DEPARTEMENT INFORMATIQUE
DE LA FACULTE DES SCIENCES

Université de Montpellier - Licence 3 Informatique

TER

Wumpus World

Auteurs :
Thomas Lemaitre
Bérénice Lemoine
Clément Manniez
Olivier Montet

Encadrante :
M^me^ Marie-Laure Mugnier

Année 2017-2018

Remerciements

Nous tenons à remercier notre tutrice, madame Marie-Laure Mugnier pour l'aide précieuse qu'elle nous a apporté tout au long de notre projet. Ses conseils et son écoute nous ont permis de surmonter les difficultés que nous avons pu rencontrer au cours de notre travail.

Table des matières

1 Introduction

Dans le cadre de notre TER de 3ème année en Licence informatique, nous avons eu pour projet l'élaboration d'un jeu basé sur le monde de Wumpus. Ce monde est composé d'une grille dans laquelle le joueur évolue. Son but : trouver l'or tout en restant en vie. Pour se faire il doit se déplacer en évitant de tomber dans un trou ou de rencontrer le terrible Wumpus, qui le mangerait ! Afin d'éviter une fin tragique le joueur doit se servir des perceptions ressenties sur les cases adjacentes aux dangers. Autour d'un trou on a la présence de brises de vent, et autour du Wumpus, de puanteurs. Le personnage a en sa possession une flèche, qu'il peut utiliser quand il le désire. Si sa flèche atteint le Wumpus, ce dernier meurt et n'est plus un danger. En cas d'échec, la flèche sera perdue et plus aucun autre tir ne sera permis. Sur les cases adjacentes à l'or on perçoit des éclats.

	B	
B	T	B
	B	

TABLE 1.1 – Placement brises/trou

	P	
P	W	P
	P	

TABLE 1.2 – Placement puanteurs/Wumpus

L'objectif de notre projet est de créer une version jouable du monde de Wumpus et d'être en mesure de générer des grilles "intéressantes". Intuitivement, une grille peut être dite intéressante si le joueur peut atteindre l'or sans prendre de risque en exploitant au maximum les déductions permises par ses perceptions, tout en n'étant pas "trop facile" à résoudre non plus (typiquement, s'il existe un chemin menant à l'or sans aucune perception sur le chemin, il n'y a aucune déduction à faire).

Prenons un exemple de déduction pouvant être faite par un joueur. Dans les schémas ci-dessous sont représentées en gris les cases non explorées et en blanc celles explorées. Lors de la première étape le joueur, qui était sur la case 1 arrive sur une case où il perçoit une brise. Il ne sait pas où peut se situer le trou parmi les cases autour de la perception. Pour ne pas prendre de risque il revient sur la case précédente puis monte d'une case, et ne détecte pas de perceptions. La case 2 est donc sûre. Puis il fait de même avec la case en bas de la case 1, qui est également vide : pas de trou en 4 non plus. Par conséquent le trou est forcément en position 3.

	2	
1	B	3
	4	

TABLE 1.3 – Étape 1

	2	
1	B	3
	4	

TABLE 1.4 – Étape 2

	2	
1	B	3
	4	

TABLE 1.5 – Étape 3

Nous devons également modéliser ce monde sous forme de formules de la logique des propositions. D'une part une base de connaissances contiendra toutes les lois du monde, d'autre part on enregistrera toutes les perceptions ressenties lors du parcours. Ceci permet l'utilisation d'un solveur SAT grâce auquel on pourra faire des inférences et en déduire si une case comporte un risque ou non. Ceci nous servira à caractériser les grilles intéressantes, mais aussi à aider le joueur en lui proposant des informations déductibles sur les cases qui l'entourent.

Dans un premier temps, nous présenterons brièvement les domaines de l'informatique centraux pour notre projet : la logique des propositions et les interfaces graphiques basées sur la programmation événementielle. Puis nous détaillons les trois grandes parties de notre travail :

— la gestion de la base de connaissances et des déductions
— la génération de grilles intéressantes
— la conception logicielle

2 Domaines de l'informatique

2.1 La logique

Application de la logique au monde de Wumpus

Le monde de Wumpus est un jeu qui appelle à la logique du joueur, sa réussite dépend entièrement de la justesse de ses déductions. En effet pour gagner la partie le joueur doit éviter de tomber dans un trou ou de se faire manger par le Wumpus. Pour cela il doit utiliser les informations à sa disposition. Celles-ci se regroupent en deux parties : les règles qui régissent le monde (par exemple : il y a présence de brises autour d'un trou) et les perceptions constatées au cours du jeu (par exemple : "je ressens une brise dans la case[i,j]").

Règles du monde et perceptions

Comme évoquées précédemment, les règles du monde sont constituées de toutes les mécaniques du jeu. Par définition elles restent les mêmes d'une partie à l'autre. Elles constituent une première base de connaissances du monde. Pour pouvoir en tirer des informations lors d'une partie il faut lui ajouter les nombreuses sensations que perçoit le joueur. À partir de cet ensemble de faits nous souhaitons que notre programme puisse déterminer si une case est sûre, s'il y a un risque, ou s'il est certain qu'elle comporte un danger.

Problème SAT et solveurs

Le problème SAT, pour Satisfaisabilité booléenne est un problème de l'informatique théorique s'appuyant sur la logique des propositions. À partir d'une formule de logique propositionnelle, constituée d'un ensemble de littéraux (c'est à dire de variables booléennes ou de leurs négations) et de connecteurs logiques, il s'agit de déterminer s'il existe une assignation des variables à vrai ou faux qui rend la formule vraie. Une telle assignation est appelée modèle.

Ce type de problème peut servir à la résolution de jeux comme le sudoku. Nous allons, pour notre part, nous en servir pour permettre les déductions dans le monde de Wumpus.

Un programme qui permet de résoudre des problèmes SAT est appelé solveur SAT. Ce type de programme a connu de grandes avancées au cours de l'histoire de l'informatique. Ceci a rendu les solveurs SAT de plus en plus performants et intéressants. Pour la réalisation de ce projet nous utilisons la bibliothèque SAT4J, un logiciel libre de l'Université d'Artois et du CNRS développé au CRIL (laboratoire d'informatique de Lens). `http://www.sat4j.org/`

2.2 Java : interface graphique

Intérêt de l'interface graphique

Le monde de Wumpus étant un jeu assez ancien : les premières versions de celui-ci se jouaient sur un terminal et le monde était représenté à l'aide de caractères ASCII voire même entièrement décrit en mode texte.
Une interface graphique permet cependant une meilleure lisibilité du jeu et rend plus agréable son utilisation. Basée sur AWT, la bibliothèque graphique SWING est la plus connue en Java. Les éléments graphiques qu'elle propose (panneaux, boutons, insertion d'image) vont nous permettre de créer l'interface graphique de notre jeu.

Programmation événementielle

La programmation événementielle est un paradigme de programmation dans lequel on définit des réactions à différents événements. Ces événements peuvent être un clic de souris ou encore l'appui sur une des touches du clavier.
Elle est indissociable de l'interface graphique d'un jeu. En effet c'est elle qui permet l'interaction entre le joueur et le programme. Lorsque le joueur décide d'effectuer une action, il utilise son clavier et sa souris, en réponse le programme exécute le code correspondant à l'événement déclenché.

3 Base de connaissances

3.1 De la formule à la clause

Dans un premier temps notre but est de faire connaître au solveur les règles qui régissent le monde du Wumpus. Les informations exploitées par SAT4J sont des formules de la logique propositionnelle sous forme d'une conjonction de clauses. Une clause étant une disjonction de littéraux (variables propositionnelles ou leurs négations), autrement dit, elle est composée d'un ensemble de littéraux reliés par un seul type de connecteur logique : le OU. Pour que le problème soit satisfiable, le solveur doit trouver un modèle qui satisfasse toutes les clauses.
Une première tâche consiste à formaliser les règles du monde. Nous utilisons ici la logique des prédicats sur un exemple car elle est plus succincte que la logique des propositions. Mais le solveur SAT n'acceptant que des formules de la logique des propositions, il nous faut ensuite passer des formules de la logique des prédicats à des formules propositionnelles. (vous pouvez retrouver toutes les règles modélisées en annexe).

Les prédicats ci-dessous gèrent ce qui peut se trouver en une case donnée. Des simples perceptions au Wumpus, ils nous permettent de définir les règles du monde (l'or ne doit pas être dans un trou, la première case est vide...).

