introduction : Présentation du jeu

Le but du jeu est de placer les nombres de 1 à n dans la grille (n correspondant au nombre de cases) de façon à former un chemin de nombres consécutifs. Deux nombres consécutifs doivent donc être voisins. Un lien entre deux cases indique un point de passage du chemin. À la fin, toute la grille doit être remplie. En général, la première et la dernière case sont données dès le départ.

Exemple :



On remplit la grille fournie (ici à gauche) avec les nombres de 1 à 36.



Les liens entre les cases réduisent les possibilités de résolution.

La grille résolue permet de tracer un chemin de nombres consécutifs de 1 à 36.

II) Modélisation des règles du jeu sous forme de symboles logiques

On note C_i^j la variable booléenne “La case i contient le nombre j”.

Règles générales

- Dans chaque case, il y a un nombre :
$$\forall i \in [1;n],\ \exists j \in [1;n],\ C_i^j$$

- Chaque nombre se trouve dans une case :
$$\forall j \in [1;n],\ \exists i \in [1;n],\ C_i^j$$

- Une case ne contient qu’un seul nombre :
$$\forall i \in [1;n],\ \forall j \in [1;n],\ \forall k \in [1;n],\ k \neq j \Rightarrow C_i^j \Rightarrow \neg C_i^k$$

- Un nombre ne peut se trouver que dans une seule case :
$$\forall i \in [1;n],\ \forall j \in [1;n],\ \forall l \in [1;n],\ i \neq l \Rightarrow C_i^j \Rightarrow \neg C_l^j$$

- Toutes les cases ont un voisin direct dont le nombre est le suivant, sauf la dernière :
$$\forall i \in [1;n],\ C_i^n \vee (\exists j \in [1;n],\ voisin(i,j) \wedge (\exists k \in [1;n-1], C_i^k \wedge C_j^{k+1}))$$

La variable $voisin(i,j)$ signifie “les cases i et j sont voisines”. Pour avoir moins de variables on ne créera qu’une seule variable $v(i,j)$ où $i <= j$ et on a $voisin(i,j) = voisin(j,i) = v(i,j)$.

Règles spécifiques à une configuration

- On ajoute pour chaque pont entre i et j deux cases:
$$\exists k \in [1;n-1],\ (C_i^k \wedge C_j^{k+1}) \vee (C_i^{k+1} \wedge C_j^k)$$
- On ajoute $voisin(i,j)$ ou $\neg voisin(i,j)$ selon si les cases sont voisines ou non
- On ajoute pour les cases i préremplies avec j C_i^j.

N.B. : Dû à nos difficultés pour traiter des cas avec un nombre de cases importants (voir en détails ci-après), nous n’avons pas traité cet aspect du problème dans notre programme.

III) Modélisation du problème en forme normale conjonctive

Nous avons choisi d'illustrer la modélisation du problème en forme normale conjonctive à travers un exemple traité par le SAT-solveur, car le nombre de clauses devient vite immense, et n'a donc plus grand intérêt à la lecture. Il s'agit d'une grille 3X3, avec trois cases inconnues (fichier "3x3_solution_unique.carte"). Voici les règles affichées en FNC par le programme :

Règle 1 :
C_7_3 C_7_5 C_7_7
C_5_7 C_5_3 C_5_5
C_1_7 C_1_3 C_1_5

Règle 2 :
C_7_5 C_1_5 C_5_5
C_1_7 C_7_7 C_5_7
C_1_3 C_5_3 C_7_3

Règle 3 :
-C_1_3 -C_1_7
-C_1_3 -C_1_5
-C_5_3 -C_5_7
-C_5_5 -C_5_3
-C_5_5 -C_5_7
-C_7_5 -C_7_7
-C_7_3 -C_7_5
-C_7_3 -C_7_7
-C_1_5 -C_1_7

Règle 4 :
-C_7_7 -C_1_7
-C_7_7 -C_5_7
-C_5_7 -C_1_7
-C_7_5 -C_1_5
-C_7_3 -C_5_3
-C_1_3 -C_7_3
-C_1_3 -C_5_3
-C_7_5 -C_5_5
-C_1_5 -C_5_5

Règle 5
C_7_5 C_1_5 C_5_5
C_5_7 C_5_3 C_5_5
C_1_3 C_1_5
C_7_3 C_7_7
C_1_7 C_5_7
C_7_3

Chaque "C_i_j" représente une clause. En particulier, "C_i_j" signifie "le nombre j se situe dans la case i", et le "-" correspond à la négation. De plus, les cases préremplies sont prises en compte dans cet ensemble de clauses. C'est-à-dire que les règles sont appliquées à ce que l'on sait déjà être vrai, et donc aussi appliquées aux cases voisines. Cela simplifie directement le problème en réduisant le nombre de clauses.
Nous avons fait en sorte de pouvoir afficher facilement les clauses pour chaque problème traité (voir Informations complémentaires).

