CentraleSupélec

Projet Solveur Bloxorz

Connaissances et raisonnement
Mention Intelligence Artificielle

Marius Nadalin : marius.nadalin@student-cs.fr
Antoine Dieu : antoine.dieu@student-cs.fr

Janvier 2025

Table des matières

Introduction

Bloxorz est un jeu de puzzle où le joueur doit contrôler un parallélépipède rectangle (concaténation de deux cubes, que l'on appellera un bloc) et le déplacer d'un point A à un point B pour réussir un niveau.

Dans ce projet, nous proposons un solveur au Bloxorz qui repose sur un solveur SAT (Gophersat). L'objectif est de modéliser les règles du jeu en contraintes logiques et d'utiliser Gophersat pour déterminer une solution optimale, si elle existe, sous la contrainte d'un nombre maximal de mouvements.

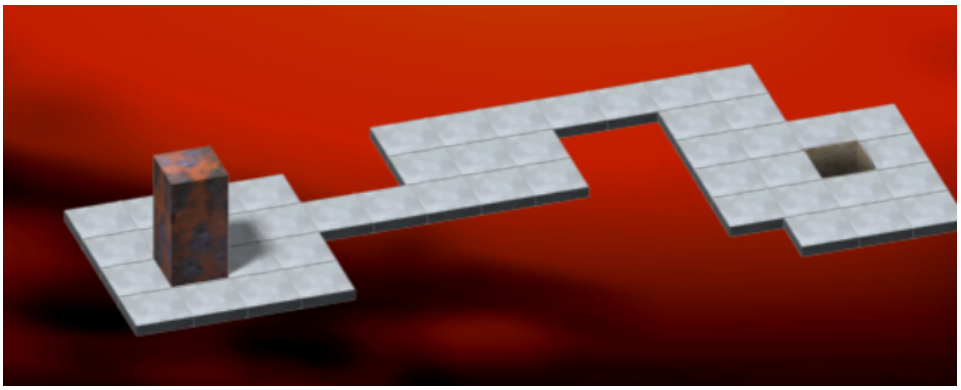


FIGURE 2 – Exemple d'un niveau du jeu original.

Lien vers le projet

Le projet est consultable sur GitHub à l'adresse suivante :
GitHub - Projet Bloxorz Solver.

1 Description des règles du Bloxorz

Le bloc ne peut prendre que trois positions :
— Debout (*up*, $\boldsymbol{UP}$)
— Allongé horizontalement (*down horizontal*, $\boldsymbol{DH}$)
— Allongé verticalement (*down vertical*, $\boldsymbol{DV}$)

Selon la position dans laquelle il se trouve, le bloc peut occuper une à deux cases adjacentes.

Chaque coup du joueur correspond à un mouvement du bloc : haut ($\boldsymbol{m_{up}}$), bas ($\boldsymbol{m_{down}}$), gauche ($\boldsymbol{m_{left}}$), droite ($\boldsymbol{m_{right}}$). Le bloc peut alors se déplacer en se couchant, en se levant, ou en roulant latérallement.

Un coup peut être joué uniquement s'il est autorisé, ce qui sera possible dans la majorité des cas si les cases occupées après déplacement sont des cases *"sol"* (i.e. des cases solides).

L'objectif du jeu est de déplacer le bloc sur la grille jusqu'à ce qu'il atteigne la case "trou" en position $\boldsymbol{UP}$.

2 Modélisation en logique propositionnelle du jeu

Chaque niveau est constitué d'une grille dont l'ensemble des coordonnées des cases *"sol"* est noté $\boldsymbol{G}$, la (ou les) case de début $B \subset \boldsymbol{G}$ et la case *"trou"* d'arrivée $e \in \boldsymbol{G}$. On notera $\boldsymbol{N} = |\boldsymbol{G}|$ le nombre de cases, et $\{\boldsymbol{c_n} \mid \boldsymbol{n} \in [1, \boldsymbol{N}]\}$ les cases dont les coordonnées seront stockées dans un dictionnaire.
On note $\boldsymbol{\sigma} : (i, j) \to \boldsymbol{n}$ la fonction qui associe à aux coordonnées d'une case son indice.

Nous reformulons l'objectif du solveur ainsi :
Soit $(\boldsymbol{G}, B, e)$ une grille représentant un niveau et $\boldsymbol{T}$ un horizon temporel, existe-t'il une combinaison de $\boldsymbol{T}$ coups qui déplace le bloc de B à e en position $\boldsymbol{UP}$? Si oui, laquelle ?

2.1 Variables

Nous souhaitons suivre l'état de la grille à chaque pas de temps $\boldsymbol{t} \in [0, \boldsymbol{T}]$.
On pose :

- $\boldsymbol{X_{t+1,n}}$, $\boldsymbol{m_t^{up}}$, $\boldsymbol{m_t^{down}}$, $\boldsymbol{m_t^{left}}$, $\boldsymbol{m_t^{right}}$ les variables de mouvements qui valent 1 si le coup est joué à l'instant $\boldsymbol{t}$ (transition entre $\boldsymbol{t}$ et $\boldsymbol{t}+1$).
- $\boldsymbol{UP_{t,n}}$, $\boldsymbol{DH_{t,n}}$, $\boldsymbol{DV_{t,n}}$ les variables d'état qui valent 1 si la case $\boldsymbol{c_n}$ est occupée par le bloc en position resp. debout, allongé horizontalement ou allongé verticalement à l'instant $\boldsymbol{t}$.

On compte donc $\boldsymbol{T} \cdot 4 + (\boldsymbol{T}+1) \cdot 3 \cdot \boldsymbol{N} = (\boldsymbol{T}+1) \cdot (3 \cdot \boldsymbol{N} + 4) - 4$ variables binaires.

2.2 Contraintes

A présent, nous souhaitons implémenter les contraintes qui permettront de diriger la résolution du problème par un solveur.