- B(x,y) Vrai s'il y a une brise dans la case aux coordonnées x :y
- G(x,y) Vrai s'il y a de l'or dans la case aux coordonnées x :y
- P(x,y) Vrai s'il y a une puanteur dans la case aux coordonnées x :y
- T(x,y) Vrai s'il y a un trou dans la case aux coordonnées x :y
- E(x,y) Vrai s'il y a un éclat dans la case aux coordonnées x :y
- W(x,y) Vrai s'il y a le Wumpus dans la case aux coordonnées x :y
- estDifferent(x,y,z,t) Vrai si $x \neq z$ ou $y \neq t$

Le dernier prédicat est nécessaire pour garantir que l'or et le Wumpus sont uniques.

Nous avons pensé à un autre prédicat pour savoir si une case n'est pas à l'extérieur de notre grille. Cependant la modélisation en formule logique devient dans ce cas trop complexe et ce prédicat est alors facilement remplaçable lors de la programmation que nous verrons plus tard.

Une grille est utilisable si et seulement si elle respecte toutes les règles du monde. Prenons pour exemple la formule qui garantit qu'un trou soit entouré de brises.

$\forall x \forall y$ (T(x,y) $\Rightarrow$ (B((x-1),y) $\wedge$ B((x+1),y) $\wedge$ B(x,(y-1)) $\wedge$ (B(x,(y+1))))

Elle est littéralement traduisible par : quels que soient x et y, si la case aux coordonnées x :y possède un trou, alors toutes ses cases adjacentes possèdent une brise. Pour mettre cette formule sous forme clausale il nous faut dans un premier temps la transformer.

$\forall x \forall y$ (($\neg$T(x,y) $\vee$ B((x-1),y)) $\wedge$ ($\neg$T(x,y) $\vee$ B((x+1),y)) $\wedge$ ($\neg$T(x,y) $\vee$ B(x,(y-1))) $\wedge$ ($\neg$T(x,y) $\vee$ B(x,(y+1))))

Sur la formule de l'exemple nous obtenons une forme clausale composée de 4 clauses.

Dans un second temps il nous faut obtenir une formule de la logique propositionnelle, c'est à dire remplacer les prédicats par des littéraux propositionnels caractérisant une observation à une case x :y (qui doit exister). Par exemple, si nous instancions x et y par 0 nous obtenons les deux clauses propositionnelles ($\neg$ T(0,0) $\vee$ B(1,0) et ($\neg$ T(0,0) $\vee$ B(0,1)). Cette étape sera effectuée lors de la génération.

3.2 Génération et stockage des connaissances

L'objectif est de faciliter l'accès et la manipulation d'une base de connaissances. Parmi les méthodes proposées par SAT4J nous nous sommes intéressés à :

```
static ISolver newLight()
```

Cette méthode static de la classe SolverFactory renvoie une instance de type ISolver. L'instance créée va permettre le stockage des clauses et sera à même d'effectuer des tests de satisfiabilité. Celles-ci seront ajoutées grâce à la méthode addClause ci-dessous.

```
IConstr addClause(IVecInt literals) throws ContradictionException
```

AddClause prend en paramètre une clause sous forme d'une instance de type IvecInt. Il s'agit d'un tableau d'entiers où chaque élément représente un littéral.

Une question se pose alors, comment allons nous représenter les littéraux sous forme d'entiers ?

Chaque littéral devra porter 3 informations :

— Le type de fait qu'il décrit (brise, Wumpus...)
— L'abscisse de la case
— L'ordonnée de la case

Ces informations doivent être encodées sous forme d'un entier, qu'on devra ensuite pouvoir décoder pour retrouver les données initiales.

On peut modéliser cela en 3 dimensions en mettant en place différentes couches de la matrice, chacune représentant un type de fait. Par exemple la couche 1 représentera les brises. On numérote toutes les cases de la première couche puis on continue la numérotation sur la seconde et ainsi de suite... La négation d'un type de fait est représentée par l'entier obtenu, précédée du signe "-".

Ainsi on peut calculer l'entier correspondant à la formule suivante :

$$((l-1)*(n*n))+((y)*n)+(x+1)$$

où x et y sont les coordonnées de la case d'une grille de taille n*n, et l le numéro attribué au littéral souhaité (Brise = 1, or = 2, puanteur = 3, trou = 4, éclat = 5 et Wumpus = 6). Par exemple, la clause "Dans une grille de taille 4*4, il y a dans la case 3 :0 aucun Wumpus ou un trou" sera modélisée par le tableau d'entiers suivant :

{-93, 61}

Nous possédons maintenant tous les éléments pour générer une base de connaissances qui implémente nos formules logiques (voir annexe). Reprenons notre exemple "un trou doit être entouré de brises". Nous l'avons modélisé ainsi dans la partie 3.1 :

$\forall x \forall y$ ($(\neg T(x,y) \vee B((x-1),y)) \wedge (\neg T(x,y) \vee B((x+1),y)) \wedge (\neg T(x,y) \vee B(x,(y-1))) \wedge (\neg T(x,y) \vee B(x,(y+1)))$)

La génération va construire des clauses pour toutes les cases de la grille et prendre en compte leurs positions. Par exemple, pour la case en position 0 :0 (donc qui ne possède que deux voisins 1 :0 et 0 :1), le code pour générer les clauses de notre exemple serait :

```
int t=getLiteralCode(0,0,n,'t');          //centre
int b1=getLiteralCode(0,1,n,'b');         //nord
int b2=getLiteralCode(1,0,n,'b');         //est
int clause1[]={-t,b1};
int clause2[]={-t,b2};
try{
    BC.addClause(new VecInt(clause1));
    BC.addClause(new VecInt(clause2));
}
catch (ContradictionException E1){
    E1.printStackTrace();
}
```

Les sensations qu'a détectées le joueur pendant la partie sont quant à elles représentées par de simples clauses unitaires (réduites à un seul littéral). Effectivement, si le joueur détecte une brise, cela signifie que tout modèle doit rendre ce littéral vrai pour valider la clause unitaire.

Il ne nous reste plus qu'à créer une classe qui représente la base de connaissances. Toute instance de cette classe comporte l'instance ISolver que nous a renvoyé la méthode Générer, et un booléen qui indique si le Wumpus est en vie.

3.3 Faire des déductions

On veut maintenant utiliser le solveur SAT fourni par SAT4J pour questionner notre base de connaissances. C'est à dire, vérifier s'il existe un modèle qui valide un littéral.

Par exemple, le joueur veut savoir si la case qu'il souhaite explorer est exempte de trou. La démarche pour déduire ceci est la suivante :

1. On cherche à vérifier {C1, ... Cn} ¬(Trou). C'est à dire si l'absence de trou (¬(Trou)) est conséquence logique de l'ensemble des clauses de la base de connaissances ({C1, ... Cn}).
2. Ceci revient à vérifier si C1 ∧ ... Cn ∧ Trou est insatisfiable.
3. On ajoute donc temporairement à la base de connaissances la négation de ce que l'on veut vérifier. Puis on interroge le solveur pour savoir si le tout est satisfiable.

Si le solveur répond que l'ensemble est insatisfiable notre conséquence logique est vraie, on peut donc affirmer qu'il n'y a pas de trou sur cette case.

Sinon l'ensemble est satisfiable, cela signifie qu'avec les connaissances actuelles il pourrait aussi bien y avoir un trou que de ne pas en avoir un. Pour préciser notre déduction on peut tester avec trou comme conséquence logique : {C1, ... Cn} (Trou). Si Trou est conséquence logique on saura qu'il y a un trou. Si l'ensemble est insatisfiable on restera dans l'incertitude.

La méthode question prend en entrée un littéral l et renvoie 1 si l est conséquence logique de la base de connaissance, 0 si ¬ l est conséquence logique et 2 si l'on ne sait pas. Elle est implémentée de la façon suivante :

```
int clauseNeg[]={-l};
int clausePos[]={l};
IProblem problem = SolveBc(); //Solveur
try {
    //C1, ... Cn |= l
    if (!problem.isSatisfiable(new VecInt(clauseNeg))) {
            present=true;
    }
    //C1, ... Cn |= !l
    if (!problem.isSatisfiable(new VecInt(clausePos))) {
            absent=true;
    }
}
catch (TimeoutException e) {
```

```
    e.printStackTrace();
}
if(present == absent){
    return 2;
}
else{
    if(present){ return 1;}
    else{ return 0;}
}
```

Les deux dangers existants dans le jeu sont les trous et le wumpus. Pour vérifier si une case est sûre on utilise la procédure question pour chacun d'eux. En fonction des résultats obtenus on pourra dire que la case est sûre, qu'elle comporte un danger ou qu'elle en comporte peut-être un. Le wumpus ne sera vérifié que s'il est en vie (d'où l'intérêt de garder en mémoire son état).

