Rapport de projet : Lightup !

ARVOUET Mickaël
DEROBERT-MAZURE Adrien

19 Octobre 2022

1 Introduction

Dans ce rapport nous allons principalement donner nos réponses et réflexions aux questions théoriques du projet telles que les algorithmes en pseudo-code, ou encore les formules de réduction.
Nous allons également donner des précisions sur notre code et pseudo-code, ainsi que préciser les questions ou améliorations d'implémentation que nous n'avons pas traitées.

2 Vérificateur

2.1 Algorithme en pseudo-code

```
all_cells_lighted(M1, M2):
    pour chaque case de M2:
        bool eclaire = false
        if case est une lampe:
            eclaire = true
        else:
            pour chaque case à gauche:
                if case est un mur:
                    continue
                if case est une lampe:
                    eclaire = true
            pour chaque case à droite:
                if case est un mur:
                    continue
                if case est une lampe:
                    eclaire = true
            pour chaque case au-dessus:
                if case est un mur:
                    continue
                if case est une lampe:
                    eclaire = true
            pour chaque case en-dessous:
                if case est un mur:
                    continue
                if case est une lampe:
                    eclaire = true
        if !eclaire -> return false
    return True
```

Dans l'implémentation du bruteforce.c, cette fonction sera légèrement différente et fera appel à plusieurs sous-fonctions qui vérifient chacune dans une direction pour une case donnée, si cette case est éclairée. Si c'est le cas on met la variable eclaire à True, si elle n'a pas été mise à true apres avoir visité toute la ligne et toute la colonne, on renvoie False

```
no_lights_aligned(M1, M2):
    pour chaque case M2(i, j):
        si la case est une lampe (M2(i, j) == 1):
            Pour chaque case a gauche:
                if case == WALL:
                    break
                si la case de M2 est une lampe:
                    return False
            Pour chaque case a droite:
                if case == WALL:
                    break
                si la case de M2 est une lampe:
                    return False
            Pour chaque case au dessus:
                if case == WALL:
                    break
                si la case de M2 est une lampe:
                    return False
            Pour chaque case en dessous:
                if case == WALL:
                    break
                si la case de M2 est une lampe:
                    return False
    return True
```

Cette fonction utilisera également les fonctions évoquées précédemment afin de savoir si une case avec une lampe est éclairée, si c'est le cas on retourne False, sinon on parcourt la grille jusqu'à la fin et retourne True

```
wall_constraints_check(M1, M2):
    pour chaque case M(i, j):
        si la case est un mur:
            cpt = 0
            goal = nombre de lampes attendues à côté du mur
            pour chaque case adjacente :
                si case est une lampe:
                    cpt ++
            si cpt != goal     -> False
    return True
```

Pour toutes les cases, si c'est un mur, on compte le nombre de lampes adjacentes, si ce nombre est différent de celui attendu, on renvoie false, sinon on continue jusqu'à atteindre la fin de la grille.

```
verifier(M1, M2):
    return r1 && r2 && r3
```

Le vérificateur teste simplement les trois fonctions définies ci-dessus sur la grille qui fait office de certificat.

2.2 Compexité de l'algorithme

— all cells lighted : $O(n * m * (n + m))$
car pour chaque case, vérifie au plus toute la colonne et toute la ligne

— no lights aligned : $O(n * m * (n + m))$
idem que pour r1

— wall constraints check : $O(n * m * 4) = O(n * m)$
pour chaque case, vérifie au plus les 4 cases adjacentes

— vérificateur : $O(n * m * (n + m)) = O(nm + nm)$
En prenant n (lignes) $> m$ (colonnes) $=> 2nn > nm + nm$
Donc complexité finale : $O(max(m, n))$

2.3 Complexité du problème Lightup

On vient de montrer que l'algorithme permettant de vérifier si un ensemble de lampes est une solution à une instance de jeu a une complexité polynomiale. Si cet algorithme renvoie false sur toutes les matrices M2, il n'y a alors pas d'ensemble de lampes qui vérifie les propriétés. De plus, les matrices M2 possibles sont de taille polynomiale en la taille de l'instance (elles ont en fait la même taille)

=> On peut donc en déduire que le problème Lightup est dans NP.

