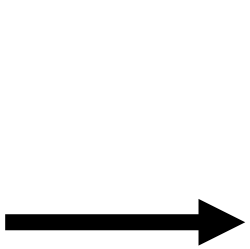
Tristan Lemoine

Candidat 41679

Résolution de nonograms à l’aide de différentes méthodes de programmation

Présentation du problème

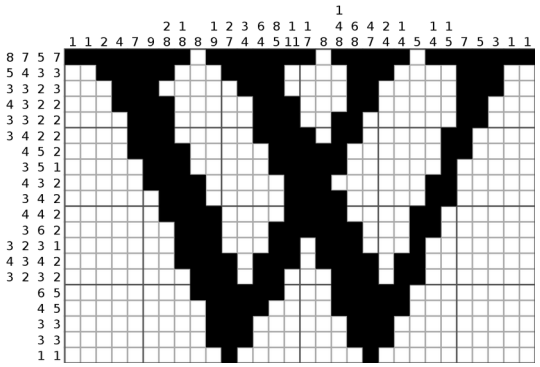
	1	5	2	5	2 1	2
2 1						
1 3						
1 2						
3						
4						
1						

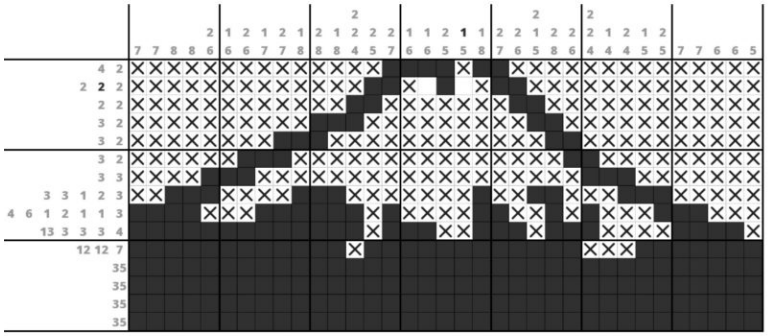


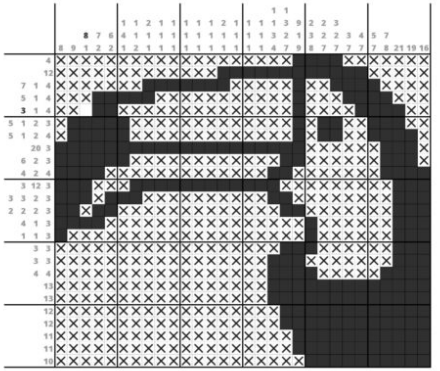
	1	5	2	5	2 1	2
2 1	■	■				■
1 3		■		■	■	■
1 2		■		■	■	
3		■	■	■		
4		■	■	■	■	
1				■		

Non résolu

résolu







Images tirées de “Constructing Simple Nonograms of Varying Difficulty“ de Kees Joost Batenburg ,la page wikipedia des nonograms et la revue “Picross For A Cause” par buried-treasaure.org

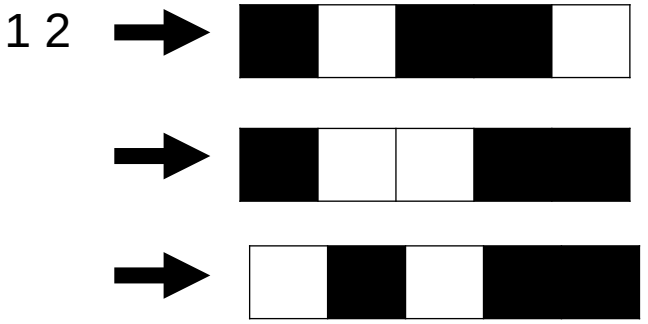
Règles du jeu

Chaque ligne contient l'ordre et la taille des blocs qui occupent la ligne
-Un bloc est un nombre succesif de blocs noirs

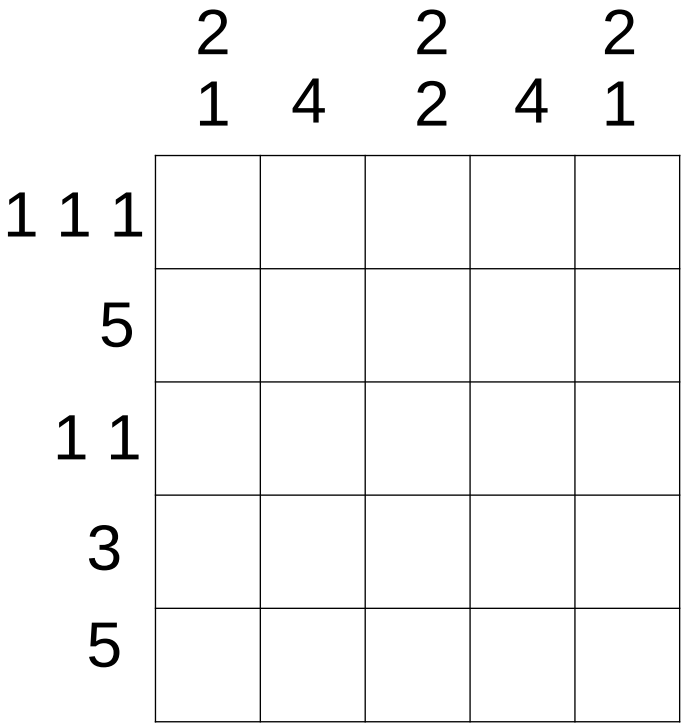
L'inconnue dans ce jeu est donc la position de ces blocs sur la ligne
-Deux blocs sont séparés par au moins une case blanche