2.2.1 Disposition initiale

Premièrement, on contraint les variables à suivre la disposition initiale du niveau :
Pour chaque case $\boldsymbol{c_n}$, si $\boldsymbol{c_n} \in B$ alors on impose $\boldsymbol{X_{0,n}}$ où $\boldsymbol{X} \in \{\boldsymbol{UP}, \boldsymbol{DH}, \boldsymbol{DV}\}$ correspond à la position de départ. Dans les autres cas (s'il ne s'agit pas d'une case de départ ou de la position de départ), on impose $\overline{\boldsymbol{X_{0,n}}}$.

2.2.2 Objectif

Ensuite, on contraint la position finale : Si $\boldsymbol{c_n} = e$ alors on impose $\boldsymbol{UP_{T,n}}$. Il est inutile de mettre les valuations des autres cases à 0 par construction du problème (cf 2.4).

2.2.3 Mouvements

A chaque étape t, exactement une direction de déplacement doit être choisit parmi les quatre possibles. Autrement dit, au moins et au plus une direction doit être choisit entre l'instant t et t+1.
Cela se traduit par les équations booléennes suivantes :

$$\forall \boldsymbol{t} \in \{0, \boldsymbol{T}-1\} : \boldsymbol{m_t^{up}} \vee \boldsymbol{m_t^{down}} \vee \boldsymbol{m_t^{left}} \vee \boldsymbol{m_t^{right}} \tag{1}$$

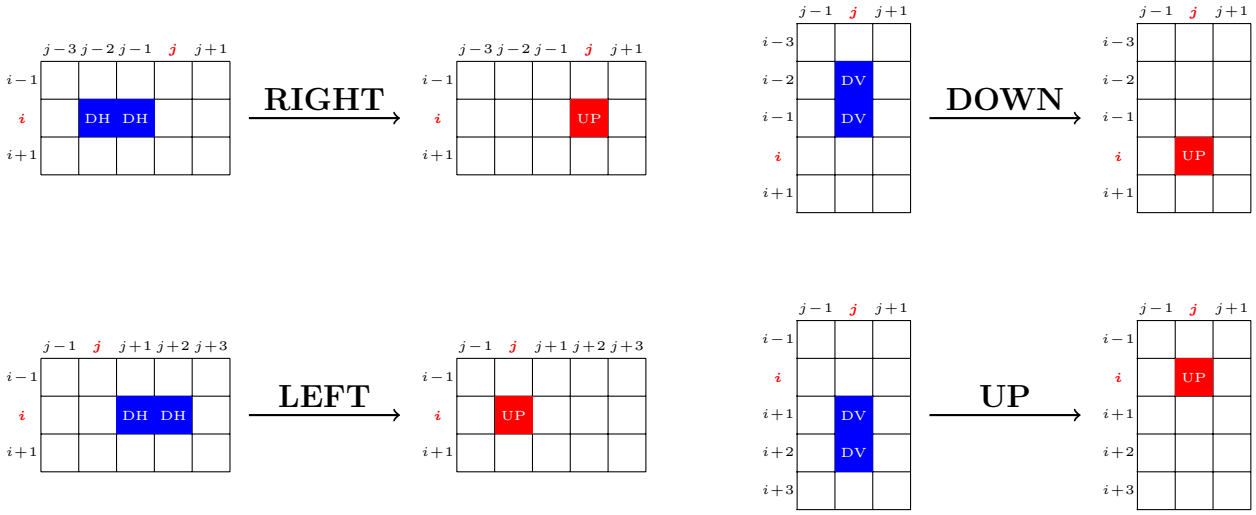
$$\forall \boldsymbol{t} \in \{0, \boldsymbol{T}-1\}, \left| \begin{array}{l} \forall \boldsymbol{dir} \in \{up, down, left, right\} \\ \forall \boldsymbol{dir'} \in \{up, down, left, right\} \\ \boldsymbol{dir} \neq \boldsymbol{dir'} \end{array} \right. : \overline{\boldsymbol{m_t^{dir}}} \vee \overline{\boldsymbol{m_t^{dir'}}} \tag{2}$$

Enfin, il reste à définir les transitions d'une grille $\boldsymbol{t}$ vers une grille $\boldsymbol{t}+1$. Pour cela, on va définir pour chaque case $\boldsymbol{c_n}$ les *conditions* permettant d'aboutir à l'état suivant $\boldsymbol{X_{t+1,n}}$, où $\boldsymbol{X} \in$

$\{\boldsymbol{UP}, \boldsymbol{DH}, \boldsymbol{DV}\}$.

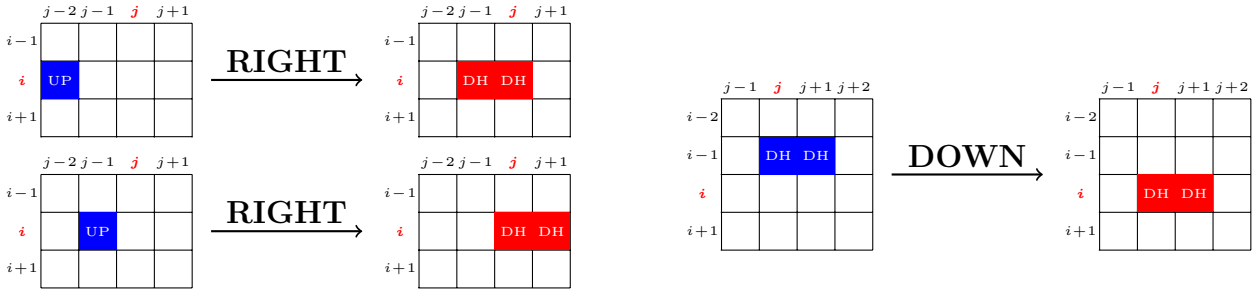
Chaque *condition* est une expression booléenne comprenant un (ou plusieurs) état de la grille précédente $\boldsymbol{X_{t,n'}}$ et un mouvement de transition $\boldsymbol{m_t}$.