4 Génération grille

4.1 Concept des grilles

4.1.1 Une première idée

Notre première idée consistait à générer une grille que l'on pourrait qualifier de pseudo-aléatoire. On plaçait le wumpus et l'or à des positions aléatoires dans notre grille, en interdisant la case de départ (0,0), et les cases adjacentes à celle-ci ((0,1) et (1,1)) pour éviter que le joueur ne gagne ou ne meurt dès son premier mouvement. Ensuite on ajoutait aléatoirement les trous, dont on définissait le nombre par T/3 (avec T : la taille de la grille). Le problème est que les grilles étaient soit infaisables soit avec beaucoup de risques, ce qui ne nous convenait pas. Nous avons donc réfléchi à une façon de positionner les trous de sorte que les grilles soient réalisables et intéressantes.

Nous avons constaté que si les trous étaient trop proches les uns des autres, alors les sensations autour (ici brises) se retrouvent soit les unes sur les autres soit bloquent le chemin. Le joueur est donc obligé de prendre un risque pour avancer. Afin d'éviter cela nous avons cherché à espacer les trous. Nous avons trouvé que si l'on a (X+Y) ≥ 5 (avec X la distance en abscisse entre le premier trou t1 et le deuxième trou t2, et Y la distance en ordonnée entre t1 et t2), alors les sensations ne bloquaient pas le chemin du joueur. Mais cette solution impose énormément de contraintes et les grilles créées seraient toutes assez ressemblantes. Nous avons donc abandonné cette solution.

	B				
B	T	B			
	B			B	
			B	T	B
				B	

TABLE 4.1 – Positionnement des trous avec un écart de 5 (Tracé bleu)

4.1.2 Génération de grille laissant un chemin

Pour pouvoir tester notre jeu nous avions besoin de grilles dans lesquelles on peut atteindre l'or sans risque. La solution la plus simple a été de laisser un chemin sûr qui ne contient pas de perception.

Pour cela on générait dans un premier temps un chemin reliant le départ à l'or. L'enchaînement de mouvements en haut et à droite était ici aléatoire. On élargissait ensuite le chemin sur les cases adjacentes. L'ensemble des cases que l'on vient de citer étaient retirées d'une liste et on plaçait les trous sur les cases restantes.

Si les grilles ainsi générées ont permis de nombreux tests elles demeurent peu intéressantes pour un joueur.

4.1.3 Vers une sélection des grilles intéressantes

Nous venons de voir qu'il était complexe de générer des grilles qui soient intéressantes. Mais si nous ne pouvons pas directement les générer nous pouvons toujours sélectionner parmi un grand nombre de grilles celles qui correspondent à nos critères. Nous allons donc évaluer les grilles sur plusieurs critères pour sélectionner celles qui correspondent au mieux à nos attentes.

4.2 Évaluation des grilles

Notre choix final s'est porté sur une génération aléatoire de grilles avec seulement quelques conditions préalables :

— aucun danger ni aucune sensation en (0,0), (0,1) et (1,0)

— le wumpus et l'or ne sont pas sur une case où il y a un trou.

Après la génération d'une grille aléatoire, nous la testons à l'aide des fonctions *grilleValide, grilleDeductible et grilleInteressante* que l'on a créées.

4.2.1 Valide

Une grille est dite valide s'il existe une possibilité d'atteindre l'or et le wumpus. Un exemple de grille non valide serait une grille sur laquelle une rangée de trous sépare le joueur de ses objectifs. Il serait alors impossible de gagner.

Pour déterminer si une grille est valide on la transpose sous forme de graphe. Chaque case n'étant pas un trou correspond à un sommet. Les arêtes représentent les cases voisines dans la grille .

Une fois sous forme de graphe on regarde s'il existe un chemin entre la position de départ (0,0), l'or et le wumpus. Autrement dit on regarde s'ils sont dans la même composante connexe.

Si une grille est valide, il existe donc au moins un chemin qui permet d'arriver à l'or et au wumpus. Cependant ce chemin peut comporter des prises de risques ne pouvant être évitées, même en faisant des déductions. Pour mettre en avant la logique plutôt que la chance nous allons chercher à détecter si une grille est faisable sans prendre de risques.

Algorithme 1 : grilleValide

Données: g une Grille
Résultat: Vrai si le wumpus, l'or et le (0,0) appartiennent à la même composante connexe, faux sinon.

```
début
    xwumpus ← abscisse du wumpus
    ywumpus ← ordonnée du wumpus
    xor ← abscisse de l'or
    yor ← ordonnée de l'or
    AExplorer ← ajouterPoint(0, 0) ; //Liste de points à explorer
    dejaVu[g.getTaille()][g.getTaille()] ← faux ; // Tableau de booléens (Vrai si le
     point i,j à été exploré)
    tant que AExplorer n'est pas vide ET (dejaVu[xwumpus][ywumpus] == faux
     OU dejaVu[xor][yor] == faux ) faire
        courant ← Premier élément de AExplorer
        dejaVu[courant.getX()][courant.getY()] ← vrai
        Retirer l'élement courant de AExplorer
        voisins[] ← voisin(g.getTaille(), courant)
        pour p in voisins[] faire
            x ←p.getX()
            y ←p.getY()
            si dejaVu[x][y] == false) alors
                si Il n'y a pas de trou en g.get(x,y) alors
                    si p n'appartient pas à AExplorer alors
                        AExplorer ← Ajouter p
                sinon
                    dejaVu[x][y] ← vrai
    si dejaVu[xwumpus][ywumpus] == dejaVu[xor][yor] == dejaVu[0][0] alors
        retourner vrai ;
    sinon
        retourner faux ;
```

4.2.2 Déductible

Une grille est dite déductible s'il existe un chemin pour arriver jusqu'à l'or sans avoir à prendre de risques. C'est-à-dire, que le joueur ne peut pas se retrouver complètement bloqué par des brises.

	B			E	O
B	T	B	B		E
	B	B	T	B	B
			B	B	T
					B

TABLE 4.2 – Formation d'un bloc de brise (IMPOSSIBLE)

Ici, par exemple, on voit sur le schéma que le joueur est obligé de prendre un risque (celui de tomber dans un trou) pour gagner.

L'algorithme grilleDéductible renvoie vrai si la grille est réalisable sans prendre de risques, faux sinon. Pour cela il utilise le solveur SAT et la base de connaissances.

Le fait qu'une grille soit déductible n'indique cependant pas la difficulté qu'on pourra avoir lors d'une partie. Certaines grilles déductibles peuvent être très simples et d'autres représenter une difficulté élevée. Il convient donc, parmi ces grilles de garder celles qui ont le plus d'intérêt pour le joueur.

Algorithme 2 : grilleDéductible

Données: g une grille valide
Résultat: Vrai si elle est déductible, faux sinon (déduction impossible).

```
début
    bc ← nouvelle base de connaissances pour une grille de taille n
    deductionImpossible ← 0
    dv ← une matrice de booléens initialisée à false
    file ← une nouvelle file possédant des éléments de type Point
    p ← un Point aux coordonnées 0 :0
    Ajouter p à file
    tant que file n'est pas vide faire
        //on teste si le nombre de cases sur lesquelles nous sommes bloqués est supérieur au nombre de cases
          stockées dans la file
        si deductionImpossible est supérieur aux nombre d'éléments dans file alors
            retourner faux ;
        p ← premier élément de file
        Retirer à file son premier élément
        x ← abscisse de p
        y ← ordonnée de p
        c ← la case de la grille g aux coordonnées x, y
        si c possède une brise alors
            Ajouter l'information brise à bc ;
        sinon
            Ajouter l'information non brise à bc ;
        ... faire pareil pour l'éclat et la puanteur...
        si c possède l'or alors
            retourner vrai
        sinon
            Ajouter l'information non or à bc ;
        voisinEncoreAExplorer ← faux
        indeductible ← vrai
        pour les voisins v de p faire
            x ← abscisse de v
            y ← ordonnée de v
            si dv[x][y] = faux alors
                suivant danger dans la case [x][y] faire
                    /
                /il y a un danger
                cas où 0 faire
                    dv[x][y] ← vrai
                    indeductible ← faux
                //il n'y a pas de danger
                cas où 1 faire
                    dv[x][y] ← vrai
                    indeductible ← faux
                    si v n'est pas dans pile alors
                        Ajouter v à pile
                cas où 2 faire
                    voisinEncoreAExplorer ← Vrai
        si voisinEncoreAExplorer alors
            Ajouter p à pile
        si indeductible alors
            deductionImpossible++
        sinon
            deductionImpossible ← 0
```

4.2.3 Intéressante

Une grille est dite intéressante si elle demande une certaine réflexion de la part du joueur. C'est-à-dire qu'il y a des déductions à faire pour atteindre l'or sans prendre de risques.

L'algorithme grilleInteressante renvoie un nombre entier, plus ce nombre est grand, plus la grille demande de réfléchir.