IV) Points critiques lors de l'implémentation

Nous avons (ou du moins pensons avoir) réussi à modéliser et faire résoudre au SAT-solveur les grilles de Rikudo. Toutefois, ce jeu étant notamment basé sur des liens entre cases voisines, il devient très difficile voire impossible pour nos machines d'exécuter le programme à partir d'un nombre de cases relativement faible…

Nous avons tenté de simplifier le problème en l'appliquant à des grilles carrées plutôt qu'hexagonales pour réduire le nombre de clauses à traiter, mais cela n'a pas pour autant permis de résoudre des problèmes de grande ampleur.

En particulier, la grille maximale que nous avons réussi à résoudre avait une taille de 2X3, ou 3X3 mais avec au moins 3 cases préremplies.

V) Traduction sous forme de fonctions des connecteurs logiques et mathématiques

Pour résoudre le problème lié au jeu, nous avons choisi de créer des classes associées aux concepts mathématiques de $\forall$, $\exists$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\neg$ les voici :

Voici la classe qui traduit les cas où nous avons besoin d'utiliser un $\forall$.
Elle a été créée afin de faire parcourir la variable liée sur le domaine de l'expression. Ensuite on effectue des simplifications, c'est-à-dire : si on a un Bottom dans l'expression alors on arrête de parcourir et on renvoie Bottom ; dans le contraire on continue.

```
class Pourtout(Expr):
    """
    Le quantificateur pourtout.
    """
    def __init__(self,
                 index: Index,
                 dom: Iterable[int],
                 expr: Expr) -> None:
        self.expr = expr  # L'expression contenue
        self.index = index  # L'indice lié par le pour tout. Dans l'exemple
au-dessus c'est x
        self.dom = dom  # Le domaine que va parcourir l'indice.

    def build(self) -> propositionnelle.Expr:
        expr = propositionnelle.Produit([])
        for i in self.dom:
            self.index.valeur = i
            e2 = self.expr.build()
            if isinstance(e2, propositionnelle.Bottom):
                return propositionnelle.Bottom()
            elif isinstance(e2, propositionnelle.Top):
                continue
            expr.termes.append(e2)
        return  expr if len(expr.termes) > 0 else propositionnelle.Top()
```

Pour la classe ∃ maintenant : de la même façon, on parcourt le domaine de l'expression. Au niveau des simplifications, contrairement à la classe précédente, c'est dans le cas où l'on a un Top dans l'expression que l'on s'arrête de parcourir et on renvoie donc Top ; dans le cas contraire, on continue.

```
class Ilexiste(Expr):
    """
    Le quantificateur il existe.
    """
    def __init__(self,
                 index: Index,
                 dom: Iterable[int],
                 expr: Expr) -> None:
        # Voir Pourtout pour les explications
        self.expr = expr
        self.index = index
        self.dom = dom

    def build(self) -> propositionnelle.Expr:
        expr= propositionnelle.Somme([])
        for i in self.dom:
            self.index.valeur = i
            e2 = self.expr.build()
            if isinstance(e2, propositionnelle.Bottom):
                continue
            elif isinstance(e2, propositionnelle.Top):
                return propositionnelle.Top()
            expr.termes.append(e2)
        return expr if len(expr.termes) > 0 else propositionnelle.Bottom()
```