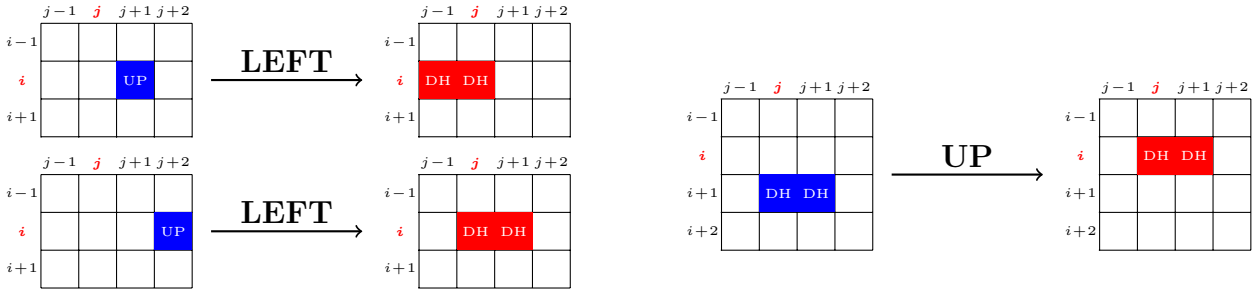
Soit $\boldsymbol{n} = \boldsymbol{\sigma}(i, j)$. On identifie les transitions suivantes :



La condition de transition vers la position $\boldsymbol{UP_{t+1,n}}$ s'écrit donc :

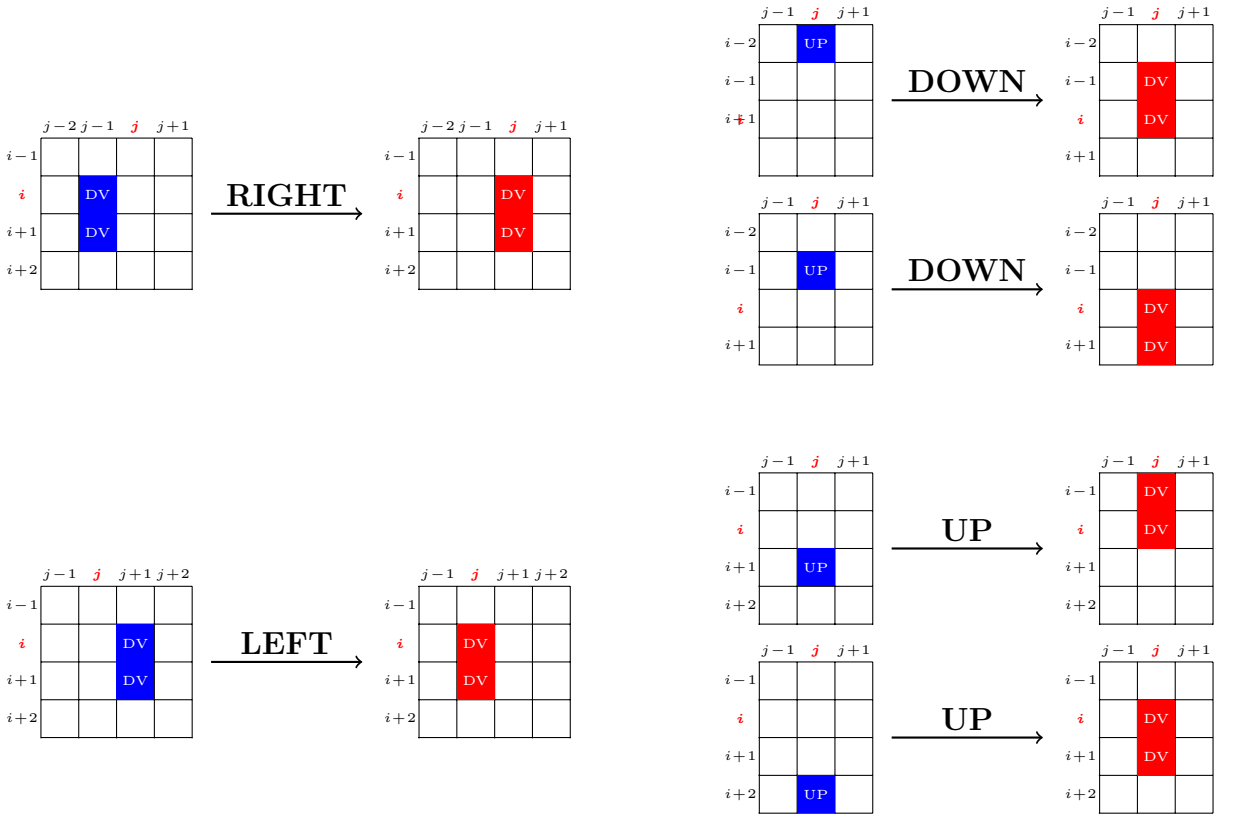
$$
\begin{aligned}
\mathit{condition}^{\boldsymbol{UP}}_{\boldsymbol{t+1,n}} = & \Big(\boldsymbol{DH_{t,\sigma(i,j-2)}} \wedge \boldsymbol{DH_{t,\sigma(i,j-1)}} \wedge \boldsymbol{m_t^{right}}\Big) \vee \\
& \Big(\boldsymbol{DV_{t,\sigma(i-2,j)}} \wedge \boldsymbol{DV_{t,\sigma(i-1,j)}} \wedge \boldsymbol{m_t^{down}}\Big) \vee \\
& \Big(\boldsymbol{DH_{t,\sigma(i,j+2)}} \wedge \boldsymbol{DH_{t,\sigma(i,j+1)}} \wedge \boldsymbol{m_t^{left}}\Big) \vee \\
& \Big(\boldsymbol{DV_{t,\sigma(i+2,j)}} \wedge \boldsymbol{DV_{t,\sigma(i+1,j)}} \wedge \boldsymbol{m_t^{up}}\Big)
\end{aligned}
\tag{3}
$$