2.4 Bruteforce en pseudo-code

```
to_binary(nb, M2):
    pour i de 0 à nb_rows:
        pour j de 0 à nb_cols:
            if (case == blank):
                M2[i][j] = nb % 2
            n = n / 2
            if (nb == 0):
                return
        }
    }
}
```

La fonction tobinary est utilisée pour remplir M2 avec la valeur en binaire d'un compteur qu'on incrémente à chaque fois qu'on teste un certificat

```
brute_force(M1) {
    creer grille M2 identique à M1
    creer une autre grille M3 identique         // grille "tampon" utilisée pour ne pas toucher aux
    if (verificateur(M1, M2)):                  // lampes déjà posées
        return True
    cpt = 1;
    while (cpt < 2^nombre de cases - 1):
        if (verificateur(M1, toBinary(cpt, M2)):
            return True
        cpt ++
    return False
}
```

Ce bruteforce appelle verifier sur tous les certificats possibles, créés avec tobinary, jusqu'à en trouver un qui satisfait toutes les règles

2.5 Complexité de l'algorithme

Comme vu en partie 2.3, le nombre de solutions testées ici dans le bruteforce est $S = 2^{i*j} - 1$, soit une complexité exponentielle, multipliée pour chaque solution testée par la complexité de la fonction verifier

2.6 Idées pour diminuer la complexité

— ne pas tester les cas où une lampe est adjacente à une autre
— ne pas tester quand une lampe est placée sur un mur
— ...

On n'a pas fait le pseudo-code du bruteforce optimisé, on a donc implémenté la version "naïve" de ce dernier.

2.7 Implémentation du bruteforce

On a donc implémenté une version du bruteforce qui n'est pas vraiment optimisée, et de plus notre fonction n'est pas récursive. Ainsi elle se limite à des grilles de taille 27 cases (fonctionne sur une grille 3x9 mais pas sur une grille 4x7). Cependant nous avons remarqué que suivant l'ordre des appels aux trois règles dans verifier, il est possible d'économiser un peu de temps de calcul puisque certaines règles seront moins souvent vérifiées, ou ont une complexité inférieure que d'autres (donc pas besoin de tester les autres). Exemple de performances du brute_force :

— 4x4.l, Brute Force successful in 0.005979 seconds (résultat correct)
— 3x3_nosol.l, Brute Force found no solution in 0.000442 seconds (résultat correct)
— 5x5.l, Brute Force successful in 3.76079 seconds (résultat correct)
— à partir de 3x10.l, Timeout in Brute force in 30 seconds !

3 Réduction à SAT

3.1 exact$_k$

$$exact_k(var, n) = \begin{cases} \bigwedge_{i=0}^{n-1} \neg var(i) & \text{si } k = 0 \\ \bigwedge_{i=0}^{n-1} var(i) & \text{si } k = n \\ \bot & \text{si } k > n \\ (var(i) \wedge exact_{k-1}(var \setminus \{var(i)\}, n-1)) \vee (\neg x_i \wedge exact_k(var \setminus \{var(i)\}, n-1)) & \text{sinon} \end{cases}$$

3.2 Fonction red

— no lights aligned :

$$\varphi_1 : \bigwedge_{\substack{(i,j),(i',j') \in G | \\ (i,j) et (i',j') "se\ voient"}} \neg x_{i,j} \vee \neg x_{i',j'}$$