Ensuite, voici les classes associées au $\wedge$ et au $\vee$ logique.
Ici aussi, s'il existe des simplifications évidentes, elles sont faites dans la fonction.
Par exemple, s'il y a un côté d'un $\wedge$ où l'on trouve Bottom, on renvoie Bottom.
Respectivement, si l'on a un Top d'un côté d'un $\vee$, alors on renvoie Top.

```
class Et(Expr):
    def __init__(self, gauche: Expr, droite: Expr):
        self.gauche = gauche
        self.droite = droite

    def build(self) -> propositionnelle.Expr:
        gauche = self.gauche.build()
        droite = self.droite.build()
        if isinstance(gauche, propositionnelle.Top) or
isinstance(droite, propositionnelle.Bottom):
            return droite
        elif isinstance(gauche, propositionnelle.Bottom) or
isinstance(droite, propositionnelle.Top):
            return gauche
        else:
            return propositionnelle.Et(gauche, droite)


class Ou(Expr):
    def __init__(self, gauche: Expr, droite: Expr):
        self.gauche = gauche
        self.droite = droite

    def build(self) -> propositionnelle.Expr:
        gauche = self.gauche.build()
        droite = self.droite.build()
        if isinstance(gauche, propositionnelle.Top) or
isinstance(droite, propositionnelle.Bottom):
            return gauche
        elif isinstance(gauche, propositionnelle.Bottom) or
isinstance(droite, propositionnelle.Top):
            return droite
        else:
            return propositionnelle.Ou(gauche, droite)
```

Pour finir, nous avons les classes qui traduisent des ⇒ et des ⇔ dans lesquelles des modifications sont effectuées.
Comme l'⇔ est traduite par deux ⇒, les simplifications sont effectuées dans cette dernière.
Si par exemple le membre de gauche est Bottom, ou si le membre de droite est Top, on renvoie Top ; mais si on a Top à gauche ou Bottom à droite, alors on renvoie Bottom.

```
class Equiv(Expr):
    def __init__(self, gauche: Expr, droite: Expr):
        self.gauche = gauche
        self.droite = droite

    def build(self) -> propositionnelle.Expr:
        return propositionnelle.Equiv(self.gauche.build(),
self.droite.build())


class Implique(Expr):
    def __init__(self, gauche: Expr, droite: Expr):
        self.gauche = gauche
        self.droite = droite

    def build(self) -> propositionnelle.Expr:
        gauche = self.gauche.build()
        droite = self.droite.build()
        if isinstance(gauche, propositionnelle.Top):
            return droite
        elif isinstance(gauche, propositionnelle.Bottom) or
isinstance(droite, propositionnelle.Top):
            return propositionnelle.Top()
        elif isinstance(droite, propositionnelle.Bottom):
            return propositionnelle.Non(gauche)
        else:
            return propositionnelle.Implique(gauche, droite)
```

Format des fichiers de grille à résoudre (.carte)

Lors de nos débuts, nous avions imaginé un format de fichier. Celui-ci a beaucoup été modifié au cours du projet. En effet, nous n'avons pas exploité le concept de pont (comme expliqué plus tôt), ou nous n'avons finalement pas donné le nombre d'arêtes qui n'a que peu de sens pour des plateaux de la taille que nous arrivons à résoudre.

Voici donc le format du fichier final :

```
NLIGNES NCOLONNES
Valeur... Valeur (NCOLONNES fois)
...   (NLIGNES FOIS)
Valeur... Valeur (NCOLONNES fois)
```

Où “Valeur” désigne soit un nombre entre 0 et NLIGNES*NCOLONNES-1 si on connaît la valeur, ou bien “?” si on ne la connaît pas.

VI) Étapes de résolution

Voici un résumé par étapes de la résolution effectuée par notre programme :

- Fichier format “.carte”
- Transformation des règles et des informations du fichier en 1er ordre
- Transformation en logique propositionnelle, Élimination des Top et Bottom
- Transformation en forme normale : élimination des implications et équivalences, déplacement des négations
- Transformation en forme normale conjonctive : transformation en produit de clauses
- Transformation en liste de clauses
- Passage au format dimacs
- Résolution du sat solveur
- Interprétation du résultat du sat solveur

VII) Informations complémentaires

Pour faciliter l’utilisation de notre programme, un fichier “config.py” a été créé.
Il permet :

- de changer facilement de SAT-solveur, en changeant tout simplement son nom dans la ligne 2 ;
- de pouvoir afficher l’ensemble des règles sous forme clausale et directement appliquées au problème traité.

Pour exécuter le programme dans son ensemble, il faut lancer le fichier “execution.py”, en entrant en paramètre la carte à résoudre.

VIII) Conclusion

Nous avons été capables de formaliser la résolution d’un problème (le Rikudo), et d’implémenter les outils de la logique dans un programme, même si notre approche très générale nous a conduit à de mauvaises performances. Notre solution est en conclusion simplement fonctionnelle.