La condition de transition vers la position $\boldsymbol{DH_{t+1,n}}$ s'écrit donc :

$$
\begin{aligned}
\mathit{condition}^{\boldsymbol{DH}}_{\boldsymbol{t+1,n}} = & \Big(\Big(\boldsymbol{UP_{t,\sigma(i,j-2)}} \vee \boldsymbol{UP_{t,\sigma(i,j-1)}}\Big) \wedge \boldsymbol{m_t^{right}}\Big) \vee \\
& \Big(\boldsymbol{DH_{t,\sigma(i-1,j)}} \wedge \boldsymbol{m_t^{down}}\Big) \vee \\
& \Big(\Big(\boldsymbol{UP_{t,\sigma(i,j+2)}} \vee \boldsymbol{UP_{t,\sigma(i,j+1)}}\Big) \wedge \boldsymbol{m_t^{left}}\Big) \vee \\
& \Big(\boldsymbol{DH_{t,\sigma(i+1,j)}} \wedge \boldsymbol{m_t^{up}}\Big)
\end{aligned}
\tag{4}
$$



La condition de transition vers la position $\boldsymbol{DV_{t+1,n}}$ s'écrit donc :

$$
\begin{aligned}
condition^{\boldsymbol{DV}}_{\boldsymbol{t+1,n}} = & \Big(\boldsymbol{DV}_{\boldsymbol{t,\sigma(i,j-1)}} \wedge \boldsymbol{m}^{\boldsymbol{right}}_{\boldsymbol{t}}\Big) \vee \\
& \Big(\Big(\boldsymbol{UP}_{\boldsymbol{t,\sigma(i-2,j)}} \vee \boldsymbol{UP}_{\boldsymbol{t,\sigma(i-1,j)}}\Big) \wedge \boldsymbol{m}^{\boldsymbol{down}}_{\boldsymbol{t}}\Big) \vee \\
& \Big(\boldsymbol{DV}_{\boldsymbol{t,\sigma(i,j+1)}} \wedge \boldsymbol{m}^{\boldsymbol{left}}_{\boldsymbol{t}}\Big) \vee \\
& \Big(\Big(\boldsymbol{UP}_{\boldsymbol{t,\sigma(i+2,j)}} \vee \boldsymbol{UP}_{\boldsymbol{t,\sigma(i+1,j)}}\Big) \wedge \boldsymbol{m}^{\boldsymbol{up}}_{\boldsymbol{t}}\Big)
\end{aligned}
\tag{5}
$$

Ainsi, on ajoute les équations booléennes suivantes :

$$
\forall \boldsymbol{t} \in \{0, \boldsymbol{T}-1\}, \forall \boldsymbol{n} \in \{0, \boldsymbol{N}\} : \begin{cases} condition^{\boldsymbol{UP}}_{\boldsymbol{t+1,n}} & \iff \boldsymbol{UP}_{\boldsymbol{t+1,n}} \\ condition^{\boldsymbol{DH}}_{\boldsymbol{t+1,n}} & \iff \boldsymbol{DH}_{\boldsymbol{t+1,n}} \\ condition^{\boldsymbol{DV}}_{\boldsymbol{t+1,n}} & \iff \boldsymbol{DV}_{\boldsymbol{t+1,n}} \end{cases} \tag{6}
$$

2.3 Règles spéciales

Certains niveaux présentent des cases spéciales, introduisant de nouvelles règles à respecter pour les résoudre.

2.3.1 Cases rouges

Des cases *"sol"* de couleur rouge apparaissent pour la première fois dans le niveau 4 du jeu original (Figure 3). Ces cases s'effondrent sous le poids du bloc si celui-ci se tient droit (position $\boldsymbol{UP}$) sur l'une d'entre elles, causant l'échec du niveau.

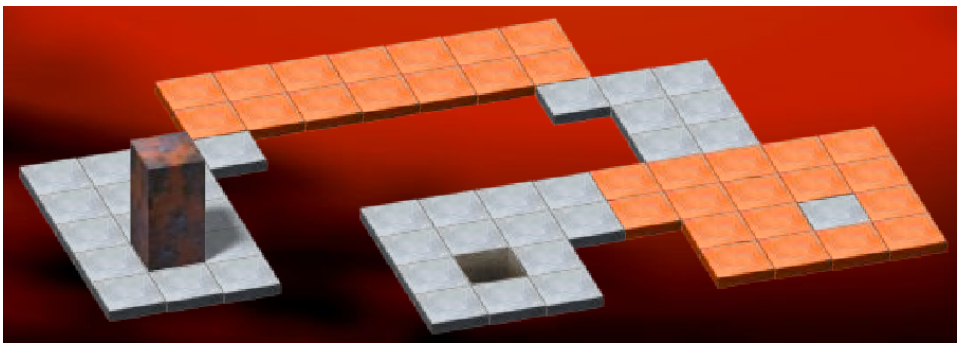


FIGURE 3 – Niveau 4 du jeu original.

Pour prendre en compte cette contrainte, il suffit d'interdire les cases concernées de comprendre l'état $\boldsymbol{UP}$ à tout instant :

$$
\forall \boldsymbol{t} \in \{0, \boldsymbol{T}\}, \forall \boldsymbol{c_n} \in \{\boldsymbol{c} \in \boldsymbol{G} \mid \boldsymbol{c} \textit{ est une case rouge}\} : \overline{\boldsymbol{UP}_{\boldsymbol{t,n}}} \tag{7}
$$

2.3.2 Boutons et cases contrôlées

Dès le deuxième niveau du jeu original (Figure 4), une nouvelle mécanique est introduite : les boutons qui contrôlent certaines cases du plateau. Lorsqu'un bouton est activé en posant le bloc dessus, il peut déclencher l'apparition et/ou la disparition de ponts et de plateformes.

Il existe plusieurs types de boutons : les ronds, qui s'activent aussi bien en position debout qu'allongée ($\boldsymbol{UP}$, $\boldsymbol{DH}$ ou $\boldsymbol{DV}$), et les croix, qui nécessitent que le bloc soit debout ($\boldsymbol{UP}$) pour être

enclenchés. On notera $\boldsymbol{G^{\circ}_{c+}}$ (resp. $\boldsymbol{G^{\circ}_{c-}}$) l'ensemble des cases comportant un bouton rond pouvant activer (resp. désactiver) la case contrôlable $\boldsymbol{c}$, et $\boldsymbol{G^{\times}_{c+}}$ (resp. $\boldsymbol{G^{\times}_{c-}}$) celles avec un bouton en croix.