Cet algorithme transforme la grille passée en paramètre en un graphe orienté pondéré. S'il n'y a aucune sensation sur un sommet alors toutes les arêtes partant de ce sommet ont un poids de 0, sinon elles prennent un poids de 1. GrilleInteressante fonctionne un peu comme l'*Algorithme de Dijkstra* : on recherche sur le graphe construit le plus court chemin reliant l'origine (0,0) et l'or. On ne prend pas en compte les sommets correspondant à des trous.

Algorithme 3 : grilleInteressante

Données: g une grille valide de taille n*n, *or* le Point ou est situé l'or dans g
Résultat: Le nombre de perceptions minimum sur un parcours pour atteindre l'or.
début
    pere ← une matrice de Point à laquelle on ajoute le point aux coordonnées [-1][-1]
    d ← une matrice d'entiers initialisés à l'infini
    traite ← une matrice d'entier initialisés à 0
    pour *les cases [x][y] de la grille g qui possèdent un trou* **faire**
        traite[x][y] ← 1
    pere[0][0] ← le point aux coordonnées [0][0]
    pour *i de 0 à n-1* **faire**
        pour *j de 0 à n-1* **faire**
            p ← prend le Point [x][y] tel que traite[x][y]=0 et que sa valeur d[x][y] soit la plus petite
            pour *tous les voisins v de p* **faire**
                z ← abscisse de v
                t ← ordonnée de v
                dist ← 0
                si *dans la case [x][y] il y a une brise ou un puanteur* **alors**
                    dist ← 1
                l ← d[x][y]+*dist*
                si *traite[z][t] == 0 et d[z][t] > 1* **alors**
                    d[z][t] ← 1
                    pere[z][t] ← p

4.3 Optimisation de la génération

Pour récapituler, nous avons donc choisi de pré-générer des grilles de différentes tailles avec un nombre de trous choisi aléatoirement. Ces grilles subissaient trois tests d'évaluation qui nous permettaient de juger leur difficulté. Cette solution est cependant loin d'être optimale.

En effet les grilles non valides sont nombreuses. De plus, si un quota de grilles est atteint (exemple les faciles qui sont globalement plus générées que les moyennes), nous aimerions que le programme se concentre davantage à générer un type de grille précis pour éviter une trop grande perte. Moins cette perte sera importante, plus le programme sera rapide. Il devient donc nécessaire de déterminer quel est le nombre de trous qui maximise la génération d'un certain type de grille.

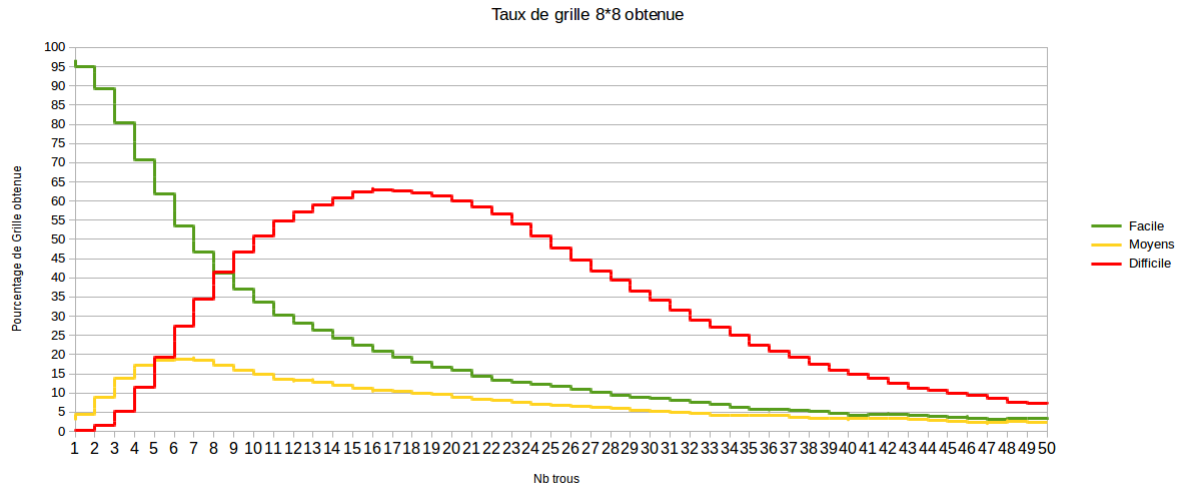
Pour analyser une quelconque corrélation il a fallu générer un grand nombre de grilles de même taille avec un nombre précis de trous (10 000 grilles avec un seul trou, puis 10 000 grilles avec deux trous...). Puis nous avons classé chaque grille suivant deux critères : si la grille est déductible ou indéductible (résultat de la méthode grilleDeductible) et son niveau d'intérêt (résultat de la méthode grilleInteressante).

Par exemple, voici le nombre de grilles déductibles de taille 8*8 pour un nombre de trous et un niveau d'intérêt précis :

Trous \ Intérêt	0	1	2
1	9677	290	0
2	9334	607	0
3	8540	1139	14

Il y a donc 607 grilles déductibles de taille 8*8 qui ne possèdent qu'un seul trou et ont un intérêt à un. Sur les 10 000 générées cela représente 0.6%.

Ainsi, pour mieux visualiser nos bornes, nous avons synthétisé ces résultats sous forme de graphiques :



Ci-dessus, les taux de grilles 8*8 faciles, moyennes et difficiles sont représentées pour un nombre de trous variant entre 1 et 50. Nous constatons par ailleurs que le taux de grilles moyennes ne dépasse pas les 20%. Si nous voulons optimiser leur création il faut borner le nombre de trous entre 5 et 10 par exemple.

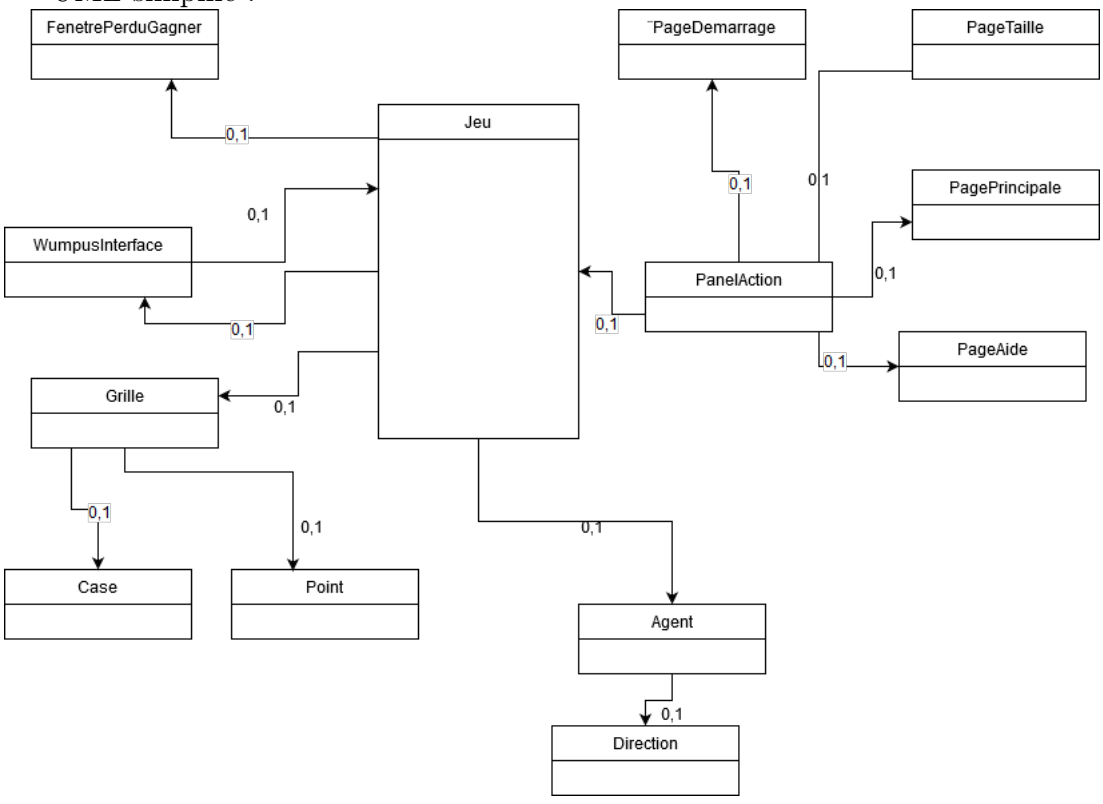
Vous pouvez retrouver les graphiques, les bornes que nous avons choisies et un lien vers toutes nos données en annexe.

5 Conception Logicielle

5.1 UML

Nous nous sommes inspirés de l'approche dite Modèle-Vue-Contrôleur (MVC) :
- le modèle contient toutes les données et les méthodes qui permettent de les manipuler.
- la partie vue est constituée des classes gérant l'interface graphique.
- enfin la partie contrôleur traite les actions de l'utilisateur et modifie les données du modèle et de la vue.