Si toutes les lignes et colonnes sont satisfaites alors la grille est résolue

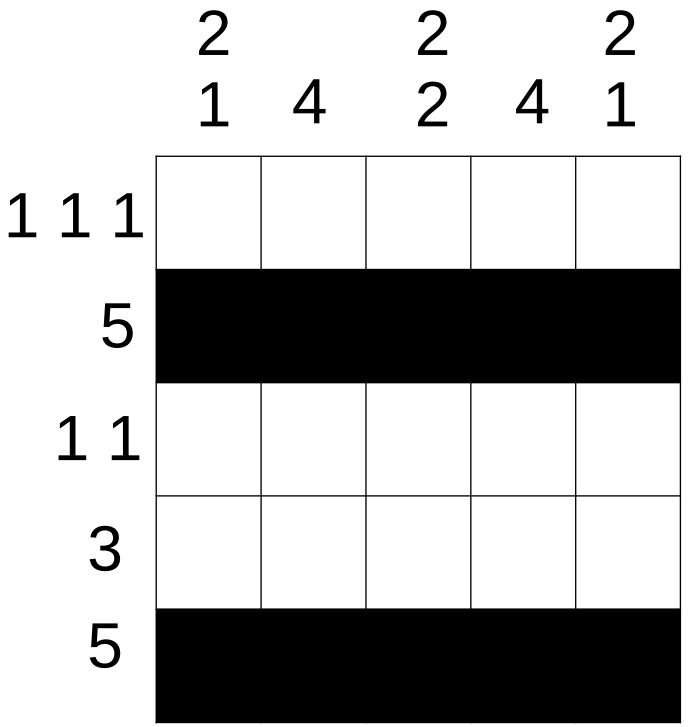
Note:Ce jeu est un problème NP-complet



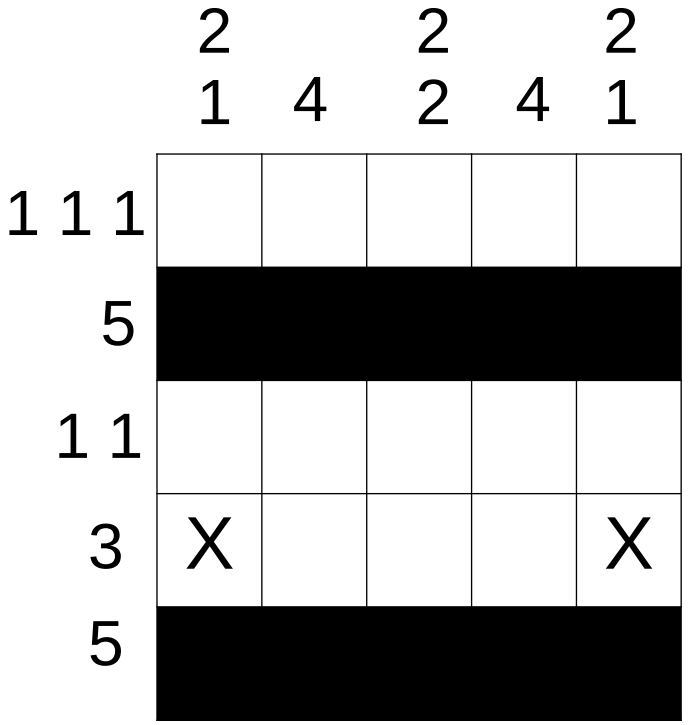
Résolution d’un nonogram



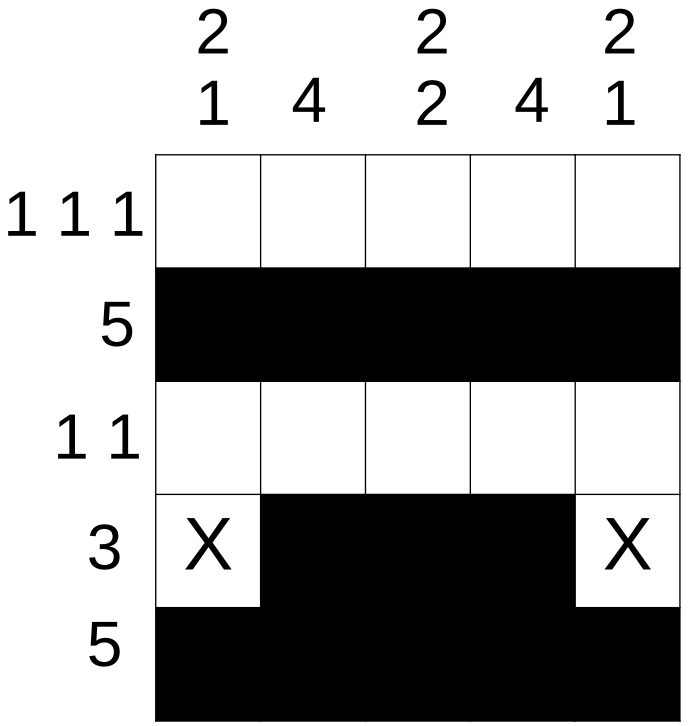
Résolution d’un nonogram



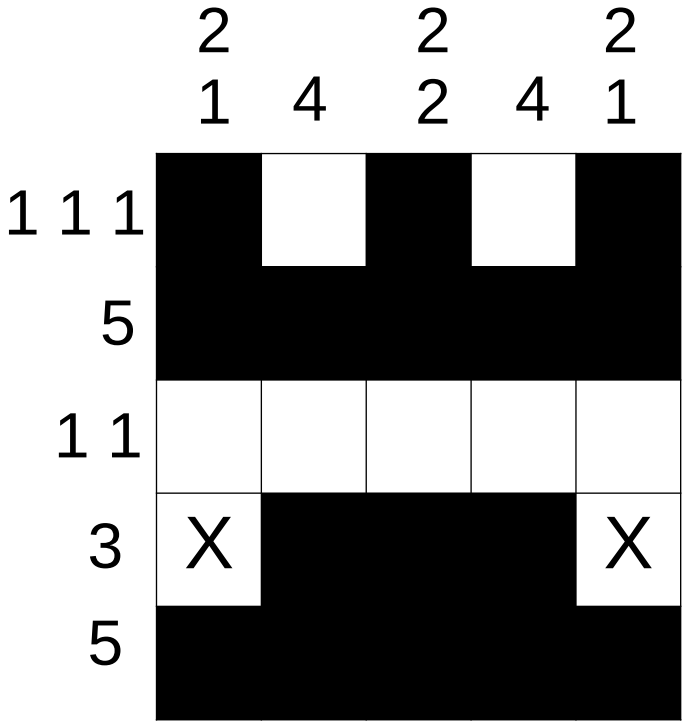
Résolution d’un nonogram



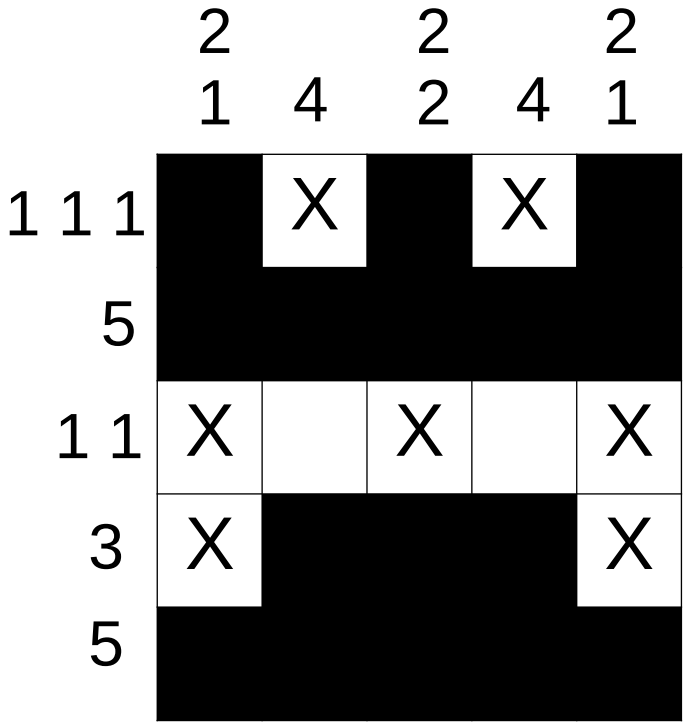
Résolution d’un nonogram



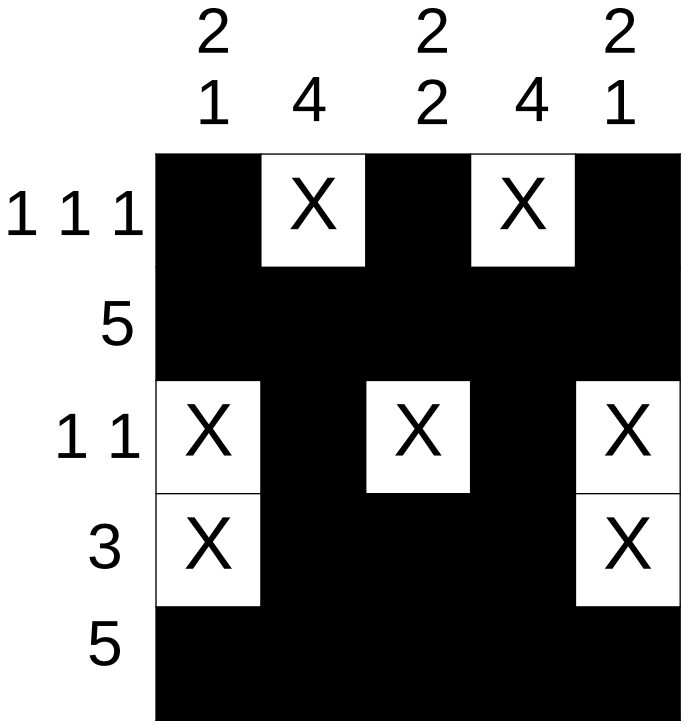
Résolution d’un nonogram



Résolution d’un nonogram



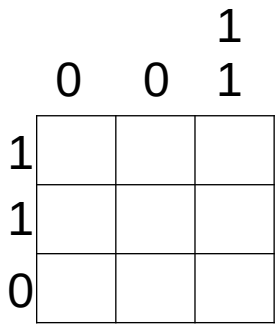
Résolution d’un nonogram



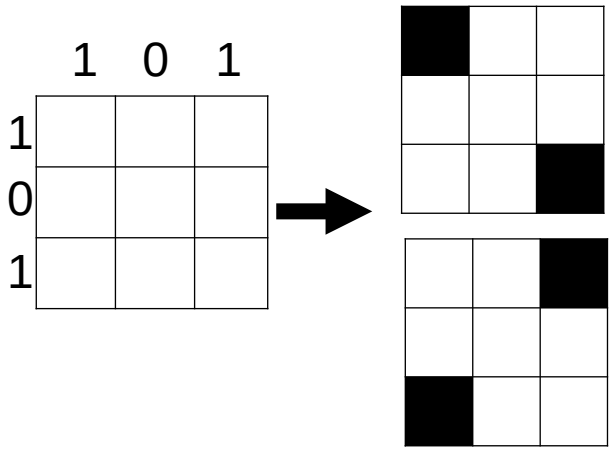
Cadre du problème

On cherche à résoudre le plus rapidement possible une grille initialment vide.

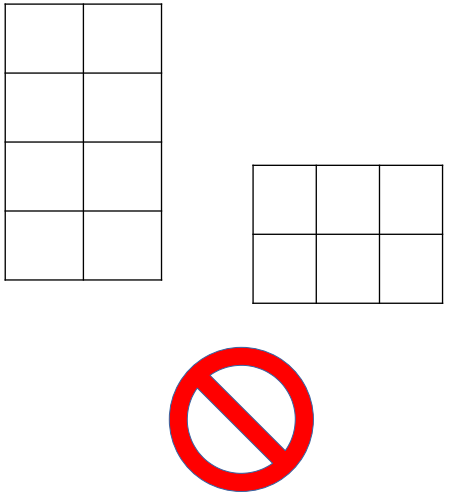
Contraintes:



Au moins une solution possible

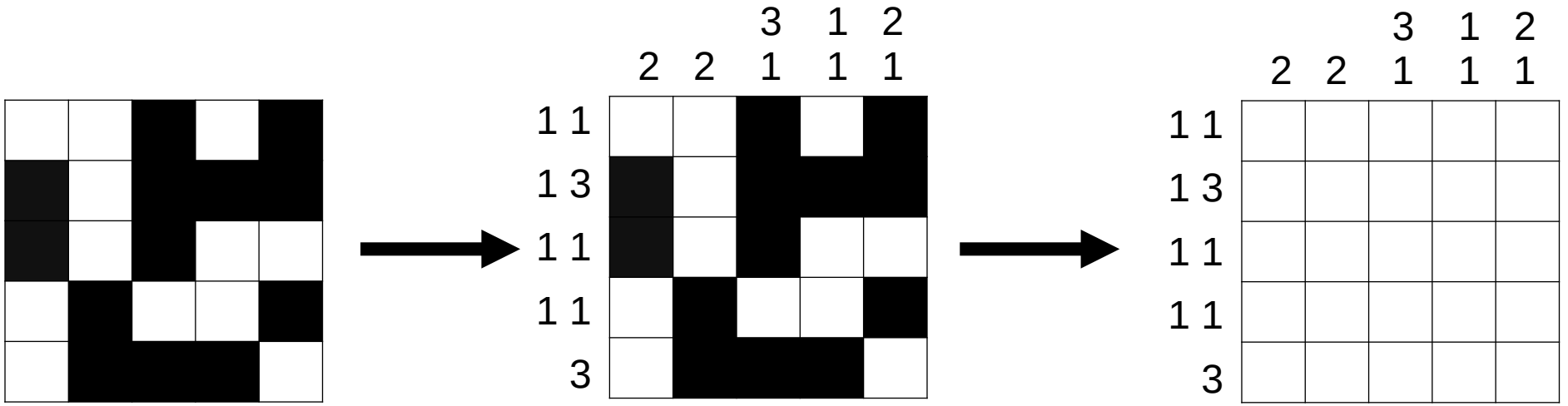


Seule une solution est nécessaire pour considerer la grille comme résolue



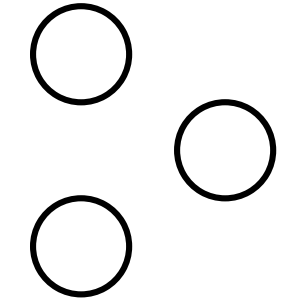
La grille est carrée

Generation de la grille

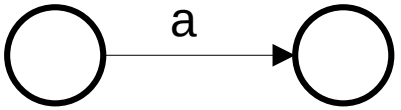


Les automates

Les automates sont construits avec:



Des états



Des relations

{0;1}

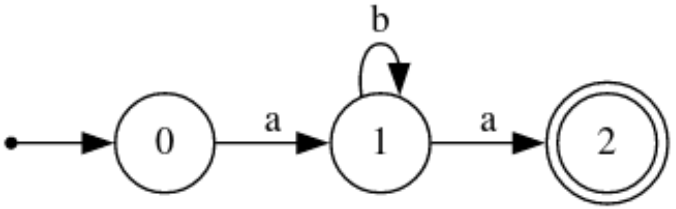
{a;b;c}

{■;□}

...

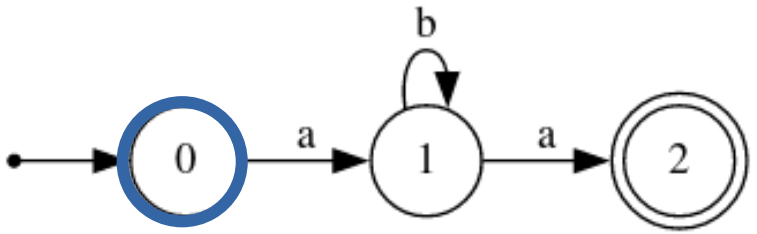
Un alphabet

Exemple:



Les automates

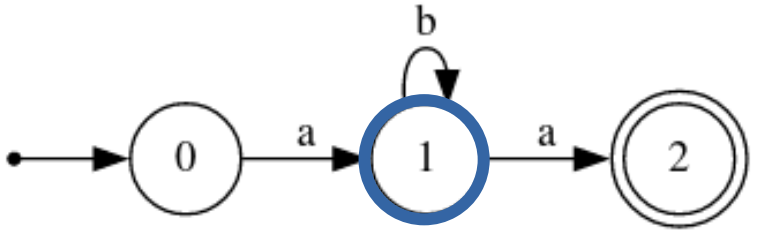
a b b a



Mots qui sont reconnus par cet automates	Mots qui ne sont pas reconnus par cet automate
-aba -aa -abba -abb...bba	-a -abaa -baaa ...

Les automates

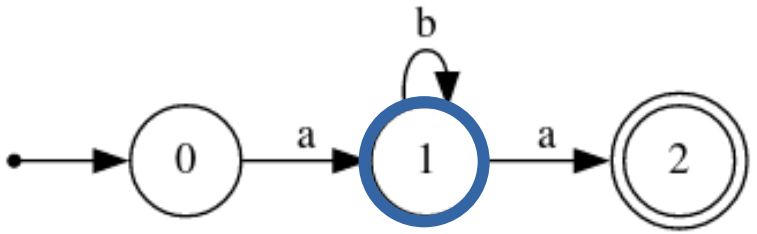
a b b a



Mots qui sont reconnus par cet automates	Mots qui ne sont pas reconnus par cet automate
-aba -aa -abba -abb...bba	-a -abaa -baaa ...

Les automates

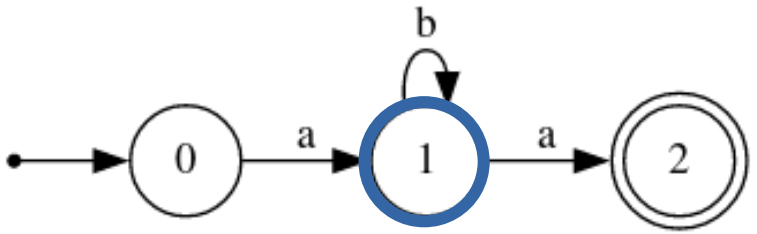
a b b a



Mots qui sont reconnus par cet automates	Mots qui ne sont pas reconnus par cet automate
-aba -aa -abba -abb...bba	-a -abaa -baaa ...

Les automates

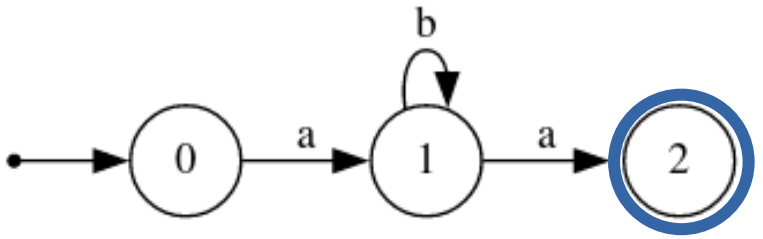
a b b a



Mots qui sont reconnus par cet automates	Mots qui ne sont pas reconnus par cet automate
-aba -aa -abba -abb...bba	-a -abaa -baaa ...

Les automates

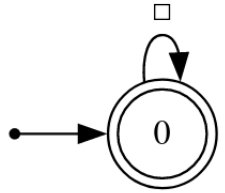
a b b a



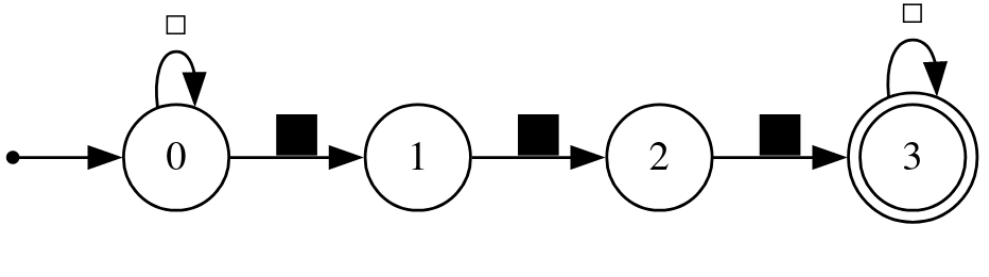
Mots qui sont reconnus par cet automates	Mots qui ne sont pas reconnus par cet automate
-aba -aa -abba -abb...bba	-a -abaa -baaa ...