— all cells lighted :
  — On crée tout d'abord une fonction $atLeast_k(var, n)$ dans le même esprit que $exact_k$ mais qui renvoie true si au moins k éléments de var sont à $\top$:

$$atLeast_k(var, n) = \begin{cases} \bot & \text{si } k = \\ \bigwedge_{i=0}^{n-1} var(i) & \text{si } k = \\ \bot & \text{si } k > \\ (var(i) \wedge atLeast_{k-1}(var \setminus \{var(i)\}, n-1)) \vee (\neg x_i \wedge atLeast_k(var \setminus \{var(i)\}, n-1)) & \text{sinon} \end{cases}$$

— On se sert ensuite de cette fonction pour vérifier que pour chaque case *BLANK* du jeu soit éclairée par au moins une lampe :

$$\varphi_2 : \bigwedge_{(i,j)|M(i,j)=BLANK} atLeast_1(visibles(i,j), nb_visible(i,j))$$

Ici, on a :
— visible retourne le tableau rempli des variables $x_{i',j'}$ ou i' et j' sont les coordonnées de toutes les cases tel que la case (i,j) et la case (i',j') se voient. Cela est valable pour $(i,j) = (i',j')$.
— $\mathrm{nb}_visible retourn la taille de ce tableau (soit le nombre de cases susceptibles d'éclairer(i,j))$

— walls constraints check

$$\varphi_3 : \bigwedge_{(i,j)|M(i,j)=WALL} exact_k(voisins, nbvoisins)$$

Ici, on a :
— k qui correspond au nombre de lampes attendues par le mur sur les cases adjacentes à ce dernier
— voisins le tableau rempli des variables $x_{i',j'}$ ou i' et j' les coordonnées des cases *BLANK* voisines au mur.
— nbvoisins la taille de ce tableau (soit le nombre total de cases adjacentes au mur)

On a donc maintenant des formules pour les trois règles du jeu, il ne nous reste plus qu'à les relier par des $\wedge$.
On obtient alors :

$$\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$$

3.3 Réduction de Lightup vers SAT ?

=>
Supposons qu'il existe une instance positive M1 du problème Lightup,
On a M2 une matrice de lampes telle que M2 est solution de M1

On construit val qui a $x_{i,j}$ associe vrai si et seulement si M2(i, j) est une lampe,

φ_1 :
Dans le certificat, aucune lampe n'en éclaire une autre, donc pour toutes paires de cases (i, j), (i', j') telles que (i, j) et (i', j') se voient (c'est-à-dire qu'elles sont sur la même ligne ou la même colonne, et qu'il n'y a pas de mur entre elles), au moins une des deux cases n'est pas une lampe.
Alors,

$$\neg x_{i,j} \vee \neg x_{i',j'}$$

est vrai,
donc φ_1 est vrai.

φ_2 :
Pour toutes les cases (i, j) telles que M1(i, j) est une case vide, cette case est éclairée
Donc pour toutes ces cases (i, j), il existe au moins une une lampe de coordonnées (i', j') telle que (i, j) et (i', j') se voient.
Alors pour toutes ces cases blank,

$$atLeast_1(voisins, nbpossibles)$$

est vrai,
donc φ_2 est vrai.

φ_3 :
Pour toutes les cases (i, j) telles que M1(i, j) est un mur (avec k le nombre de lampes attendues par ce mur sur ses cases adjacentes), toutes ces cases ont k lampes sur les cases adjacentes à ces dernières. Donc pour tous ces murs,

$$exact_k(var, nbvoisins)$$

est vrai,
donc φ_3 est vrai.

=> Donc $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ est vrai

---

<=

Supposons qu'il existe $val \models \varphi_1 \wedge \varphi_2 \wedge \varphi_3$

On cherche à construire M2,

φ_1 :
Cette règle est validée par le fait que pour toutes paires de cases qui "se voient", au moins l'une d'entre elle n'est pas une lampe grâce à :

$$\neg x_{i,j} \vee \neg x_{i',j'}$$

Ainsi la solution M2 générée par val ne comporte aucune ligne ou colonne avec une lampe qui en éclaire une autre une fois dans M1.