UML simplfié :



[uml]Uml complet en annexe.

Notre partie modèle contient les classes relatives à la génération de grilles et la classe jeu, classe centrale qui réunit toutes les informations de notre projet :

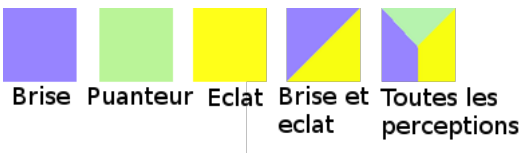
— un objet grille, matrice de cases réunissant toutes les informations relatives à la grille comme la position du Wumpus, celles des perceptions...
— l'objet Agent qui permet de stocker les informations et méthodes relatives à l'agent.
— notre objet Base de Connaissance.

Pour la partie vue, on avons la fenêtre WumpusInterface qui appelle successivement les différents panneaux (instances de JPanel) comme les classes PagePrincipale, PageDemarrage... Ces classes sont appelées en fonction des actions souhaitées et permettent d'afficher les différents boutons.

Enfin la partie contrôleur est constituée de la classe PanelAction qui regroupe les écouteurs d'évènements. Elle détecte les actions de l'utilisateur faites sur l'interface et manipule les informations de la classe jeu.

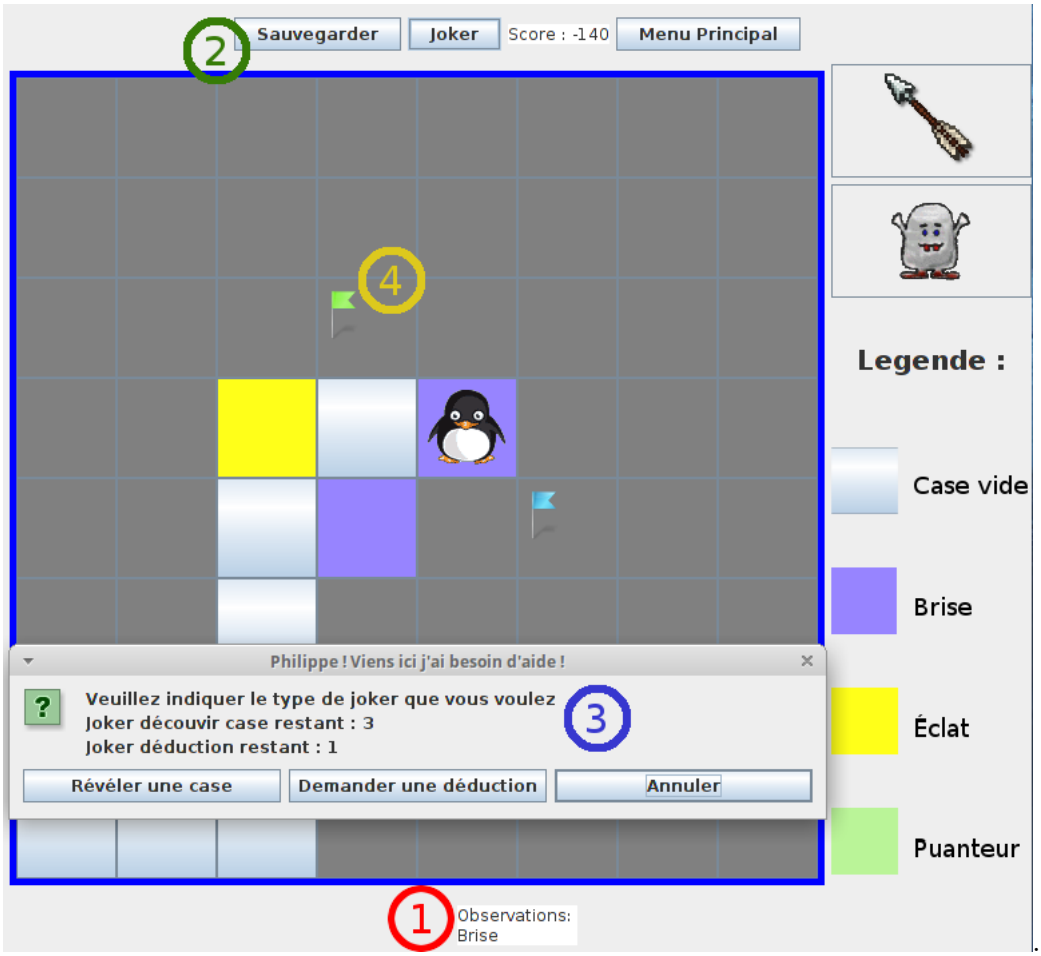
5.2 Implémentation

Dans un premier temps nous avons du créer un plateau de jeu. Pour ce faire nous avons implémenté la grille sous la forme d'un tableau. Elle est constituée de n x n boutons (instances de JButton) qui représentent les différentes cases dans lesquelles évolue le joueur. Ces boutons permettent au joueur d'interagir via la souris et de réaliser différentes actions. Ceci sans nous contraindre à devoir traduire les coordonnées souris en coordonnées de la grille, ce qui facilite la gestion des clics. Un ensemble d'images est également affiché sur les boutons pour représenter les différentes perceptions tout au long de la partie.



Toutes les grilles que le joueur peut utiliser sont stockées dans différents dossiers (en fonction de la taille et de la difficulté). Le fichier est choisi aléatoirement et chargé dans la classe jeu grâce à la dé-sérialisation. Les fonctions données par l'interface Serializable enregistrent les données sous forme de chaînes de caractères.

5.3 Fonctionnalités supplémentaires



1 Durant l'implémentation nous avons constaté que le joueur devait rapidement pouvoir connaître les caractéristiques d'une case (présence de brise...). Nous avons donc mis en place un panneau qui affiche les perceptions de la case dans laquelle se trouve le joueur.

2 Pour pouvoir sauvegarder une partie en cours nous utilisons les fonctions données par l'interface Serializable.

3 Une fonctionnalité importante est celle des Jokers. Nous avons décidé que le joueur pourrait avoir accès à deux types d'aides. Le premier type, "Révéler une case" permet de révéler entièrement n'importe quelle case du jeu, en affichant la perception de la case. Le deuxième type de Joker, "Déduire une case" utilise le solveur Sat pour savoir s'il y a un danger sur une case adjacente au joueur en affichant :

— si elle est sûre ✓.
— s'il y a un danger ⚠.
— si le solveur ne peut pas savoir si la case est sûre ou pas ? .

4 Une dernière fonctionnalité mise en place a été la possibilité de marquer une case non

découverte d'un drapeau en y effectuant un clique droit. Ceci pour permettre au joueur de noter les cases qu'il a déduites comme étant dangereuses.

6 Conclusion

Pour conclure, à l'issue de ce projet nous avons réussi à obtenir un jeu de réflexion fonctionnel réalisé en Java. Il consiste en une interface graphique avec différentes fonctionnalités telles que :

— un bouton d'aide avec les informations sur le jeu
— des jokers (découverte de case/ déduction danger) pour aider le joueur
— la sauvegarde/restauration d'une partie en cours
— un mode d'observation des grilles (pour les programmeurs)

Ce TER nous aura permis d'appliquer nos connaissances en logique et d'utiliser un solveur SAT dans un cadre concret. En s'appuyant sur ce dernier nous avons pu mettre en place des mécanismes de déduction prenant en compte les règles du jeu et les informations acquises en cours de partie. Ce projet nous aura également permis de compléter nos acquis théoriques sur les interfaces graphiques et la programmation événementielle.

Nous avons envisagé d'ajouter des options telles qu'un mode de jeu avec plusieurs wumpus et plusieurs tas d'or à découvrir, mais ceci aurait demandé de revoir toute la partie logique et génération déjà existante. On aurait également pu imaginer une sauvegarde des meilleurs scores afin de créer un classement local auquel les joueurs auraient eu accès. Le programme aurait enregistré différents profils et aurait supprimé les grilles déjà jouées.