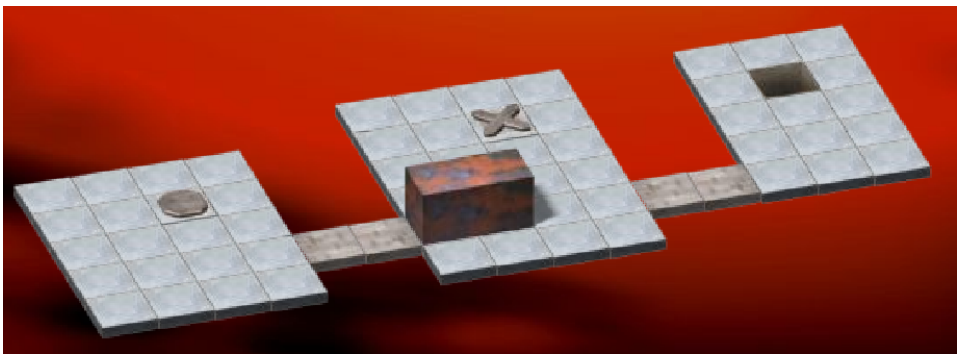


FIGURE 4 – Niveau 2 du jeu original. Le pont de gauche se déploie après pression du bouton rond en position quelconque. Le pont droit se déploie un fois le bloc debout sur le bouton en forme de croix.

Pour modéliser cette interaction, il faut d'abord introduire de nouvelles variables booléennes qui traduisent l'état des cases contrôlables (plateformes déployables).

On pose $\boldsymbol{active_{t,c}}$ la variable qui vaut 1 si la case contrôlable $\boldsymbol{c}$ est activée (solide) à l'instant $\boldsymbol{t}$, et 0 sinon.

En notant $\boldsymbol{N_c}$ le nombre de cases contrôlables, on compte alors $\boldsymbol{T} \cdot 4 + (\boldsymbol{T}+1) \cdot (3 \cdot \boldsymbol{N} + \boldsymbol{N_c}) = (\boldsymbol{T}+1) \cdot (3 \cdot \boldsymbol{N} + \boldsymbol{N_c} + 4) - 4$ variables binaires.

L'évolution de cette variable au cours du temps est la suivante : Si la case contrôlable était active précédemment et qu'aucun bouton pouvant la désactiver n'est enclenché, alors elle reste active. Autrement, si la case était désactivée et qu'un bouton pouvant l'activer est pressé, alors elle devient active. Inversement pour la désactivation.

Soit $\boldsymbol{c}$ une case contrôlable;

$$
\begin{aligned}
\forall \boldsymbol{t} \in \{1, \boldsymbol{T}\} : \boldsymbol{active_{t,c}} \iff & \Big(\Big(\bigvee_{\boldsymbol{c_n} \in \boldsymbol{G^{\times}_{c+}}} \boldsymbol{UP_{n,t}} \Big) \wedge \overline{\boldsymbol{active_{t-1,c}}} \Big) \vee \\
& \Big(\Big(\bigvee_{\boldsymbol{c_n} \in \boldsymbol{G^{\circ}_{c+}}} \boldsymbol{UP_{n,t}} \vee \boldsymbol{DH_{n,t}} \vee \boldsymbol{DV_{n,t}} \Big) \wedge \overline{\boldsymbol{active_{t-1,c}}} \Big) \vee \\
& \Big(\Big(\bigwedge_{\boldsymbol{c_n} \in \boldsymbol{G^{\times}_{c-}}} \overline{\boldsymbol{UP_{n,t}}} \Big) \wedge \boldsymbol{active_{t-1,c}} \Big) \vee \\
& \Big(\Big(\bigwedge_{\boldsymbol{c_n} \in \boldsymbol{G^{\circ}_{c-}}} \overline{\boldsymbol{UP_{n,t}}} \wedge \overline{\boldsymbol{DH_{n,t}}} \wedge \overline{\boldsymbol{DV_{n,t}}} \Big) \wedge \boldsymbol{active_{t-1,c}} \Big)
\end{aligned}
\tag{8}
$$

Maintenant, il suffit de modifier les conditions de déplacement (6) pour prendre en compte l'état des cases contrôlables :

$$\forall \boldsymbol{t} \in \{0, \boldsymbol{T}-1\}, \forall \boldsymbol{n'} \in \{\boldsymbol{n} \mid \boldsymbol{c_n} \textit{ est contrôlable}\} :$$

$$\begin{cases} condition^{\boldsymbol{UP}}_{\boldsymbol{t+1,n'}} \wedge \boldsymbol{active_{t+1,c_{n'}}} & \iff \boldsymbol{UP_{t+1,n'}} \\ condition^{\boldsymbol{DH}}_{\boldsymbol{t+1,n'}} \wedge \boldsymbol{active_{t+1,c_{n'}}} & \iff \boldsymbol{DH_{t+1,n'}} \\ condition^{\boldsymbol{DV}}_{\boldsymbol{t+1,n'}} \wedge \boldsymbol{active_{t+1,c_{n'}}} & \iff \boldsymbol{DV_{t+1,n'}} \end{cases} \tag{9}$$

2.4 Démonstration de la correction de la modélisation

Nous avons posé en 2.2 des contraintes sur la position initiale, la position finale, et les mouvements licites. Nous cherchons maintenant à montrer que ces contraintes sont suffisantes, c'est à dire que toutes les positions intermédiaires seront par construction correctes pour une solution correcte (cas où le problème est satisfiable).