Example d’automates

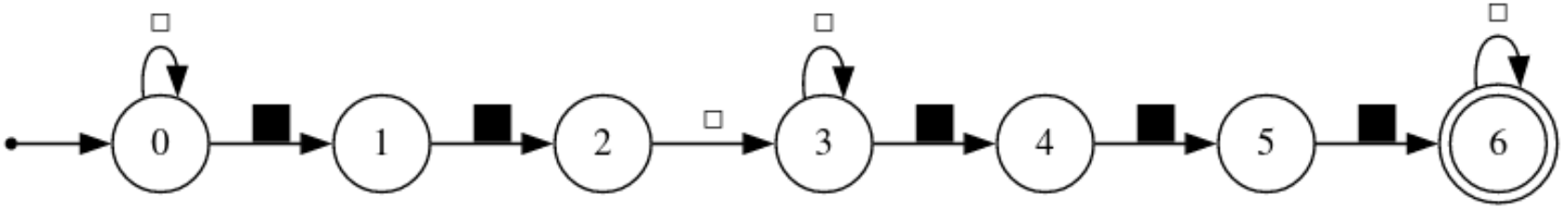
0



3

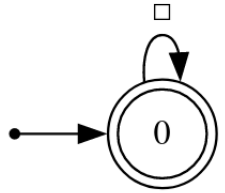


2 3

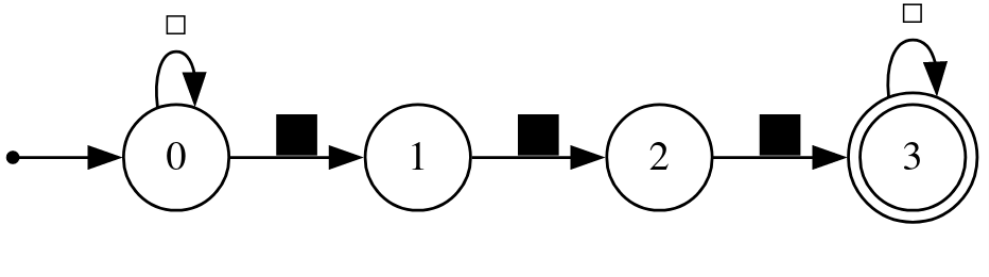


Example d’automates

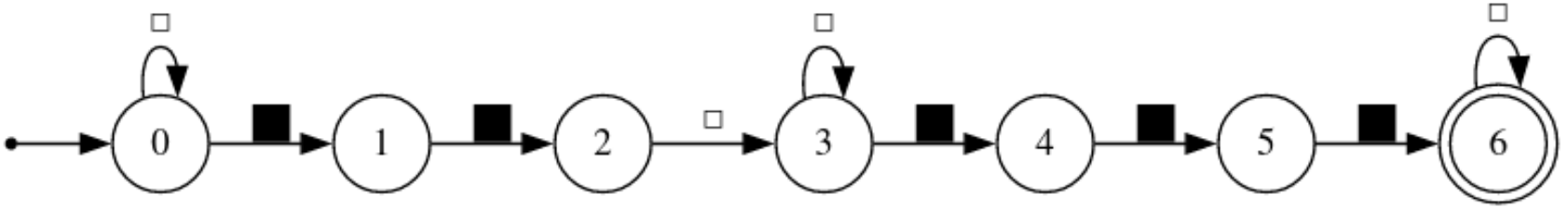
0



3



2 3



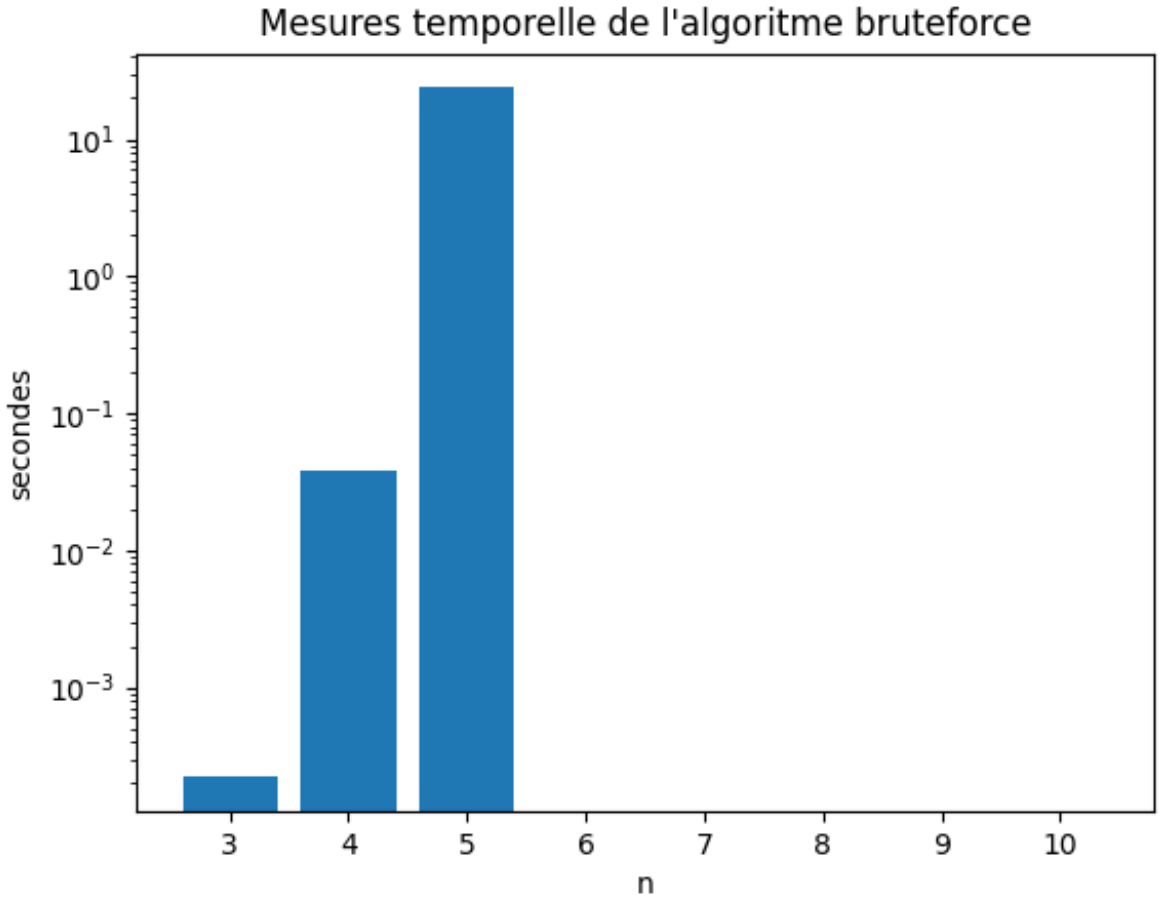
Verifier une ligne est linéaire en la taille de la ligne

Premier algorithme

Algorithme de force brute:Tenter toutes les grilles possibles jusqu’a trouver la bonne

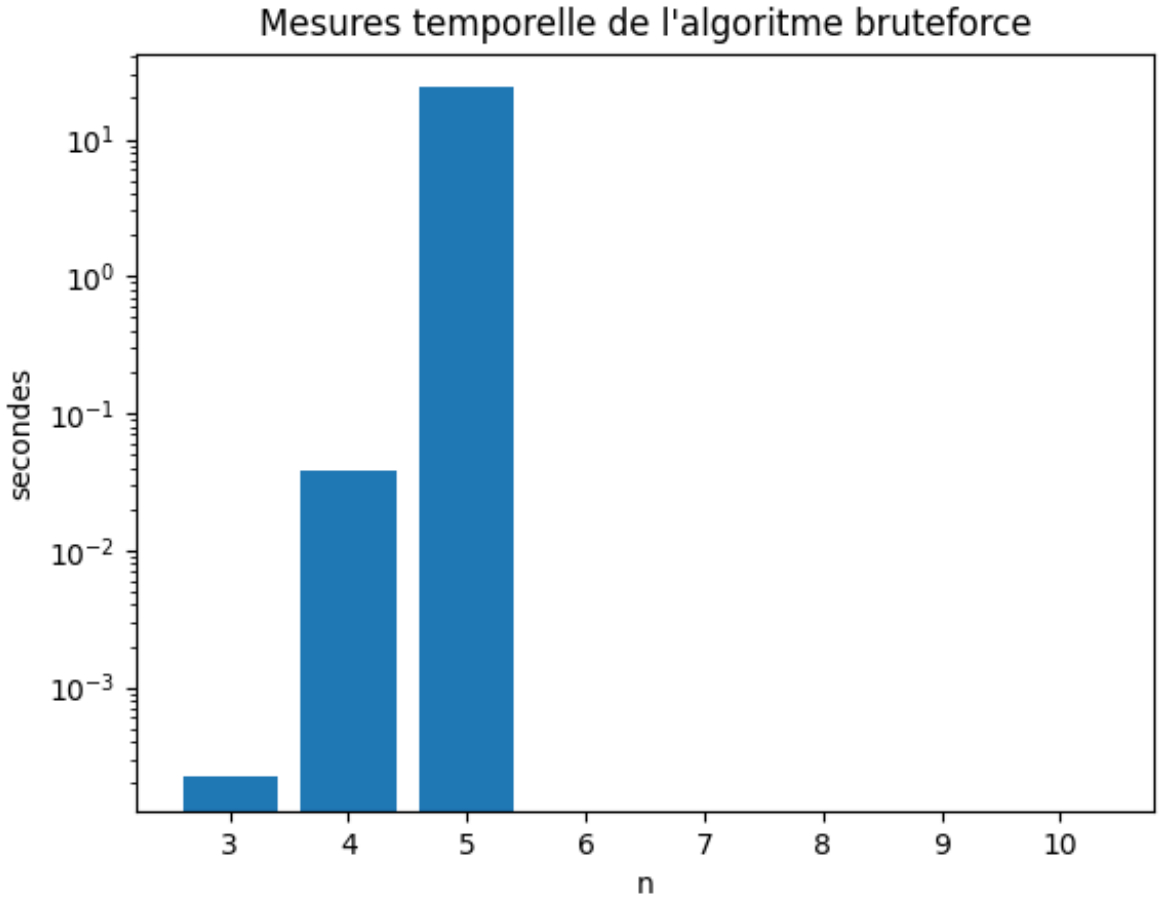
Premier algorithme

Algorithme de force brute:Tenter toutes les grilles possibles jusqu'a trouver la bonne



Premier algorithme

Algorithme de force brute:Tenter toutes les grilles possibles jusqu'a trouver la bonne

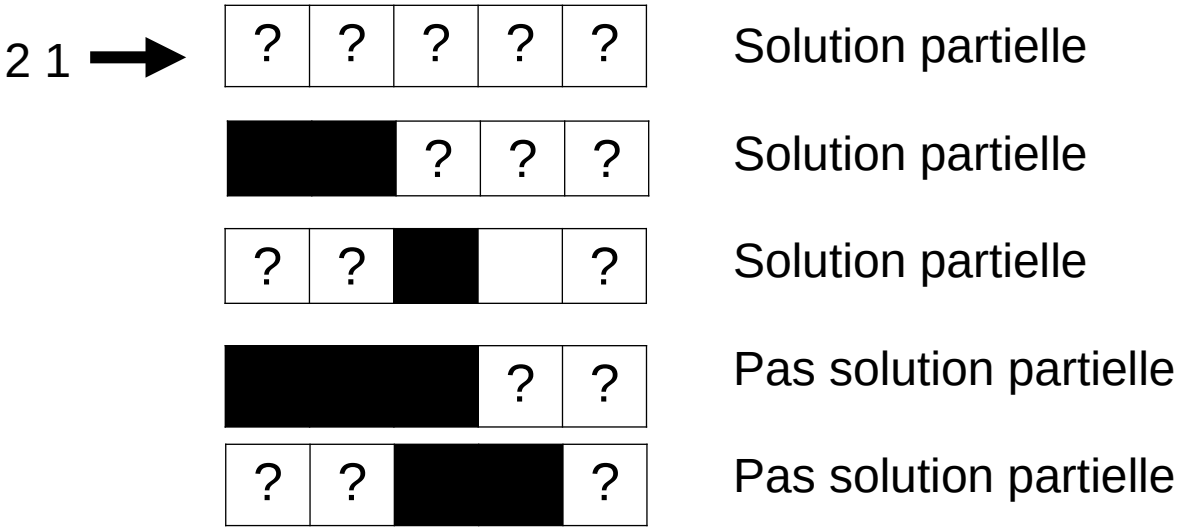


La complexité de cet algorithme est en $O(2^{n^2}n^2)$

Les solutions partielles

On introduit la case “inconnue” ?