7 Annexe

7.1 Formules logiques

Nous utiliserons ici la logique des prédicats pour représenter nos informations. Dans un premier temps il nous faut nommer et définir nos différents prédicats :

- B(x,y) Vrai s'il y a une brise dans la case aux coordonnées x :y
- G(x,y) Vrai s'il y a de l'or dans la case aux coordonnées x :y
- P(x,y) Vrai s'il y a une puanteur dans la case aux coordonnées x :y
- T(x,y) Vrai s'il y a un trou dans la case aux coordonnées x :y
- E(x,y) Vrai s'il y a un éclat dans la case aux coordonnées x :y
- W(x,y) Vrai s'il y a le wumpus dans la case aux coordonnées x :y
- estDifferent(x,y,z,t) Vrai si x $\neq$ z ou y $\neq$ t

Il ne nous reste plus qu'à formaliser les règles du monde et les modéliser.
Soit une grille g de taille n*n, prenons D comme domaine d'interprétation tel que D est l'ensemble des naturels compris entre 0 et n-1.
Notons i, j correspondant à des éléments de ce domaine

- Il n'y a qu'un seul wumpus dans ce monde. Donc il existe une case i :j qui possède le wumpus et quelle que soit la case k :l, si ce sont des cases différentes alors il n'y a pas de wumpus dans la case k :l
  $\exists i\ \exists j\ \forall k\ \forall l\ (W(ij) \wedge (\ EstDifferent(i,j,k,l) \Rightarrow \neg W(k,l)))$

- Il n'y a qu'un tas d'or dans ce monde. Donc il existe une case i :j qui possède de l'or et quelle que soit la case k :l, si ce sont des cases différentes alors il n'y a pas d'or dans la case k :l
  $\exists i\ \exists j\ \forall k\ \forall l\ (G(ij) \wedge (\ EstDifferent(i,j,k,l) \Rightarrow \neg G(k,l)))$

- Il n'y a ni or, ni danger, ni perception dans la première case (i = 0, j = 0)
  $\neg W(0,0) \wedge \neg B(0,0) \wedge \neg T(0,0) \wedge \neg G(0,0) \wedge \neg P(0,0) \wedge \neg E(0,0)$

- S'il y a un wumpus en i :j alors ses cases adjacentes ont une puanteur.
  $\forall i\ \forall j\ (\ W(i,j) \Rightarrow (\ P((i-1),j) \wedge P((i+1),j) \wedge P(i,(j-1)) \wedge (P(i,(j+1)))\ ))$

— S'il y a une puanteur en i :j alors une et une seule de ses cases adjacentes possède le wumpus.
$\forall i \, \forall j \, (P(i,j) \Rightarrow (W((i-1),j) \vee W((i+1),j) \vee W(i,(j-1)) \vee W(i,(j+1))))$

— S'il y a un trou en i :j alors ses cases adjacentes ont une brise.
$\forall i \, \forall j \, (T(i,j) \Rightarrow (B((i-1),j) \wedge B((i+1),j) \wedge B(i,(j-1)) \wedge (B(i,(j+1))))$

— S'il y a une brise en i :j alors il y a au moins une case adjacente qui possède un trou.
$\forall i \, \forall j \, (B(i,j) \Rightarrow ((T((i-1),j) \vee T((i+1),j)) \vee (T(i,(j-1)) \vee T(i,(j+1)))))$

— S'il y a l'or en i :j alors ses cases adjacentes ont un éclat brillant.
$\forall i \, \forall j \, (G(i,j) \Rightarrow (E((i-1),j) \wedge E((i+1),j) \wedge E(i,(j-1)) \wedge E(i,(j+1))))$

— S'il y a un éclat brillant en i :j alors une et une seule de ses cases adjacentes possède l'or.
$\forall i \, \forall j \, (E(i,j) \Rightarrow (G((i-1),j) \vee G((i+1),j) \vee G(i,(j-1)) \vee G(i,(j+1))))$

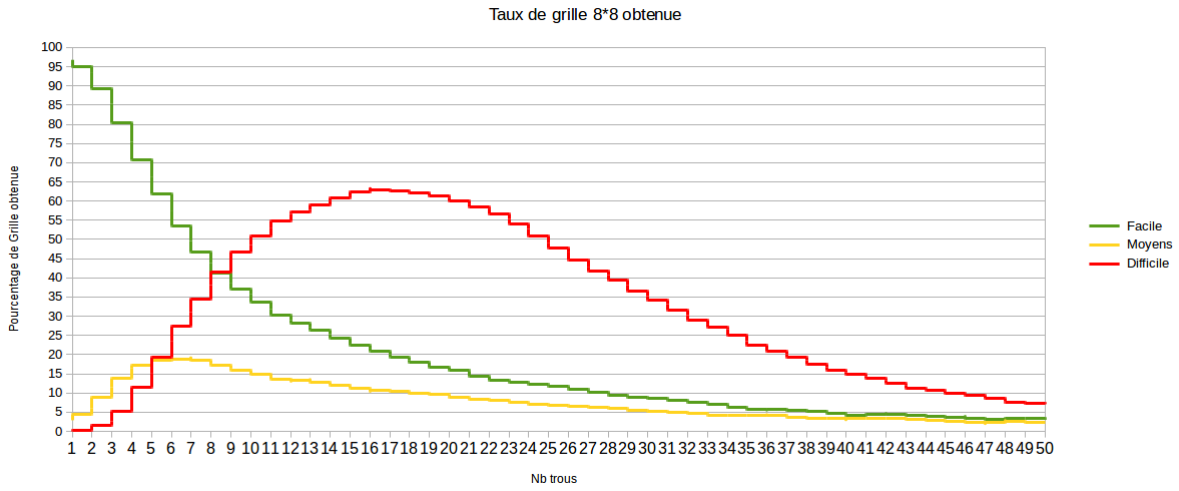
— S'il y a un trou en i :j alors il n'y a pas le wumpus en i :j.
$\forall i \, \forall j \, (T(i,j) \Rightarrow \neg W(i,j))$

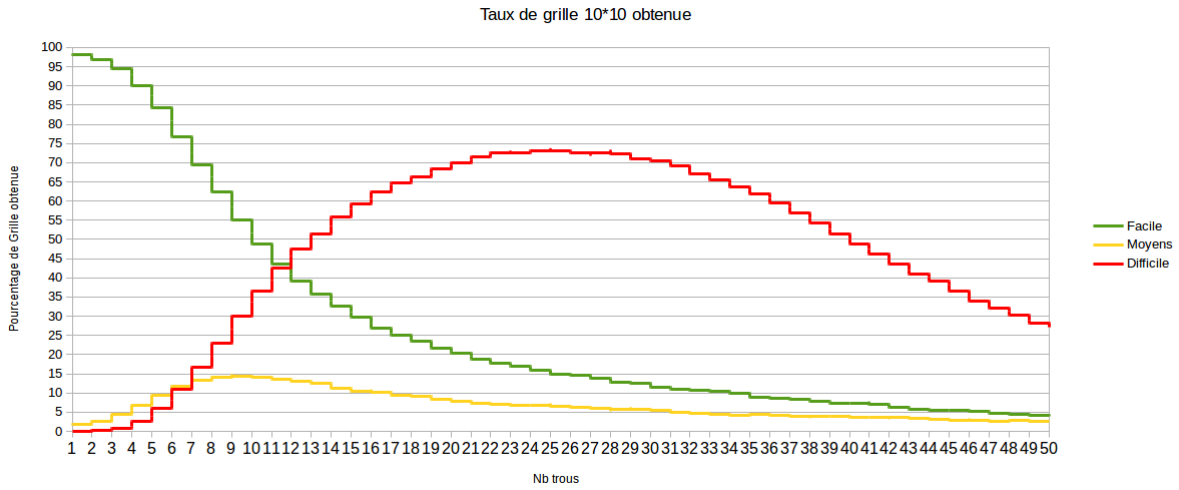
— S'il y a un trou en i :j alors il n'y a pas l'or en i :j.
$\forall i \, \forall j \, (T(i,j) \Rightarrow \neg G(i,j))$

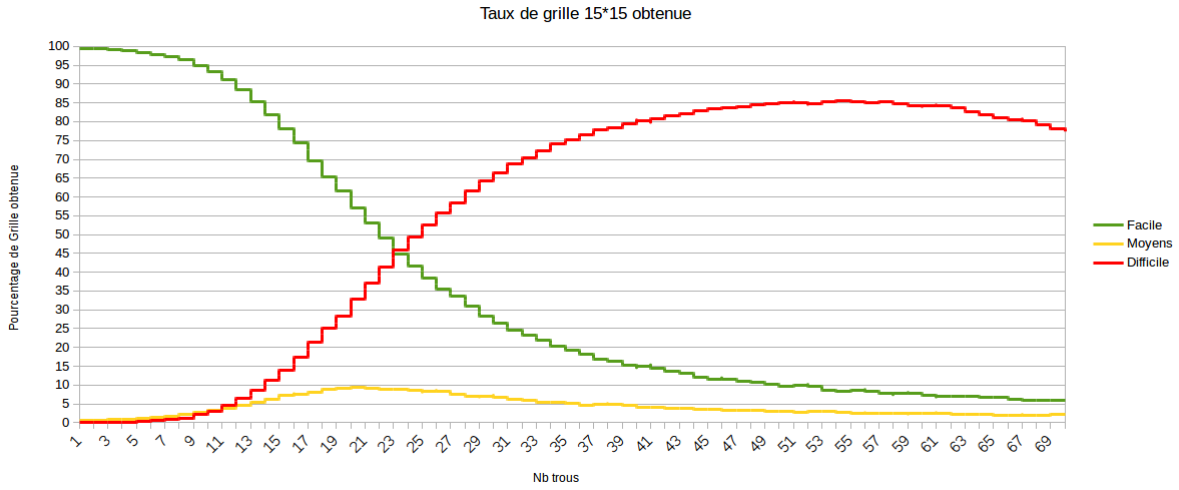
— Regroupement des 2 formules précédentes :
$\forall i \, \forall j \, (T(i,j) \Rightarrow (\neg W(i,j) \wedge \neg G(i,j)))$

7.2 Graphiques et statistiques

Vous trouverez ici trois graphiques. Un pour chaque taille de grilles qui décrit le taux de grilles faciles, moyennes, difficiles générées pour un nombre précis de trous.