On pose les variables booléennes suivantes :

— P_t^{UP} : "Le bloc est en position debout à l'instant $\boldsymbol{t}$" (exactement une variable de position $UP_{t,n}$ vaut 1 à l'instant $\boldsymbol{t}$),

— P_t^{DH} : "Le bloc est en position allongé horizontalement à l'instant $\boldsymbol{t}$" (exactement deux variables de position adjacentes DH_{t,n_1} et DH_{t,n_2} valent 1 à l'instant $\boldsymbol{t}$),

— P_t^{DV} : "Le bloc est en position allongé verticalement à l'instant $\boldsymbol{t}$" (exactement deux variables de position adjacentes DV_{t,n_1} et DV_{t,n_2} valent 1 à l'instant $\boldsymbol{t}$).

Nous cherchons alors à montrer que : $\forall \boldsymbol{t} \in \{0, \boldsymbol{T}\}, P_t^{UP} + P_t^{DH} + P_t^{DV} = 1$

La propriété est vérifiée à l'instant $\boldsymbol{t} = 0$ d'après la contrainte 2.2.1.

Supposons la propriété vraie à l'instant $\boldsymbol{t}$, montrons qu'elle le sera à l'instant $\boldsymbol{t+1}$.

D'après (1) et (2), il existe $\boldsymbol{dir} \in \{up, down, left, right\}$ tel que $m_t^{dir} = 1$.

Par hypothèse de récurrence, il existe n (ou n_1 et n_2) tel que $UP_{t,n} = 1$ (ou $DH_{t,n_1} = 1$ et $DH_{t,n_2} = 1$ ou $DV_{t,n_1} = 1$ et $DV_{t,n_2} = 1$).

Comme décrit en 2.2.3, chaque couple de $\boldsymbol{dir}$ et d'état de grille $\boldsymbol{X_{t,n}}$ appartient à une des clauses des variables conditions. Il en résulte d'après (9) qu'une des propriétés P_t^{UP}, P_t^{DH} ou P_t^{DV} est vraie. Nous sommes assuré de l'existence de la clause (la case destination est libre) car la solution existe.

L'unicité de la direction (2), l'hypothèse de récurrence ainsi que l'équivalence de (6), donnent qu'une seule des variables P peut être vraie, et donc la propriété est vérifiée au temps $\boldsymbol{t+1}$. On peut d'ailleurs remarquer que l'équivalence de (6) aurait pu être allégée en une implication en ajoutant en échange la propriété démontrée comme contrainte au problème.

3 Description du code

Nous proposons de décrire dans cette section les parties importantes de l'algorithme permettant de résoudre un niveau. Le solveur **SAT** sera considéré comme une boite noire renvoyant une assignation de variables logiques uniquement si le fichier DIMACS fourni est **satisfiable**, et son fonctionnement ne sera pas étudié.

3.1 Résolution du problème grâce au solveur

L'objectif est de trouver, pour un niveau donné, la solution (séquence de déplacements autorisés) permettant de résoudre ce niveau en un nombre de coups minimal.

Pour cela, il suffit de mettre sous forme normale conjonctive (**CNF**) l'ensemble des équations booléennes nécessaires à la modélisation logique du niveau considéré pour un nombre de mouvement (transitions) fixé $\boldsymbol{T}$. Ensuite, le solveur SAT détermine s'il est possible de trouver une solution avec le fichier *DIMACS* correspondant aux **CNF**. Si oui, on décode la solution (passage des variables booléennes vers l'état correspondant) pour obtenir la séquence solution. Sinon, on recommence avec un nombre de transition plus grand.

Ainsi, le nombre de coup minimal sera le premier pour lequel une solution est trouvée.

Algorithm 1 Algorithm to find an optimal solution to Bloxorz

Require: $\boldsymbol{T_{min}} \geq 0$ ▷ Minimum number of time steps
Require: $\boldsymbol{T_{max}} \geq \boldsymbol{T_{min}}$ ▷ Maximum number of time steps
Require: Level ▷ The level to solve
  $\boldsymbol{T} \leftarrow \boldsymbol{T_{min}} - 1$
  Solvable $\leftarrow \boldsymbol{False}$ ▷ Boolean variable
  while ($\boldsymbol{T} < \boldsymbol{T_{max}}$) $\wedge$ (Solvable == $\boldsymbol{False}$) **do**
    $\boldsymbol{T} \leftarrow \boldsymbol{T} + 1$
    DIMACS_file $\leftarrow$ *write_cnf*(Level, $\boldsymbol{T}$) ▷ Write the **CNF** in a DIMACS file
    Solvable $\leftarrow \boldsymbol{SAT}$.*is_solvable*(DIMACS_file) ▷ Check for **SATISFIABILITY**
  end while
  if (Satisfiable == $\boldsymbol{True}$) **then**
    $\boldsymbol{T_{opt}} \leftarrow \boldsymbol{T}$
    solution_vars $\leftarrow \boldsymbol{SAT}$.*get_solution*(DIMACS_file) ▷ Get a solution
    solution_sequence $\leftarrow$ *decode*(solution_vars) ▷ Decode the logical variables
  end if

3.2 Architecture globale

Nous détaillerons ici la structure du code et les fonctions des différents fichiers trouvables sur le github du projet.

Les niveaux du jeu sont convertis au préalable en fichiers JSON pour être facilement manipulables par le solveur. Pour les résoudre, il faut appeler le fichier principal `bloxorz_solver.py` en passant le niveau en argument.

Le programme cherchera à vérifier si le niveau est solvable en T coups, en incrémentant successivement T jusqu'à trouver une solution (ou atteindre $\boldsymbol{T_{max}}$, le problème est alors considéré comme impossible).