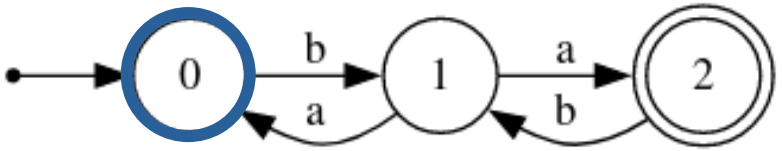
Une ligne est une solution partielle si il existe une disposition des cases inconnues tel que la ligne puisse etre validée



Les automates non déterministes

Un automate qui pour au moins un état possede plusieurs relations portant le même symbole est un automate non-deterministe

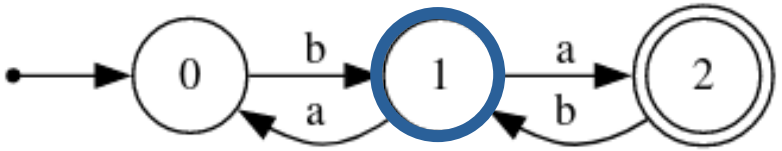
b a b a



Les automates non déterministes

Un automate qui pour au moins un état possede plusieurs relations portant le même symbole est un automate non-deterministe

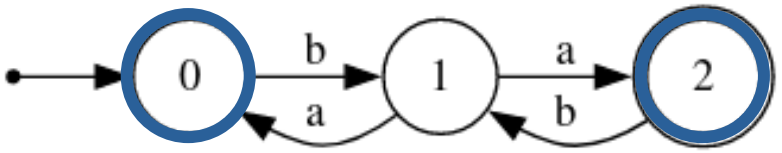
b a b a



Les automates non déterministes

Un automate qui pour au moins un état possede plusieurs relations portant le même symbole est un automate non-deterministe

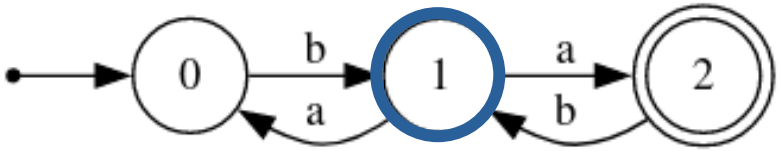
b a b a



Les automates non déterministes

Un automate qui pour au moins un état possede plusieurs relations portant le même symbole est un automate non-deterministe

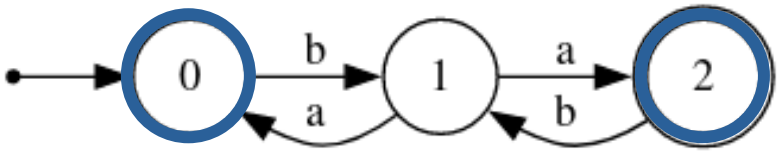
b a b a



Les automates non déterministes

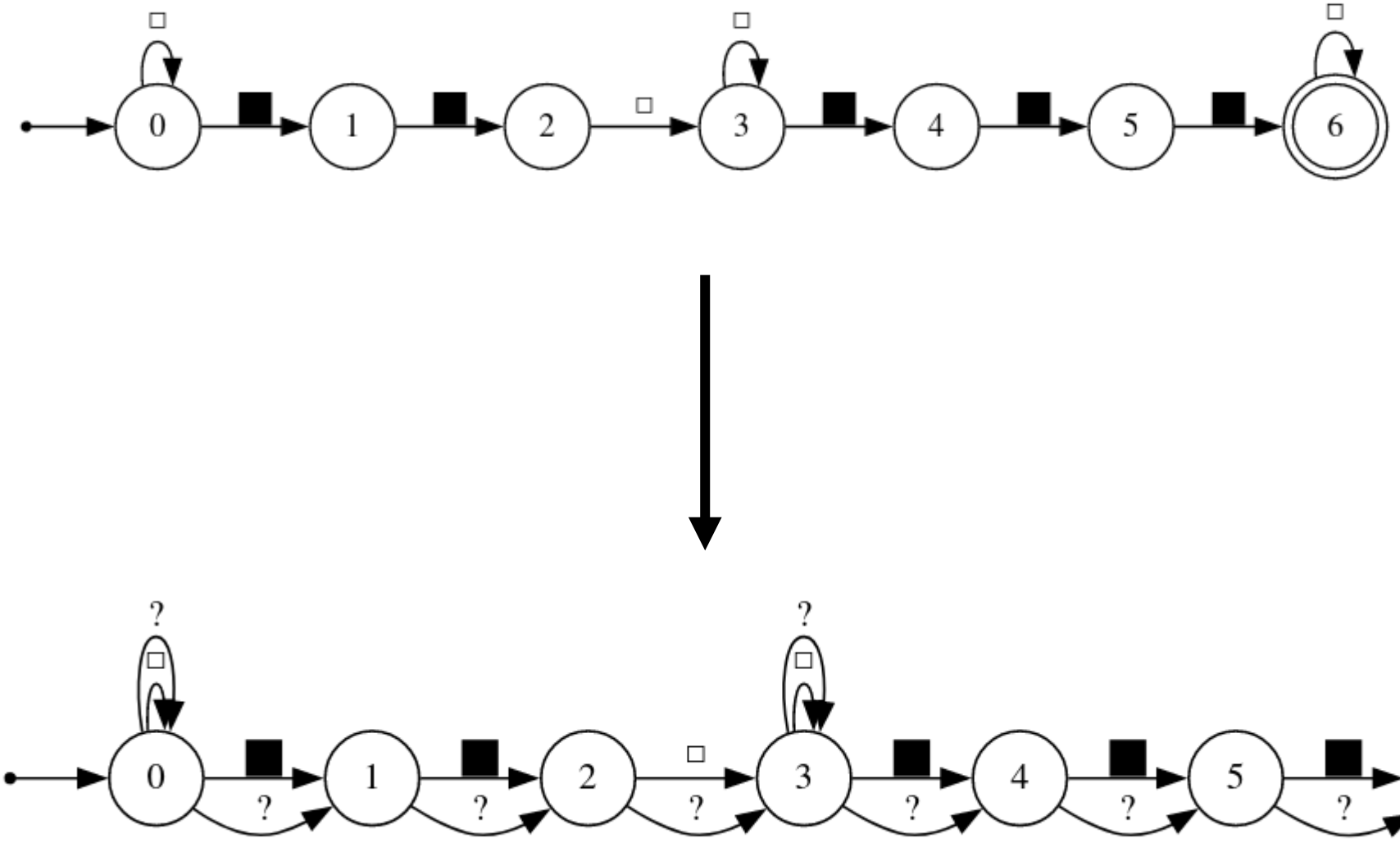
Un automate qui pour au moins un état possede plusieurs relations portant le même symbole est un automate non-deterministe

b a b a



La complexité de faire entrer un mot dans un automate non-deterministe dépend de la taille du mot et de la taille de l'automate

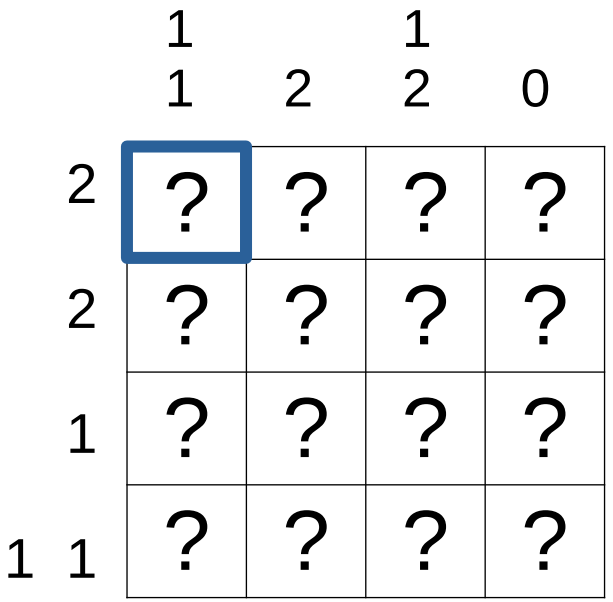
Création de l'automate partiel pour les lignes



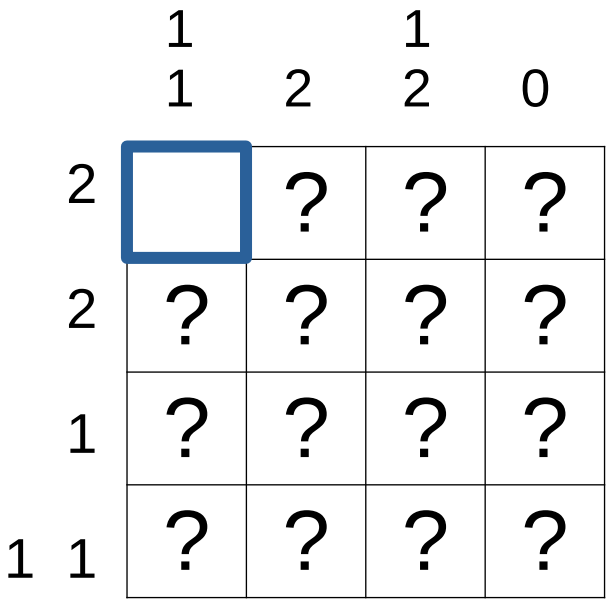
Determiniser l’automate?

Determiniser l’automate	Laisser l’automate non-deterministe
-Creer l’automate est extremement couteux	-Creer l’automate est tres rapide
-Passer un mot à travers l’automate est rapide	-Passer un mot a travres l’automate est plus long