Remarque : Pour les grilles 15*15 nous sommes allé jusqu'à 70 trous car sinon nous ne pouvions pas observer (avec seulement 50 trous) la décroissance de courbe des difficiles.

Ces informations vont nous permettre de borner le nombre de trous pour maximiser chacun de nos cas suivants :

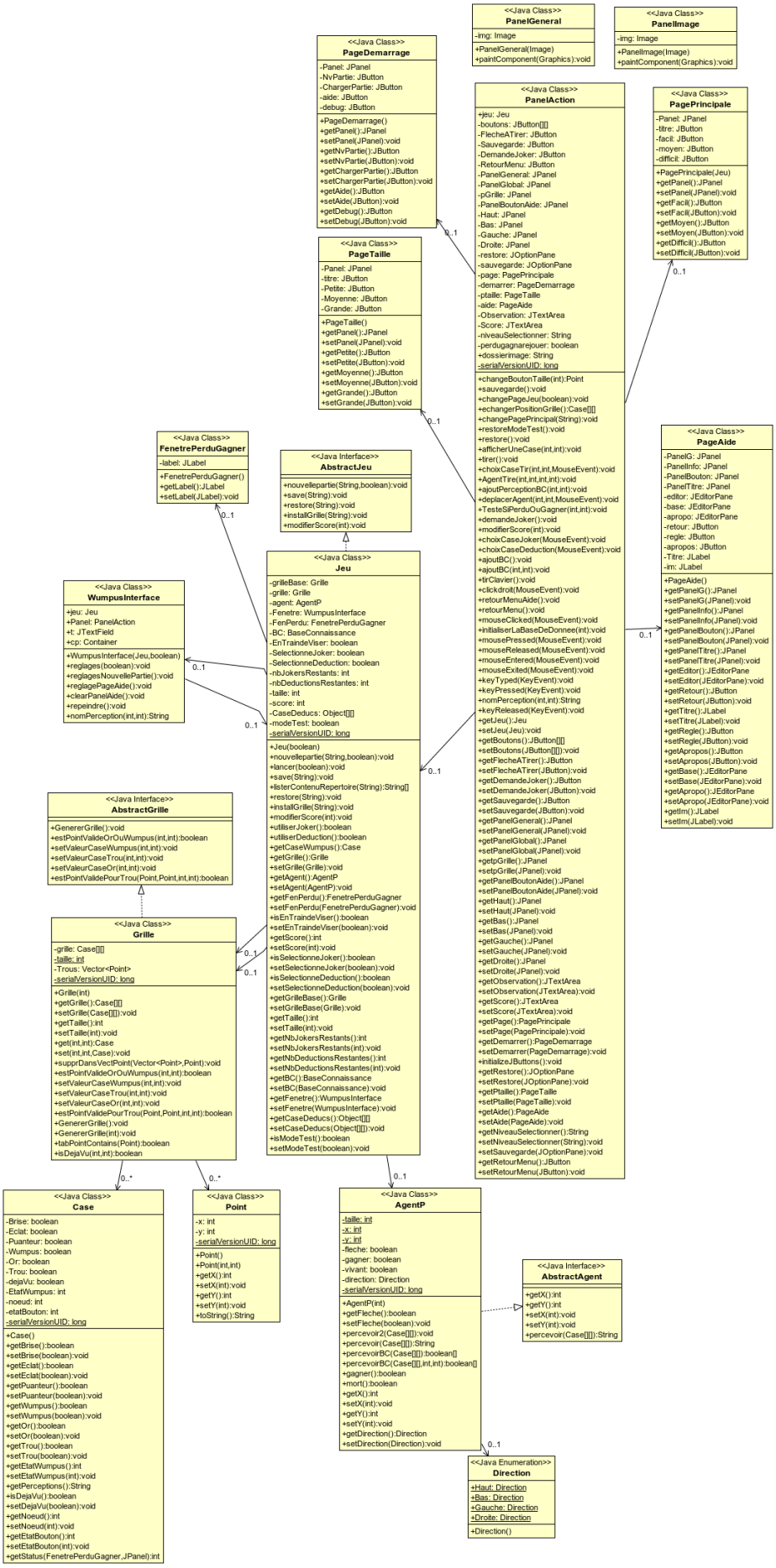
	8*8	10*10	15*15
Facile	7,20	11,24	23,34
Moyenne	5,18	7,14	18,28
Difficile	11,24	15,30	39,65

Bornes choisies pour une taille de grille et sa difficulté.

Ci-dessous le lien vers toutes nos données trouvées :