Les différents objets et fonctions utilisés sont définis dans les fichiers suivants :

— `level_manager.py`,
— `cnf_generator.py`,
— `clauses.py`,
— `solver_run.py`,
— `graphic_display.py`.
L'architecture globale peut être résumée ainsi :

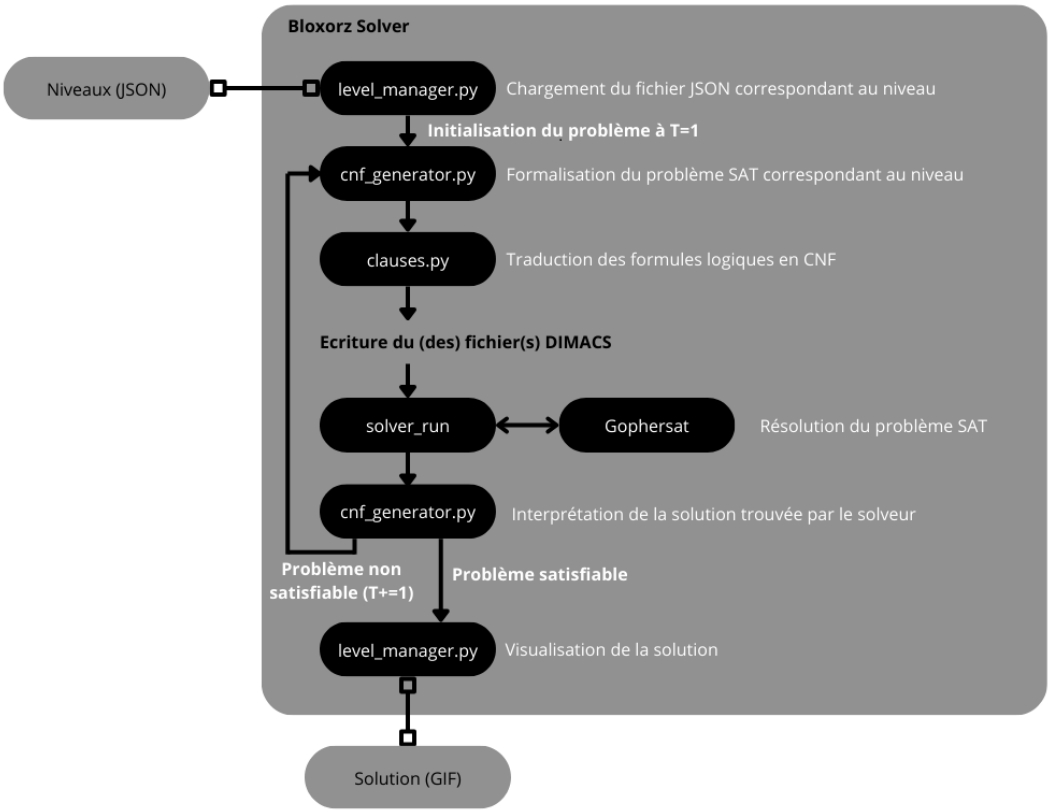


FIGURE 5 – Organisation du code avec les différents fichiers et leur intégration

3.3 De la logique propositionnelle à la forme normale conjonctive

Dans cette section, nous présentons une implémentation orientée objet de la manipulation d'expressions logiques, permettant notamment leur conversion sous forme normale conjonctive (**CNF**). Le code `python` s'appuie sur une hiérarchie de classes représentant les différents opérateurs logiques (variables, disjonctions, conjonctions et implications) et fournit une méthode standardisée `get_cnf_list` permettant d'obtenir la représentation **CNF** d'une expression sous la forme d'une liste de clauses.

1. Classe Abstraite `Clause`

— **Description :** La classe `Clause` définit l'interface commune à toutes les expressions logiques. Elle sert de classe abstraite dont héritent toutes les autres classes spécialisées.
— **Méthodes abstraites :**

- `not_()` : Retourne la négation de l'expression courante.
- `get_cnf_list()` : Renvoie la représentation de l'expression en **CNF**, c'est-à-dire une liste de clauses (où chaque clause est une liste de littéraux reliés par un **OU**) combinées par un **ET**.

2. **Classe** `VAR`
   - **Description :** La classe `VAR` représente une variable logique, identifiée par un entier.
   - **Méthodes principales :**
     - `get_cnf_list()` : Retourne la variable sous forme d'une clause unique, i.e. une liste contenant une seule sous-liste avec l'entier représentant la variable (*exemple* : `[[VAR]]`), ce qui constitue déjà une **CNF** triviale.
     - `not_()` : Retourne l'expression opposée en inversant le signe de la variable (par exemple, la négation de `3` donne `-3`).
3. **Classe** `OR`
   - **Description :** La classe `OR` modélise l'opérateur logique disjonctif (**OU**). Une instance de `OR` contient une liste d'objets de type `Clause`. On note que l'opérateur `OR` appliqué à une liste vide est traité comme équivalent à **False**.
   - **Méthodes principales :**
     - `get_cnf_list()` : La méthode vise à fusionner les **CNF** des clauses filles pour obtenir la **CNF** globale de l'expression. Pour chaque sous-expression, on récupère d'abord sa représentation **CNF** sous forme de liste de clauses. Puis, pour combiner deux **CNF** successives, la méthode distribue le disjonctif en associant chaque clause de la première **CNF** à chaque clause de la seconde. Lors de cette fusion, les doublons sont éliminés et la présence simultanée d'un littéral et de son opposé est vérifiée, de sorte qu'une clause présentant cette contradiction est alors considérée comme une **tautologie** et ignorée.
     - `not_()` : Applique la loi de *De Morgan* pour inverser l'expression : la négation d'un **OU** devient un **ET** des négations des clauses internes.
     - `add_clause(Clause)` : Permet l'ajout d'un clause à l'instance.
     - `is_false()` : Indique qu'une instance de `OR` est fausse (**False**) si la liste des clauses est vide.
4. **Classe** `AND`
   - **Description :** La classe `AND` représente l'opérateur logique conjonctif (**ET**) et contient, tout comme `OR`, une liste d'objets de type `Clause`. On note que l'opérateur `AND` appliqué à une liste vide est traité comme équivalent à `True`.
   - **Méthodes principales :**
     - `get_cnf_list()` : La méthode a pour objectif de fusionner les **CNF** issues des clauses internes en exploitant le fait qu'une conjonction de clauses **CNF** reste sous forme de **CNF**, ce qui se traduit par une simple concaténation des listes de clauses de chaque sous-expression.
     - `not_()` : Applique la loi de *De Morgan*, transformant la négation d'un **ET** en un **OU** des négations des clauses internes.
     - `add_clause(Clause)` : Permet l'ajout d'un clause à l'instance.
     - `is_true()` : Considère une instance de `AND` comme vraie (**tautologie**) si la liste des clauses est vide.