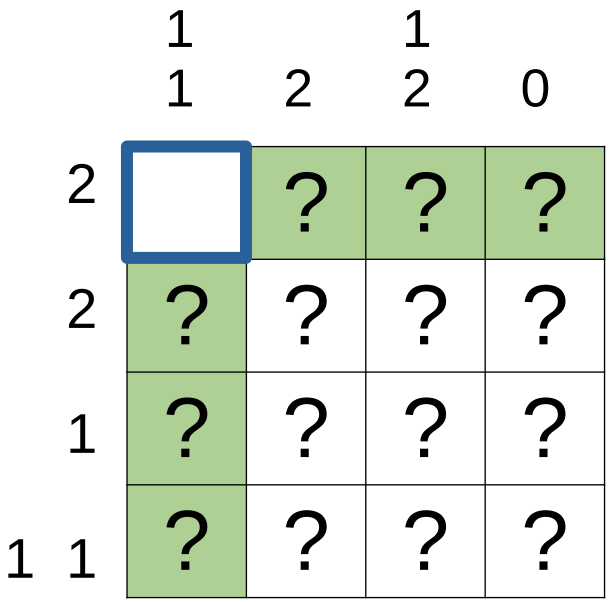
Algorithme de backtracking



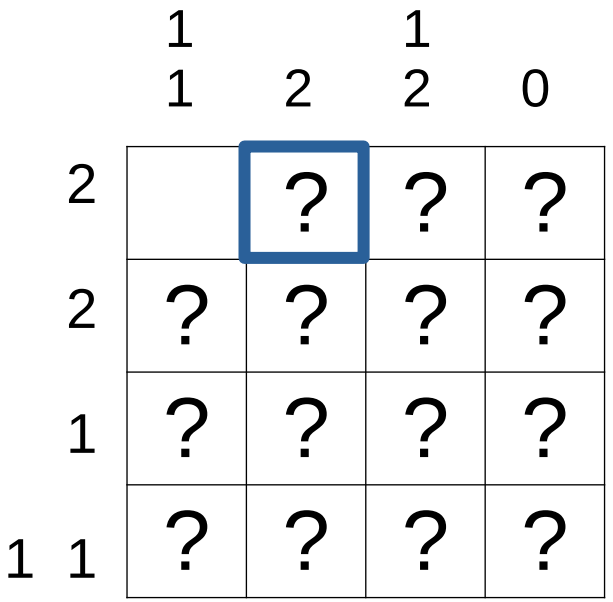
Algorithme de backtracking



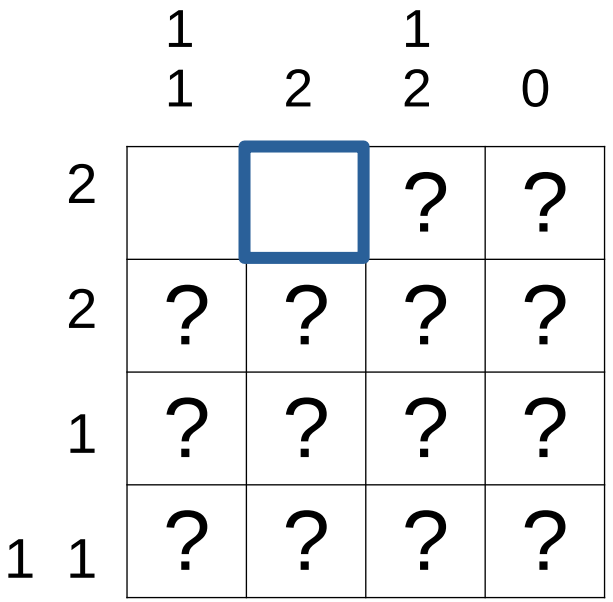
Algorithme de backtracking



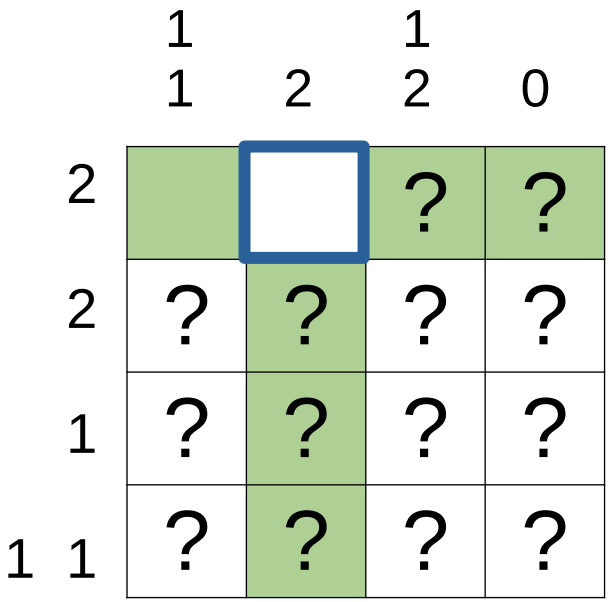
Algorithme de backtracking



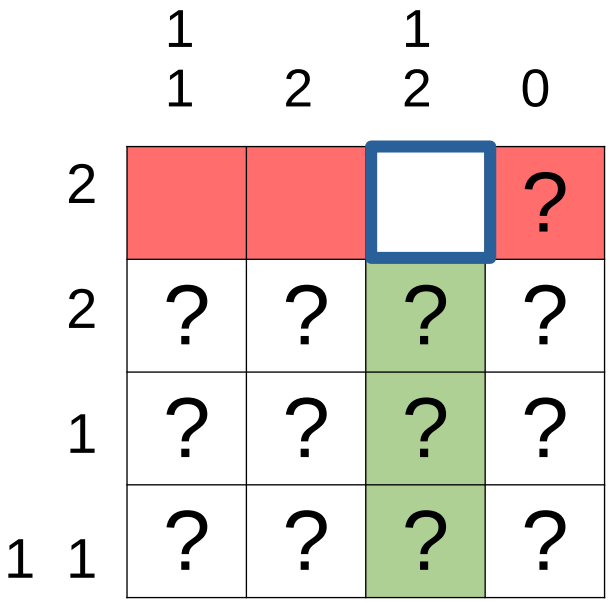
Algorithme de backtracking



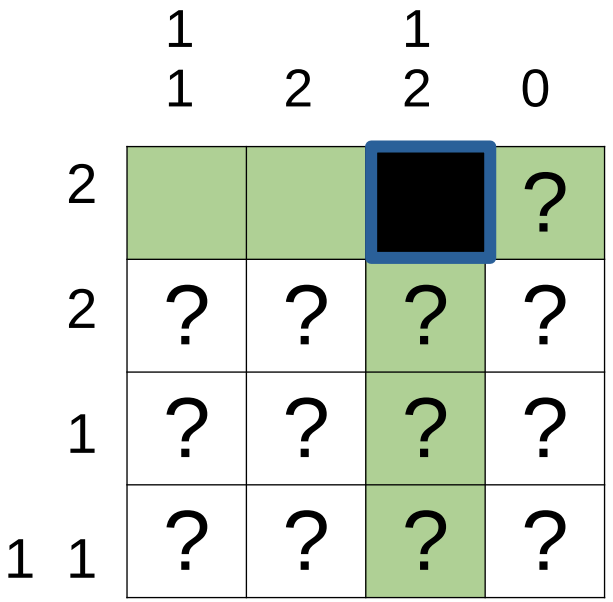
Algorithme de backtracking



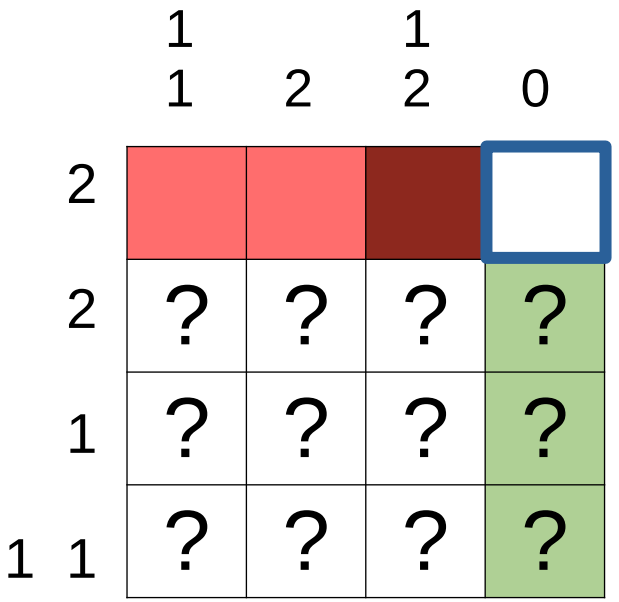
Algorithme de backtracking



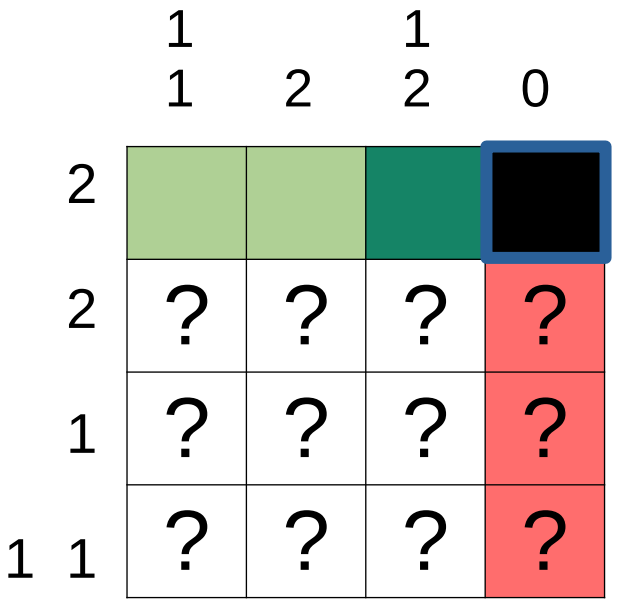
Algorithme de backtracking



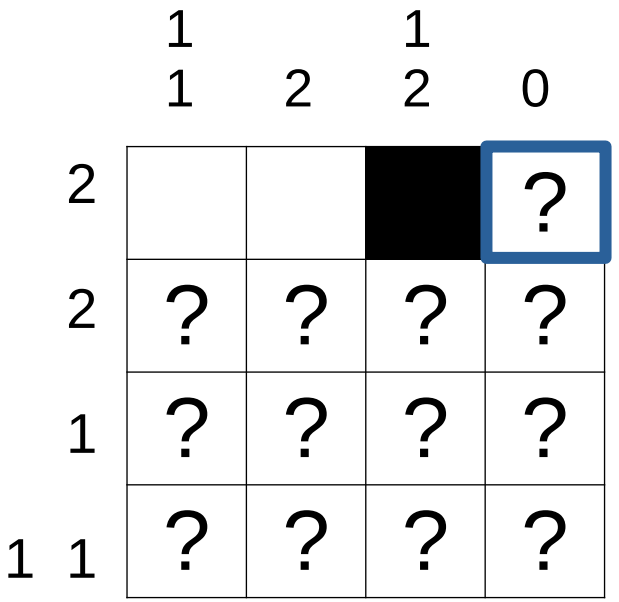
Algorithme de backtracking



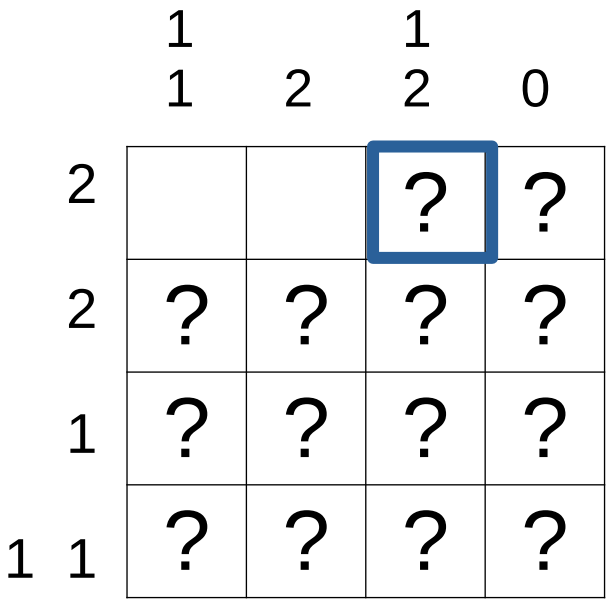
Algorithme de backtracking



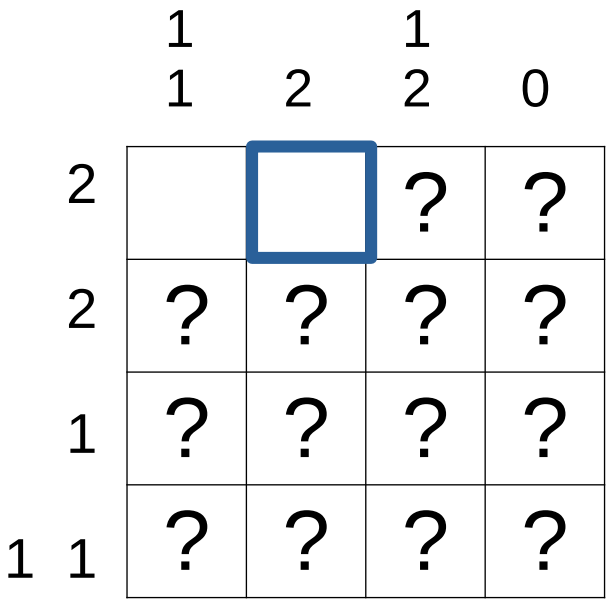
Algorithme de backtracking



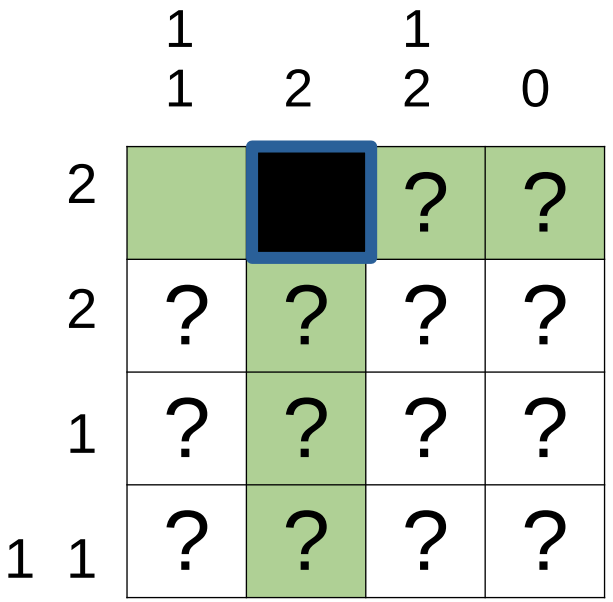
Algorithme de backtracking



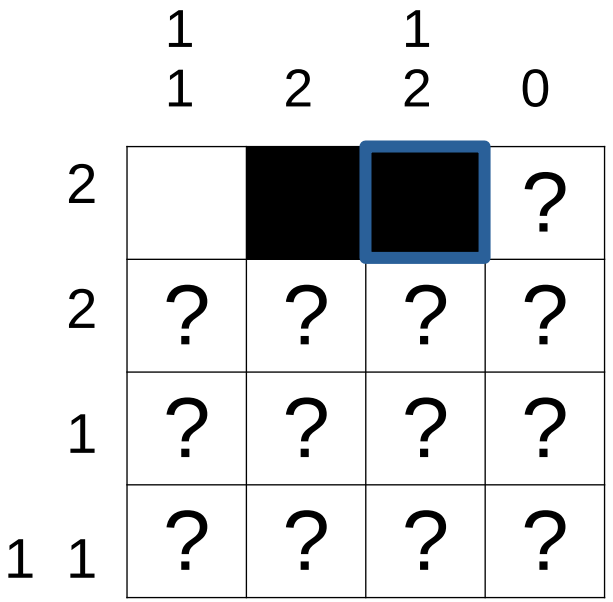
Algorithme de backtracking



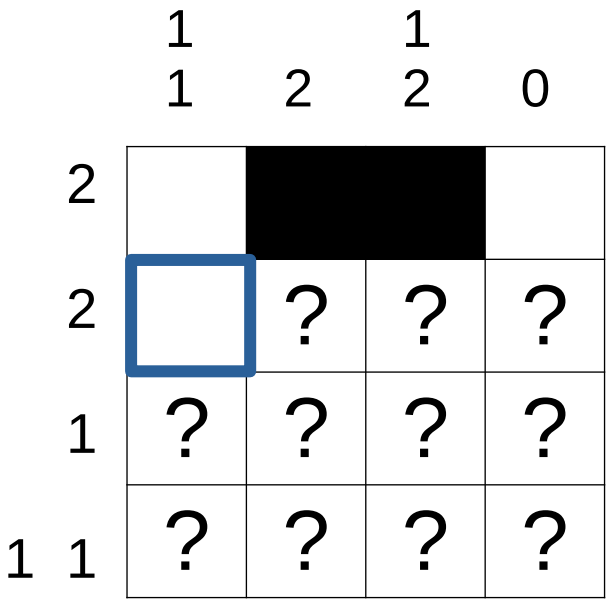
Algorithme de backtracking



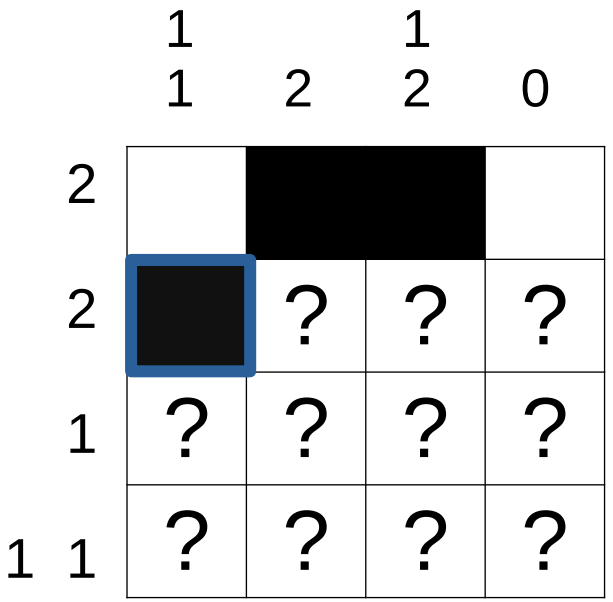
Algorithme de backtracking



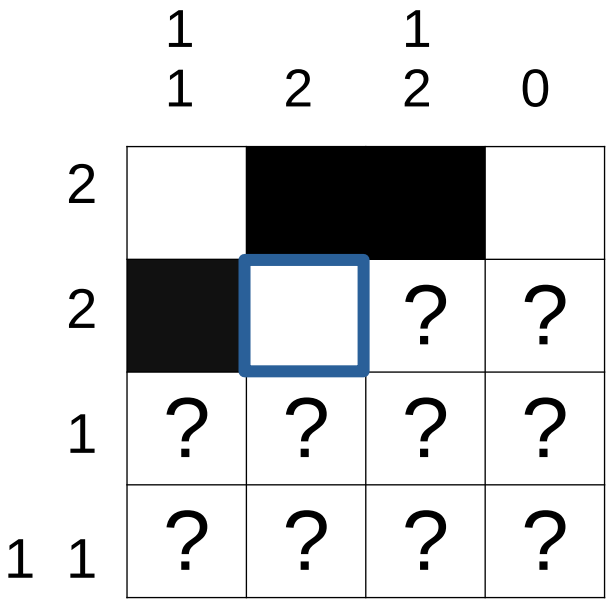
Algorithme de backtracking



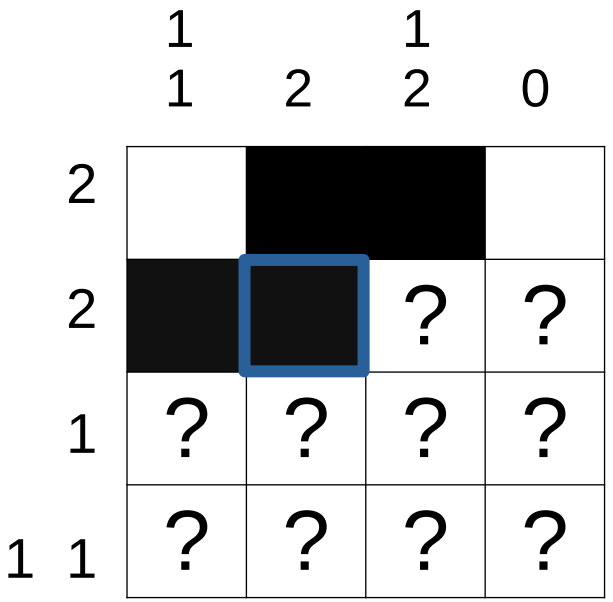