φ_2 :
Cette règle est validée par le fait que pour chaque case blank (i, j), il y ait au moins 1 $x_{i',j'}$ à vrai parmi les cases qui "voient" (i, j), grâce à :

$$atLeast_1(voisins, nbpossibles)$$

Ainsi M1 ne comporte aucune case blank non éclairée par une lampe de M1.

φ_3 :
Cette règle est validée par le fait que pour chaque mur (i, j), avec k la valeur du mur, il y a exactement k lampes adjacentes au mur, grâce à :

$$exact_k(voisins, nbvoisins)$$

Ainsi la solution que l'on construit respecte le fait qu'aucun mur de M1 ne verra sa contrainte de lampes adjacentes non vérifiée vérifiée.

=> Les 3 règles de Lightup sont vérifiées, donc M2 est bien un certificat prouvant que M1 est une instance positive.

3.4 Taille de red(G)

red(G) est de taille polynomiale en la taille de G.

On prend C = nbcols(G) et R = nbrows(G)

$\varphi_1 : O(R * C * (R + C))$
$atLeast() : O(R + C)$
$=> \varphi_2 : O(R * C * (R + C))$
$exact() : O(4)$
$=> \varphi_2 : O(R * C * 4) = O(R * C)$

$=> \varphi : O(R * C * (R + C))$

3.5 Complexité de Lightup

Nous avons réussi à réduire le problème Lightup à SAT (qui est dans NP) par une formule de taille polynomiale en la taille de l'instance.
=> On peut donc en conclure que le problème Lightup est dans NP.

3.6 NP-complet ?

Nous n'avons pas, au cours de ce projet, démontré que le problème Lightup est NP-complet. Pour cela, sachant maintenant qu'il est NP, il aurait également fallut démontrer qu'il est NP-difficile.

3.7 Implémentation reduction.c

Notre implémentation de reduction.c est fonctionnelle est obtient un résulat correct pour toutes les instances de grilles.
Nous avons implémenté une fonction $visible(ctx, g, i, j, *squares)$ qui rempli le tableau squares des valeurs dans le certificat des cases que voient (i, j) dans g.
Et en plus de la fonction $exact_k(var, n)$ nous avons implémenté, comme indiqué en 3.2, une fonction $atLeast_k(var, n)$ qui va vérifier le tableau rempli grâce à $visible()$, si au moins un des cases qui voient (i, j) est une lampe. Fonction que nous utilisons pour vérifier la règle "all cells lighted". Exemple de performances de la réduction :

- 3x3.l, Formula solved in 0.011068 seconds (résultat correct)
- 8x8_v3_nosol.l Formula solved in 0.011893 seconds. No solution found (résultat correct)
- 100x100.l, Formula solved in 0.162523 seconds (résultat correct)

4 Bonus : Grilles à une ligne

4.1 Ajout possible ou contradiction

- Un mur est marqué par 3 ou 4 : contradiction (dans un jeu sur une seul ligne, un mur a aux plus 2 cases adjacentes)

- Un mur marqué par 2 : possible d'ajouter des $\top$ ou des $\bot$ (en l'occurrence il faudra des $\top$ sur les 2 cases adjacentes à ce mur, s'il n'est pas au début ou à la fin de la grille)

- Un mur marqué par 1 avec un voisin marqué par : possible d'ajouter des $\top$ ou des $\bot$ (encore une fois, seulement si ce mur n'est pas à un bout de la grille, il faudra rajouter un $\top$ sur le voisin non marqué de ce mur)

- Deux murs séparés par une ou plusieurs cases vides, toutes marquées $\bot$: contradiction (les cases entre ces deux murs ne pourront jamais être éclairées)

- Deux cases vides sans mur entre elles dont une est marquée $\top$: possible d'ajouter des $\top$ ou des $\bot$ (il faudra mettre à $\bot$ les cases non marquées)

4.2 remplissage valide

4.3 Algorithme grilles à une ligne