5. Classe `IMPLIES`

- **Description :** La classe `IMPLIES` modélise l'implication logique ($A \implies B$).
- **Méthodes principales :**
  - `get_cnf_list()` : Une implication **CNF** est équivalente à $\neg A \vee B$. La méthode convertit l'implication en une expression de type `OR` puis délègue la conversion en **CNF** à la méthode `get_cnf_list()` de `OR`.
  - `not_()` : Retourne la négation de l'implication, qui est équivalente à $A \wedge \neg B$.

La définition d'une interface commune via la classe abstraite `Clause` permet d'ajouter aisément les opérateurs logiques de base : `OR`, `AND`, `IMPLIES`. Ces classes facilitent l'implémentation des règles logiques (notamment les lois de *De Morgan*) et la distribution nécessaire à la conversion en **CNF**.

On note que l'équivalence $A \iff B$ utilisée dans les expressions logiques (6), (8) et (9) est implémentée sous la forme $(A \implies B) \wedge (B \implies A)$.

4 Conclusion

Lors de ce projet nous avons donc réalisé un solveur déterministe au jeu Bloxorz en nous appuyant sur le solveur SAT Gophersat.

Les premiers niveaux du jeu (terrains de $\approx 10 \times 15$ cases) sont résolus en moins de dix secondes et 40 coups, mais nous pouvons nous interroger sur les performances du solveur sur des plus grosses instances. En effet, pour le niveau 12 qui est résolu en 65 coups, le temps de résolution est déjà de 2 minutes.

Il est impossible de donner la complexité du problème en fonction de l'entrée, puisque le nombre de variables au problème SAT est proportionnel au nombre de cases et polynomial en le nombre de coups (qu'on ne peut connaître à l'avance). Cependant, le problème SAT étant intrinséquement NP-complet, pour un niveau et un nombre de coups autorisés donnés nous pouvons prévoir une complexité au moins exponentielle. Empiriquement, nous voyons en effet une tendance exponentielle entre le temps de résolution du niveau 12 et le nombre de coups autorisés.

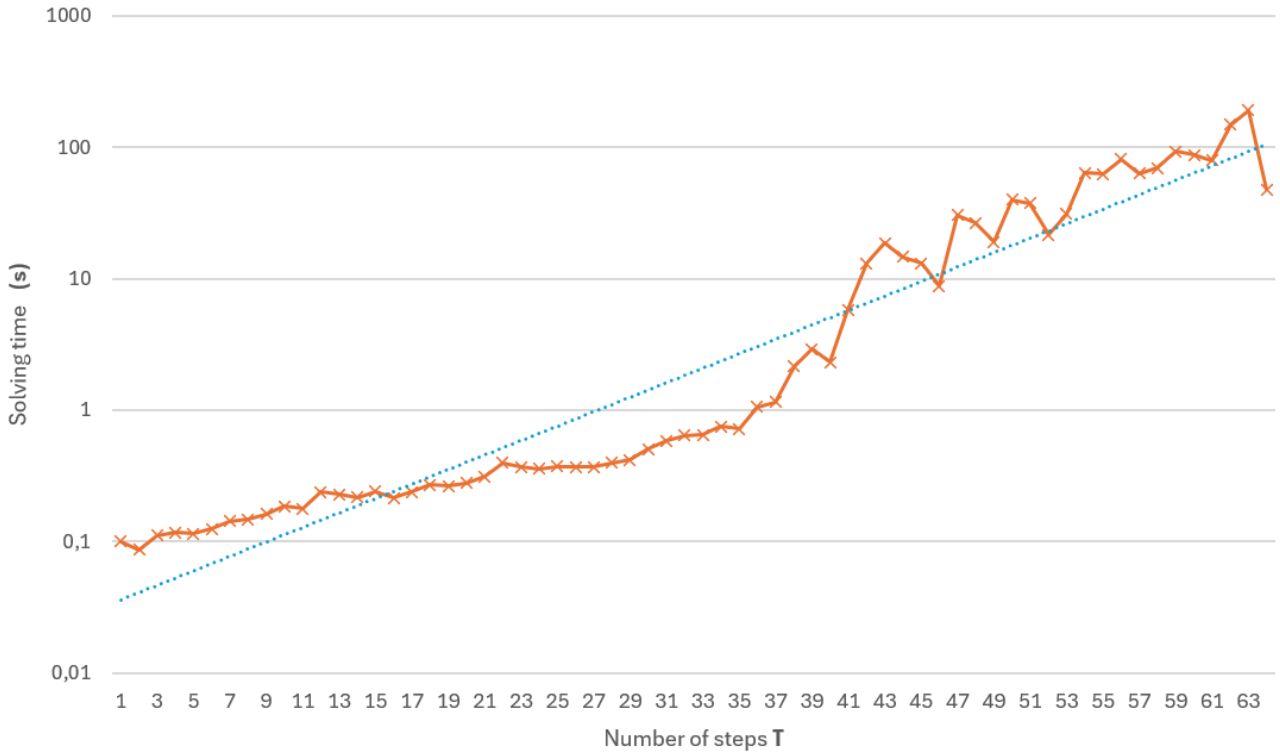


FIGURE 6 – En orange, l'évolution du temps résolution du niveau 12 du jeu original en fonction du nombre de mouvements $\boldsymbol{T}$. En bleu, la courbe de tendance exponentielle associée. Pour $\boldsymbol{T} < 65$, aucune solution n'est trouvée. Pour $\boldsymbol{T} = 65$, une solution est trouvée.

Pour des instances plus complexes, le temps de calcul deviendra rapidement très long. Dans ces cas, il pourrait être intéressant d'implémenter des algorithmes de recherche de solution non déterministes mais plus rapides, tels que des algorithmes heuristiques ou évolutionnaires.