Algorithme de backtracking



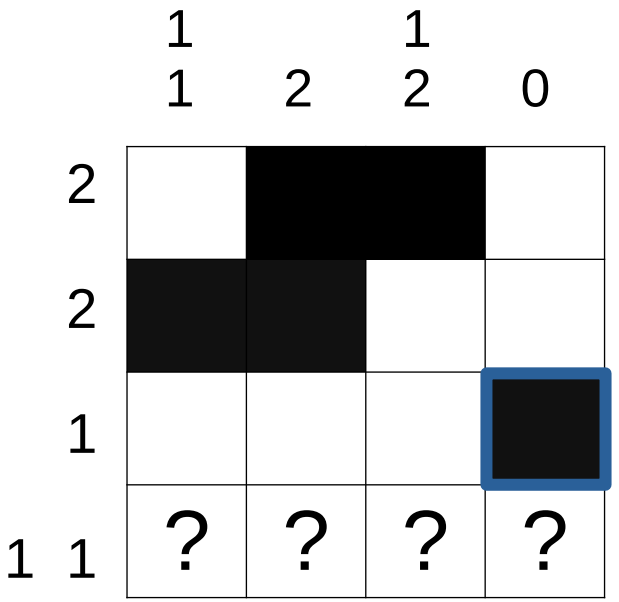
Algorithme de backtracking



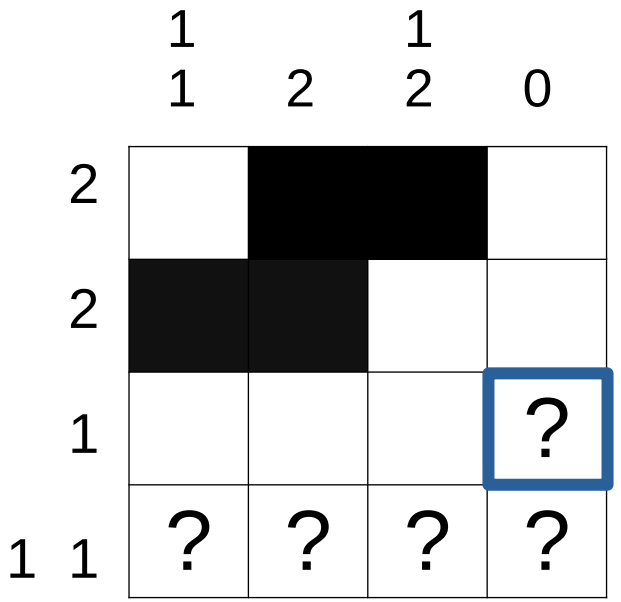
Algorithme de backtracking



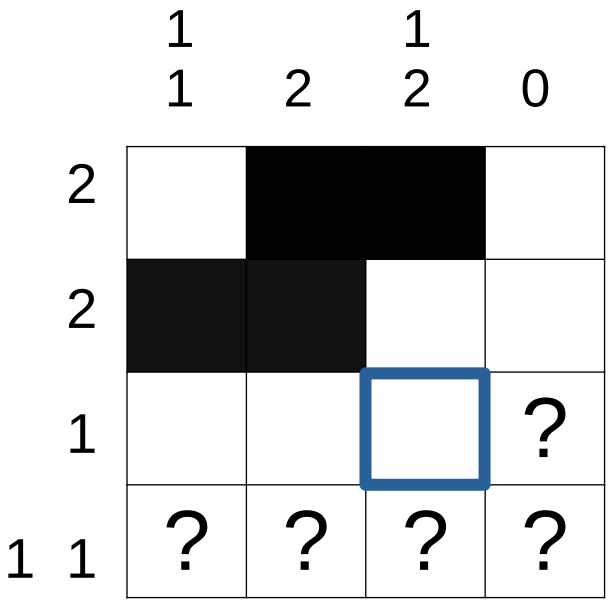
Algorithme de backtracking



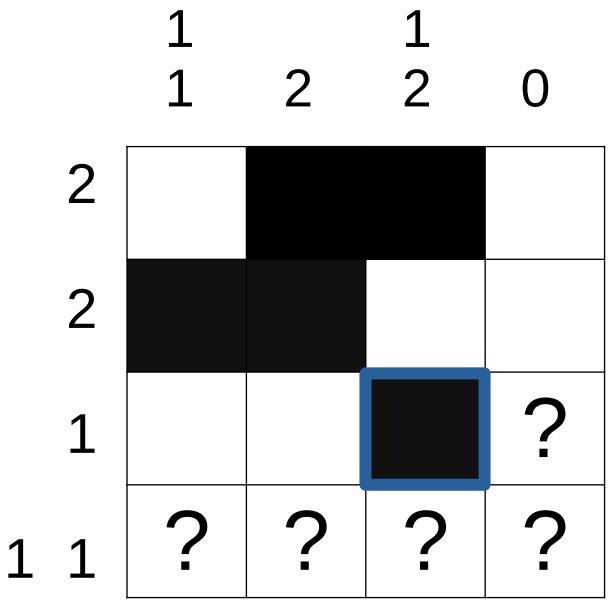
Algorithme de backtracking



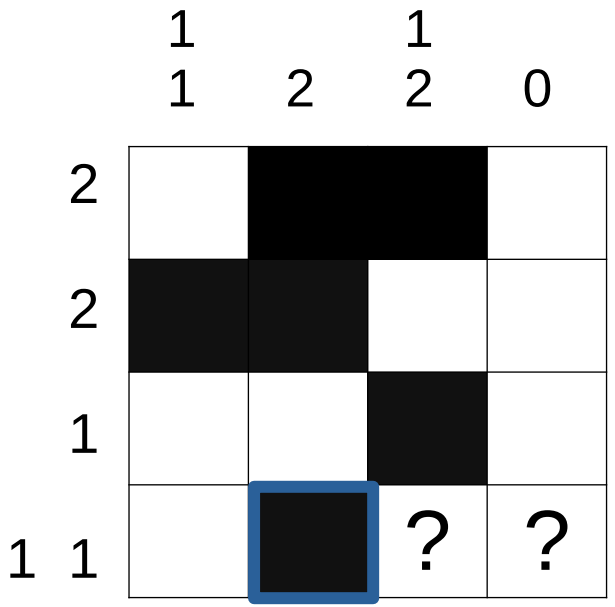
Algorithme de backtracking



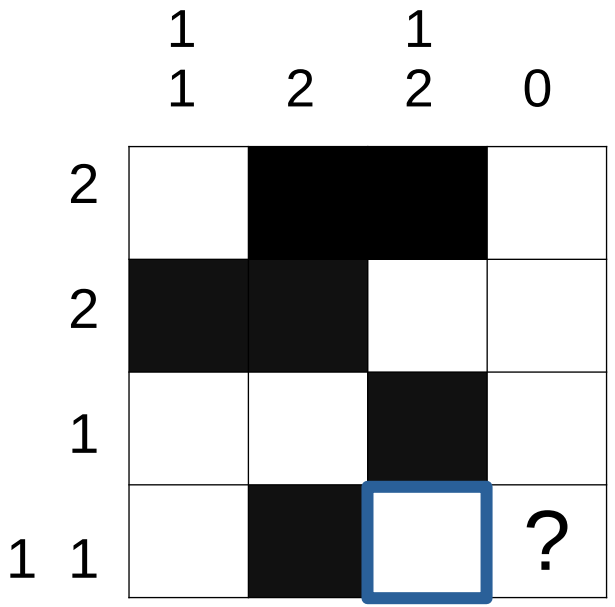
Algorithme de backtracking



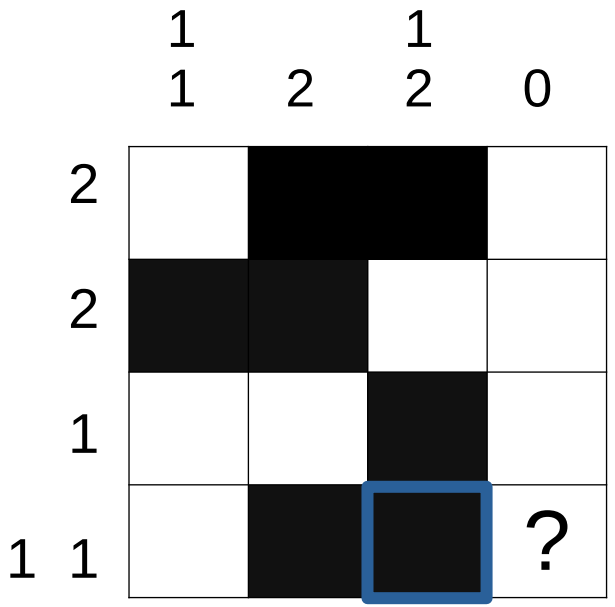
Algorithme de backtracking



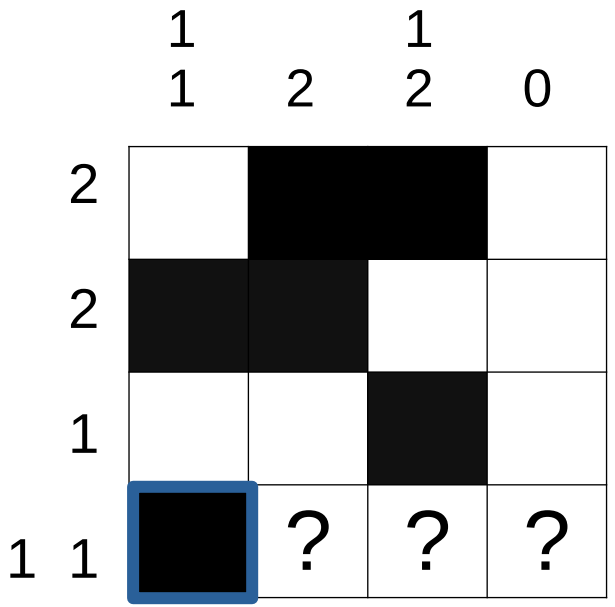
Algorithme de backtracking



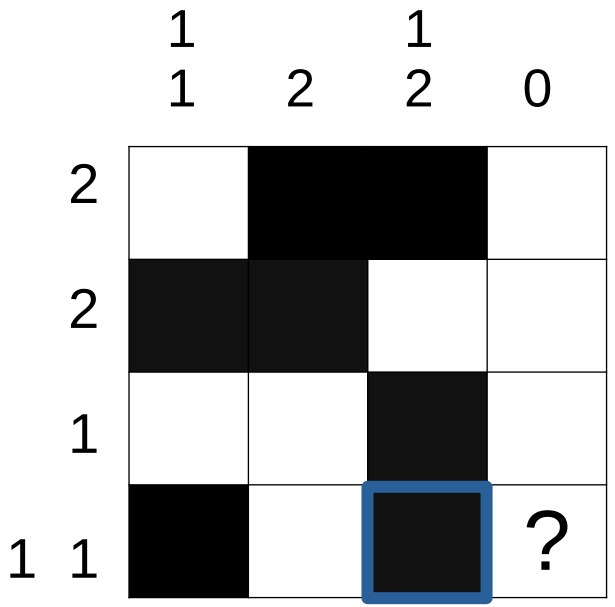
Algorithme de backtracking



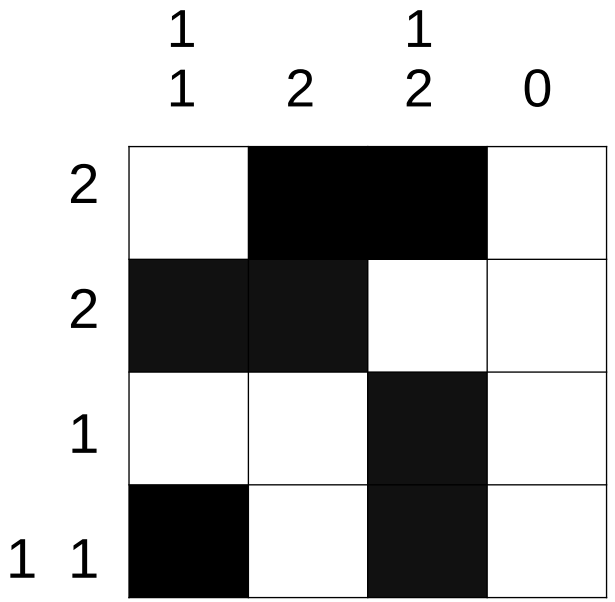
Algorithme de backtracking



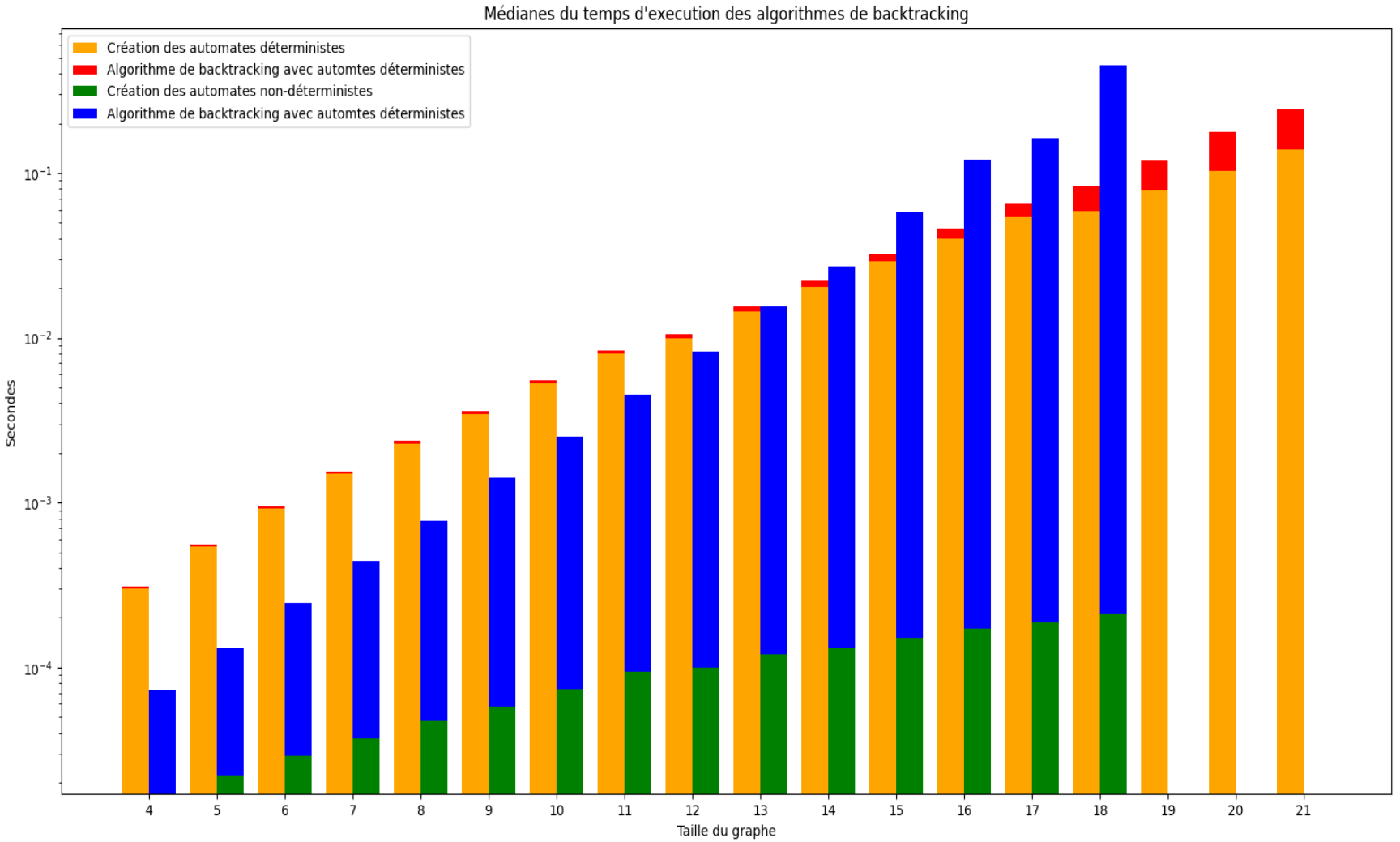
Algorithme de backtracking



Algorithme de backtracking



Algorithme de backtracking



Règles logiques

Les regles logiques sont un moyen aux algorithmes de backtracking d’avoir moins de grilles a parcourir

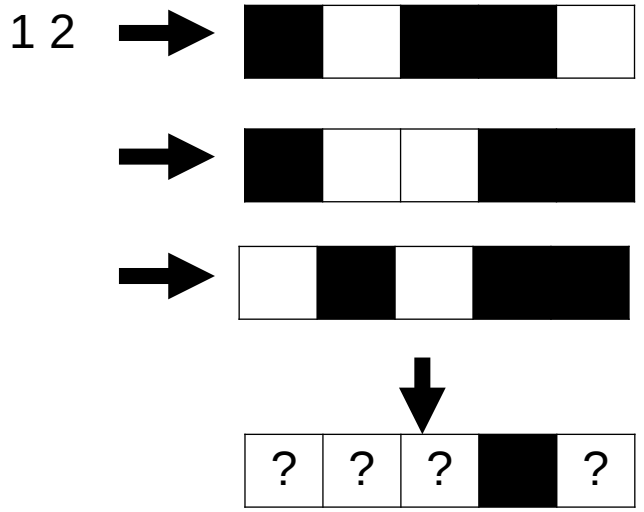
Première regle:
Si une ligne possede comme numéro 0 alors toutes ces cases sont blanches

Règles logiques

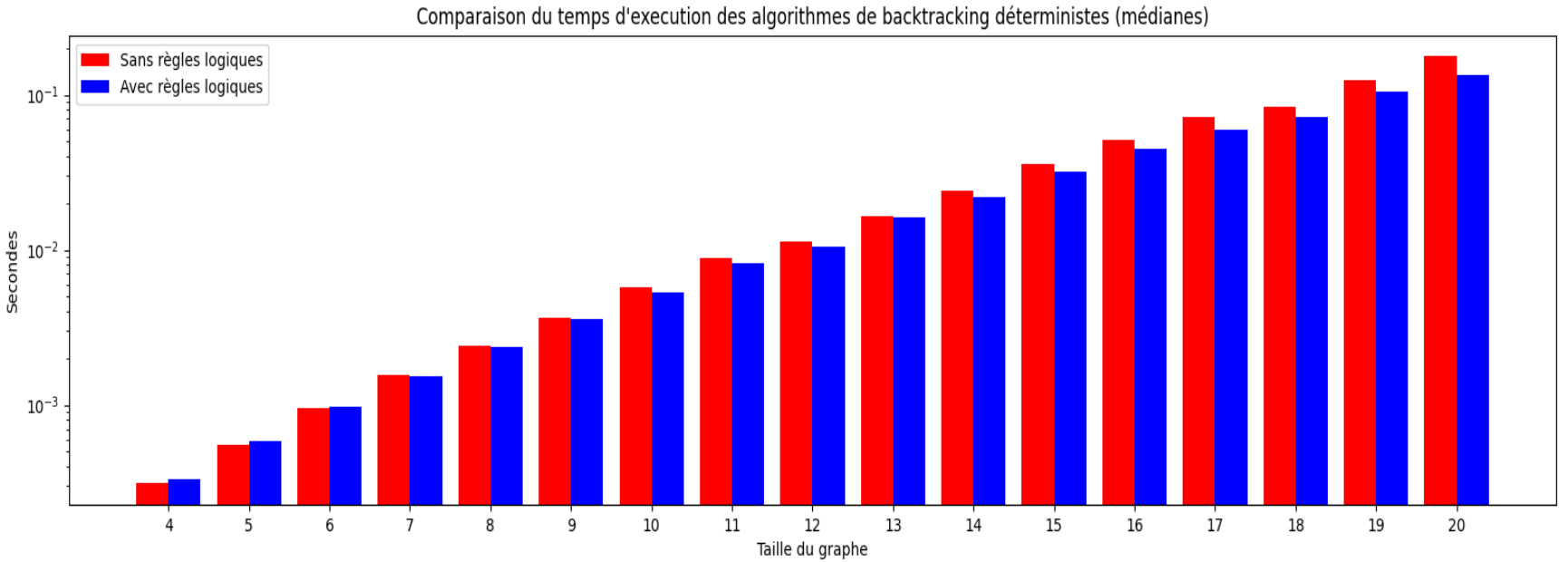