`http://wumpus.000webhostapp.com/StatGrille.ods`

7.3 UML


<<Java Class>>
PageDemarrage
-Panel: JPanel
-NvPartie: JButton
-ChargerPartie: JButton
-aide: JButton
-debug: JButton
+PageDemarrage()
+getPanel():JPanel
+setPanel(JPanel):void
+getNvPartie():JButton
+setNvPartie(JButton):void
+getChargerPartie():JButton
+setChargerPartie(JButton):void
+getAide():JButton
+setAide(JButton):void
+getDebug():JButton
+setDebug(JButton):void
<<Java Class>>
PanelGeneral
-img: Image
+PanelGeneral(Image)
+paintComponent(Graphics):void
<<Java Class>>
PanelImage
-img: Image
+PanelImage(Image)
+paintComponent(Graphics):void
<<Java Class>>
PageTaille
-Panel: JPanel
-titre: JButton
-Petite: JButton
-Moyenne: JButton
-Grande: JButton
+PageTaille()
+getPanel():JPanel
+setPanel(JPanel):void
+getPetite():JButton
+setPetite(JButton):void
+getMoyenne():JButton
+setMoyenne(JButton):void
+getGrande():JButton
+setGrande(JButton):void
<<Java Class>>
PanelAction
+jeu: Jeu
-boutons: JButton[][]
-FlecheATirer: JButton
-Sauvegarde: JButton
-DemandeJoker: JButton
-RetourMenu: JButton
-PanelGeneral: JPanel
-PanelGlobal: JPanel
-pGrille: JPanel
-PanelBoutonAide: JPanel
-Haut: JPanel
-Bas: JPanel
-Gauche: JPanel
-Droite: JPanel
-restore: JOptionPane
-sauvegarde: JOptionPane
-page: PagePrincipale
-demarrer: PageDemarrage
-ptaille: PageTaille
-aide: PageAide
-Otservation: JTextArea
-Score: JTextArea
-niveauSelectionner: String
-perdugagnarejouer: boolean
+dossierimage: String
-serialVersionUID: long
+changeBoutonTaille(int):Point
+sauvegarde():void
+changePageJeu(boolean):void
+echangerPositionGrille():Case[][]
+changePagePrincipal(String):void
+restoreModeTest():void
+restore():void
+afficherUneCase(int,int):void
+tirer():void
+choixCaseTir(int,int,MouseEvent):void
+AgentTire(int,int,int,int):void
+ajoutPerceptionBC(int,int):void
+deplacerAgent(int,int,MouseEvent):void
+TesteSiPerduOuGagner(int,int):void
+demandeJoker():void
+modifierScore(int):void
+choixCaseJoker(MouseEvent):void
+choixCaseDeduction(MouseEvent):void
+ajoutBC():void
+ajoutBC(int,int):void
+tirClavier():void
+clickdroit(MouseEvent):void
+retourMenuAide():void
+retourMenu():void
+mouseClicked(MouseEvent):void
+initialiserLaBaseDeDonnee(int):void
+mousePressed(MouseEvent):void
+mouseReleased(MouseEvent):void
+mouseEntered(MouseEvent):void
+mouseExited(MouseEvent):void
+keyTyped(KeyEvent):void
+keyPressed(KeyEvent):void
+nomPerception(int,int):String
+keyReleased(KeyEvent):void
+getJeu():Jeu
+setJeu(Jeu):void
+getBoutons():JButton[][]
+setBoutons(JButton[][]):void
+getFlecheATirer():JButton
+setFlecheATirer(JButton):void
+getDemandeJoker():JButton
+setDemandeJoker(JButton):void
+getSauvegarde():JButton
+setSauvegarde(JButton):void
+getPanelGeneral():JPanel
+setPanelGeneral(JPanel):void
+getPanelGlobal():JPanel
+setPanelGlobal(JPanel):void
+getpGrille():JPanel
+setpGrille(JPanel):void
+getPanelBoutonAide():JPanel
+setPanelBoutonAide(JPanel):void
+getHaut():JPanel
+setHaut(JPanel):void
+getBas():JPanel
+setBas(JPanel):void
+getGauche():JPanel
+setGauche(JPanel):void
+getDroite():JPanel
+setDroite(JPanel):void
+getObservation():JTextArea
+setObservation(JTextArea):void
+getScore():JTextArea
+setScore(JTextArea):void
+getPage():PagePrincipale
+setPage(PagePrincipale):void
+getDemarrer():PageDemarrage
+setDemarrer(PageDemarrage):void
+initializeJButtons():void
+getRestore():JOptionPane
+setRestore(JOptionPane):void
+getPtaille():PageTaille
+setPtaille(PageTaille):void
+getAide():PageAide
+setAide(PageAide):void
+getNiveauSelectionner():String
+setNiveauSelectionner(String):void
+setSauvegarde(JOptionPane):void
+getRetourMenu():JButton
+setRetourMenu(JButton):void
<<Java Class>>
PagePrincipale
-Panel: JPanel
-titre: JButton
-facil: JButton
-moyen: JButton
-difficil: JButton
+PagePrincipale(Jeu)
+getPanel():JPanel
+setPanel(JPanel):void
+getFacil():JButton
+setFacil(JButton):void
+getMoyen():JButton
+setMoyen(JButton):void
+getDifficil():JButton
+setDifficil(JButton):void
<<Java Class>>
PageAide
-PanelG: JPanel
-PanelInfo: JPanel
-PanelBouton: JPanel
-PanelTitre: JPanel
-editor: JEditorPane
-base: JEditorPane
-apropo: JEditorPane
-retour: JButton
-regle: JButton
-apropos: JButton
-Titre: JLabel
-im: JLabel
+PageAide()
+getPanelG():JPanel
+setPanelG(JPanel):void
+getPanelInfo():JPanel
+setPanelInfo(JPanel):void
+getPanelBouton():JPanel
+setPanelBouton(JPanel):void
+getPanelTitre():JPanel
+setPanelTitre(JPanel):void
+getEditor():JEditorPane
+setEditor(JEditorPane):void
+getRetour():JButton
+setRetour(JButton):void
+getTitre():JLabel
+setTitre(JLabel):void
+getRegle():JButton
+setRegle(JButton):void
+getApropos():JButton
+setApropos(JButton):void
+getBase():JEditorPane
+setBase(JEditorPane):void
+getApropo():JEditorPane
+setApropo(JEditorPane):void
+getIm():JLabel
+setIm(JLabel):void
<<Java Class>>
FenetrePerduGagner
-label: JLabel
+FenetrePerduGagner()
+getLabel():JLabel
+setLabel(JLabel):void
<<Java Interface>>
AbstractJeu
+nouvellepartie(String,boolean):void
+save(String):void
+restore(String):void
+installGrille(String):void
+modifierScore(int):void
<<Java Class>>
Jeu
-grilleBase: Grille
-grille: Grille
-agent: AgentP
-Fenetre: WumpusInterface
-FenPerdu: FenetrePerduGagner
-BC: BaseConnaissance
-EnTraindeViser: boolean
-SelectionneJoker: boolean
-SelectionneDeduction: boolean
-nbJokersRestants: int
-nbDeductionsRestantes: int
-taille: int
-score: int
-CaseDeducs: Object[][]
-modeTest: boolean
-serialVersionUID: long
+Jeu(boolean)
+nouvellepartie(String,boolean):void
+lancer(boolean):void
+save(String):void
+listerContenuRepertoire(String):String[]
+restore(String):void
+installGrille(String):void
+modifierScore(int):void
+utiliserJoker():boolean
+utiliserDeduction():boolean
+getCaseWumpus():Case
+getGrille():Grille
+setGrille(Grille):void
+getAgent():AgentP
+setAgent(AgentP):void
+getFenPerdu():FenetrePerduGagner
+setFenPerdu(FenetrePerduGagner):void
+isEnTraindeViser():boolean
+setEnTraindeViser(boolean):void
+getScore():int
+setScore(int):void
+isSelectionneJoker():boolean
+setSelectionneJoker(boolean):void
+isSelectionneDeduction():boolean
+setSelectionneDeduction(boolean):void
+getGrilleBase():Grille
+setGrilleBase(Grille):void
+getTaille():int
+setTaille(int):void
+getNbJokersRestants():int
+setNbJokersRestants(int):void
+getNbDeductionsRestantes():int
+setNbDeductionsRestantes(int):void
+getBC():BaseConnaissance
+setBC(BaseConnaissance):void
+getFenetre():WumpusInterface
+setFenetre(WumpusInterface):void
+getCaseDeducs():Object[][]
+setCaseDeducs(Object[][]):void
+isModeTest():boolean
+setModeTest(boolean):void
<<Java Class>>
WumpusInterface
+jeu: Jeu
+Panel: PanelAction
+t: JTextField
+cp: Container
+WumpusInterface(Jeu,boolean)
+reglages(boolean):void
+reglagesNouvellePartie():void
+reglagePageAide():void
+clearPanelAide():void
+repeindre():void
+nomPerception(int,int):String
<<Java Interface>>
AbstractGrille
+GenererGrille():void
+estPointValideOrOuWumpus(int,int):boolean
+setValeurCaseWumpus(int,int):void
+setValeurCaseTrou(int,int):void
+setValeurCaseOr(int,int):void
+estPointValidePourTrou(Point,Point,int,int):boolean
<<Java Class>>
Grille
-grille: Case[][]
-taille: int
-Trous: Vector<Point>
-serialVersionUID: long
+Grille(int)
+getGrille():Case[][]
+setGrille(Case[][]):void
+getTaille():int
+setTaille(int):void
+get(int,int):Case
+set(int,int,Case):void
+supprDansVectPoint(Vector<Point>,Point):void
+estPointValideOrOuWumpus(int,int):boolean
+setValeurCaseWumpus(int,int):void
+setValeurCaseTrou(int,int):void
+setValeurCaseOr(int,int):void
+estPointValidePourTrou(Point,Point,int,int):boolean
+GenererGrille():void
+GenererGrille(int):void
+tabPointContains(Point):boolean
+isDejaVu(int,int):boolean
<<Java Class>>
Case
-Brise: boolean
-Eclat: boolean
-Puanteur: boolean
-Wumpus: boolean
-Or: boolean
-Trou: boolean
-dejaVu: boolean
-EtatWumpus: int
-noeud: int
-etatBouton: int
-serialVersionUID: long
+Case()
+getBrise():boolean
+setBrise(boolean):void
+getEclat():boolean
+setEclat(boolean):void
+getPuanteur():boolean
+setPuanteur(boolean):void
+getWumpus():boolean
+setWumpus(boolean):void
+getOr():boolean
+setOr(boolean):void
+getTrou():boolean
+setTrou(boolean):void
+getEtatWumpus():int
+setEtatWumpus(int):void
+getPerceptions():String
+isDejaVu():boolean
+setDejaVu(boolean):void
+getNoeud():int
+setNoeud(int):void
+getEtatBouton():int
+setEtatBouton(int):void
+getStatus(FenetrePerduGagner,JPanel):int
<<Java Class>>
Point
-x: int
-y: int
-serialVersionUID: long
+Point()
+Point(int,int)
+getX():int
+setX(int):void
+getY():int
+setY(int):void
+toString():String
<<Java Class>>
AgentP
-taille: int
-x: int
-y: int
-fleche: boolean
-gagner: boolean
-vivant: boolean
-direction: Direction
-serialVersionUID: long
+AgentP(int)
+getFleche():boolean
+setFleche(boolean):void
+percevoir2(Case[][]):void
+percevoir(Case[][]):String
+percevoirBC(Case[][]):boolean[]
+percevoirBC(Case[][],int,int):boolean[]
+gagner():boolean
+mort():boolean
+getX():int
+setX(int):void
+getY():int
+setY(int):void
+getDirection():Direction
+setDirection(Direction):void
<<Java Interface>>
AbstractAgent
+getX():int
+getY():int
+setX(int):void
+setY(int):void
+percevoir(Case[][]):String
<<Java Enumeration>>
Direction
+Haut: Direction
+Bas: Direction
+Gauche: Direction
+Droite: Direction
+Direction()
0..1
0..*