Les regles logiques sont un moyen aux algorithmes de backtracking d'avoir moins de grilles a parcourir

Première regle:
Si une ligne possede comme numéro 0 alors toutes ces cases sont blanches

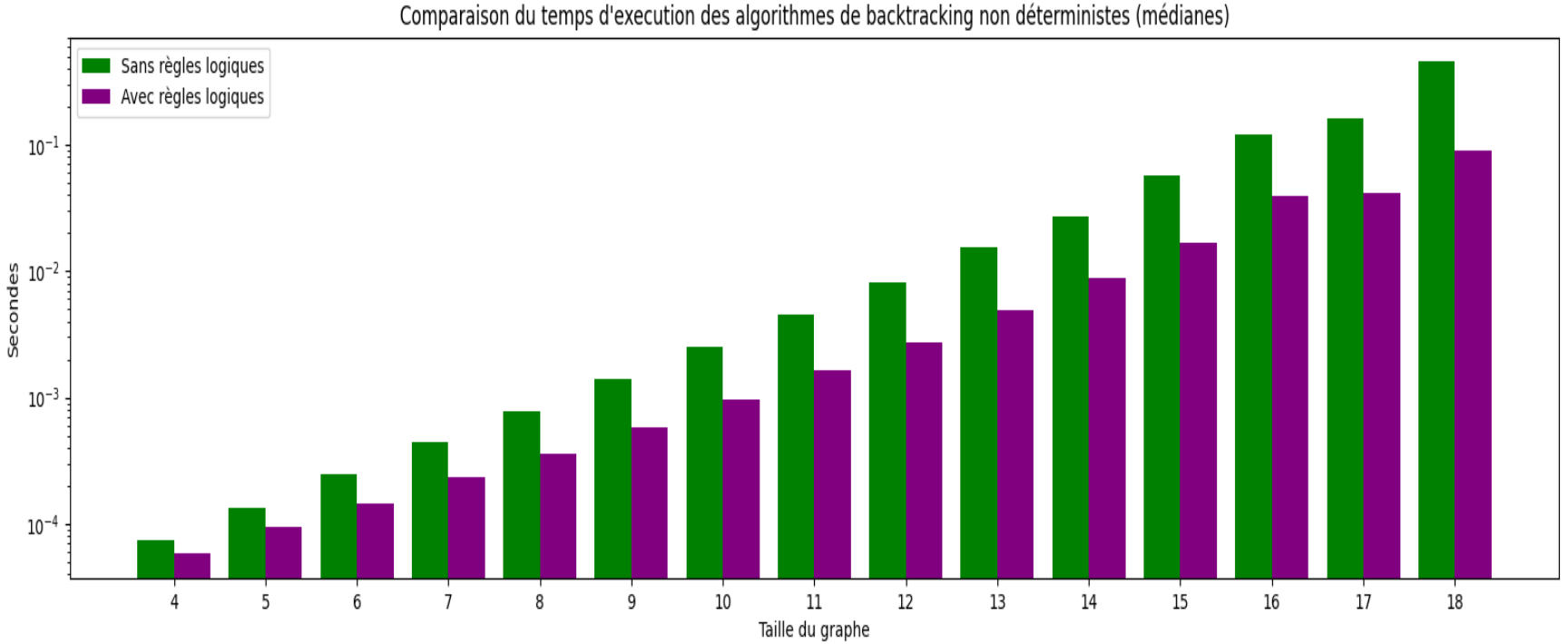
Deuxieme regle:
Si une case est noire dans chaque positions valide de la ligne alors celle ci est noire



Règles logiques



Règles logiques



Conclusion

Pour des grilles de petite tailles:

Utilisation d’un backtracking avec automates non-deterministes

Pour des grilles de grande tailles:

Utilisation d’un backtracking avec automates deterministes

Pistes pour améliorer les algorithmes:

-Utilisation de la mémoïsation

-Ajout de règles logiques

-Utilisation des règles logiques pandant le backtraking

Annexe

Force brute
- -50 000 tests n=3
- -50 000 tests n = 4
- -50 tests n = 5

Backtrack avec automate déterministe
- -50 000 tests pour n de 4 à 11
- -5 000 tests pour n de 12 à 17
- -500 tests pour n = 18
- -100 tests pour n de 19 à 21

Backtrack avec automate non-déterministe
- -50 000 tests pour n de 4 à 11
- -5 000 tests pour n de 12 à 13
- -500 tests pour n de 14 à 15
- -50 tests pour n de 16 à 18

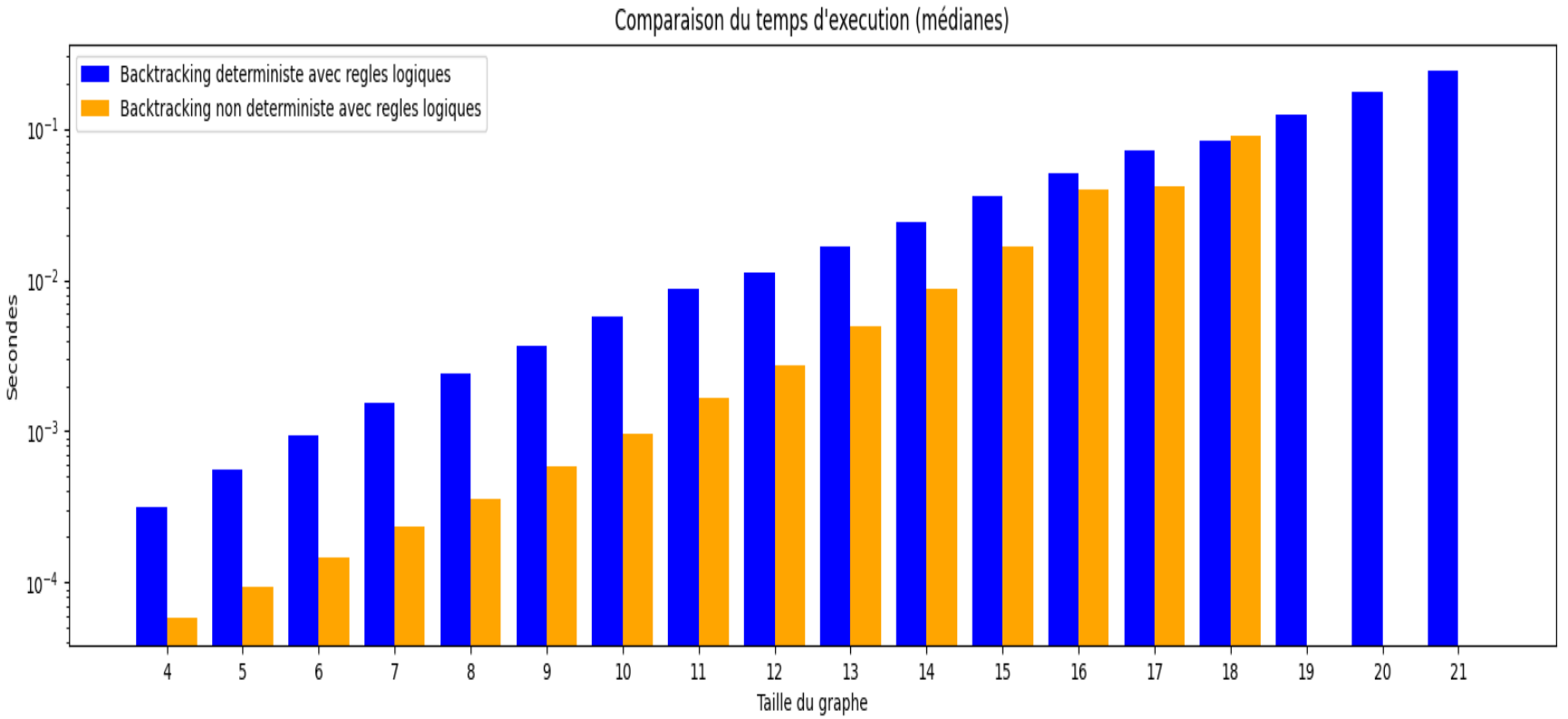
Backtrack avec automate déterministe
Et règles logiques
- -50 000 tests pour n de 4 à 11
- -5 000 tests pour n de 12 à 17
- -1000 tests pour n=18
- -100 tests pour n de 19 à 20

Backtrack avec automate non-déterministe
Et règles logiques
- -50 000 tests pour n de 4 à 11
- -5 000 tests pour n de 12 à 13
- -500 tests pour n de 14 à 15
- -50 tests pour n de 16 à 18

Tests de remplissage des cases par règles logiques
- -50 000 tests par dizaines de pourcent de probabilité pour n = 4,6,8,10
- -5 000 tests par dizaines de pourcent de probabilité pour n = 4,6,8,10

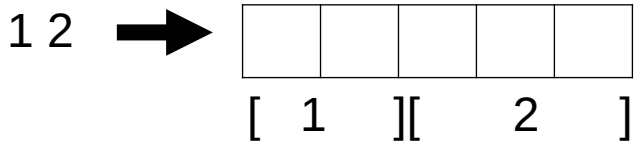
Nombre de tests:3 587 550

Annexe


Comparaison du temps d'execution (médianes)
Backtracking deterministe avec regles logiques
Backtracking non deterministe avec regles logiques
Secondes
10^{-1}
10^{-2}
10^{-3}
10^{-4}
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
Taille du graphe

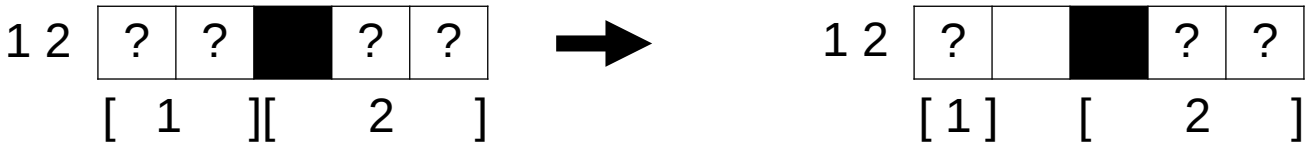
Annexe

La portée d'un bloc est l'intervalle sur lequel le bloc peut etre placé

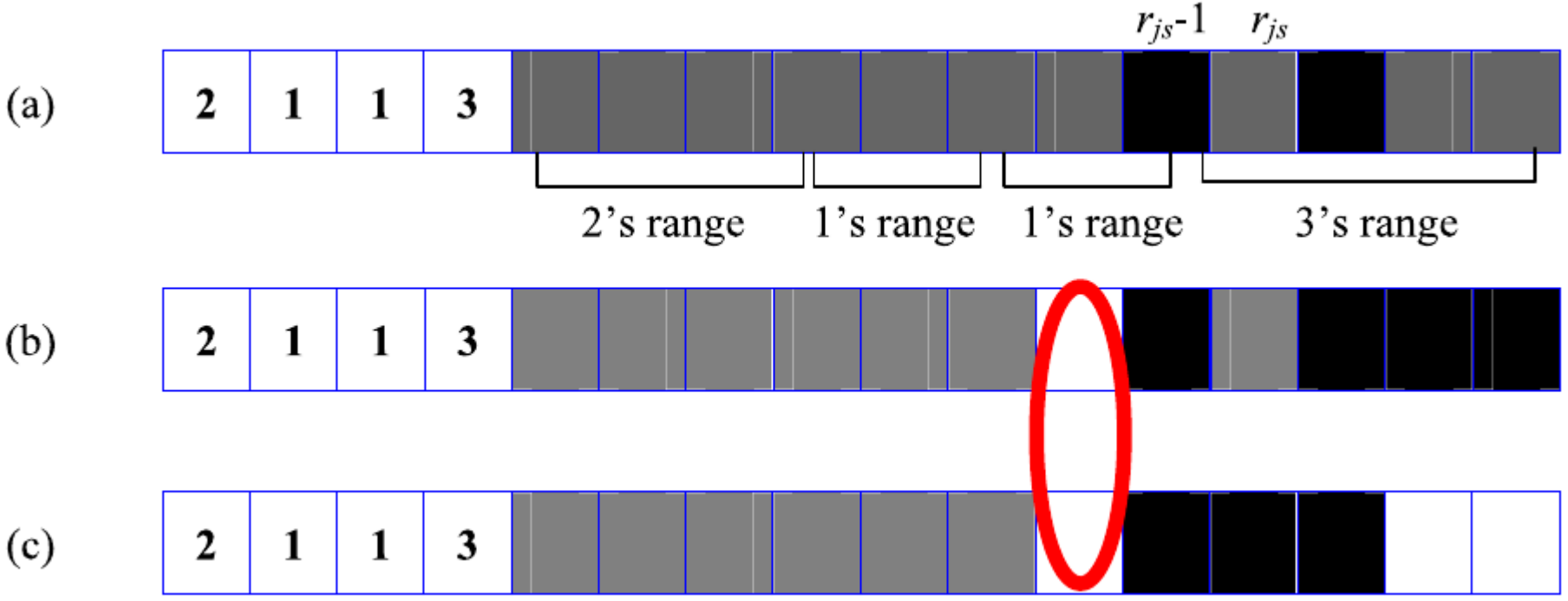


Troisieme regle:
Si un bloc noir est a coté de l'extememité d'une portée d'un bloc,on peut réduire cette portée

Quatrieme regle:
Si une case n'est dans aucune portée, cette case est blanche

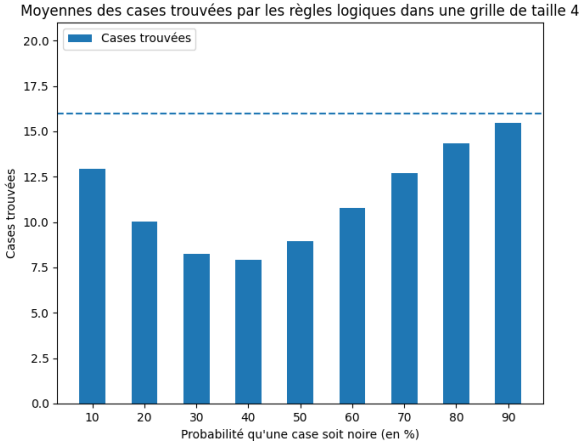


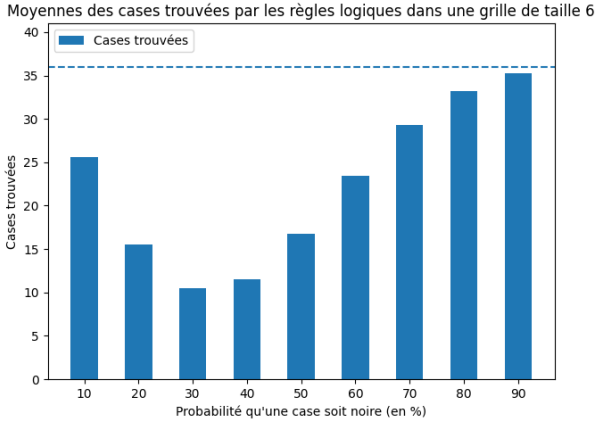
Annexe

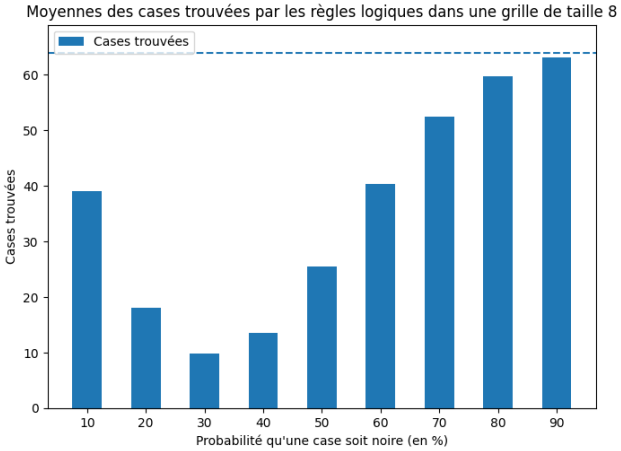


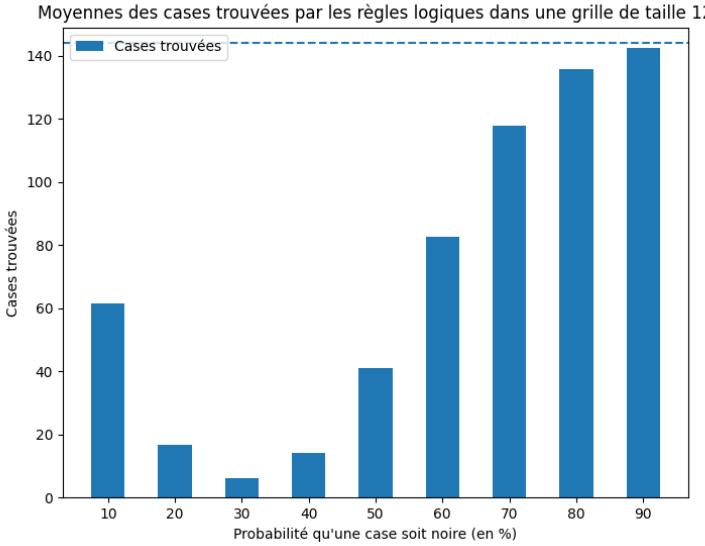
Tirée de “An efficient algorithm for solving nonograms”

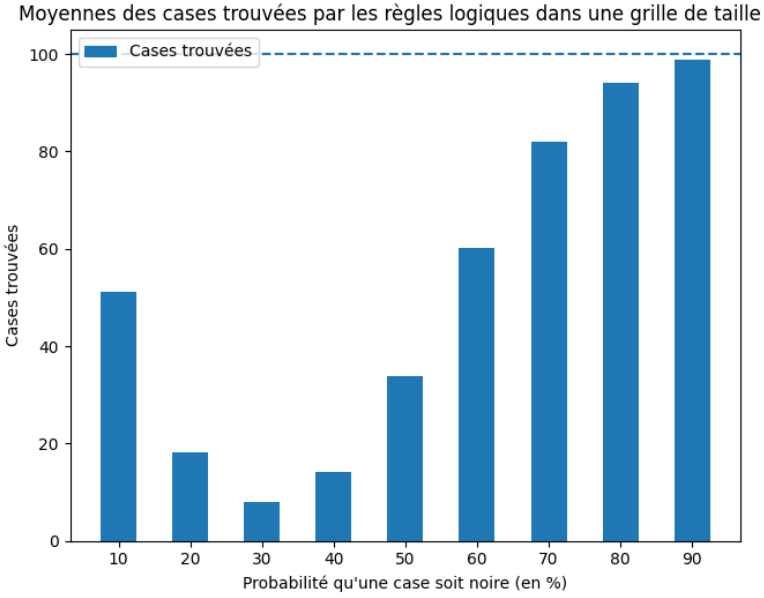
Annexe


Moyennes des cases trouvées par les règles logiques dans une grille de taille 4
Cases trouvées
Cases trouvées
Probabilité qu'une case soit noire (en %)

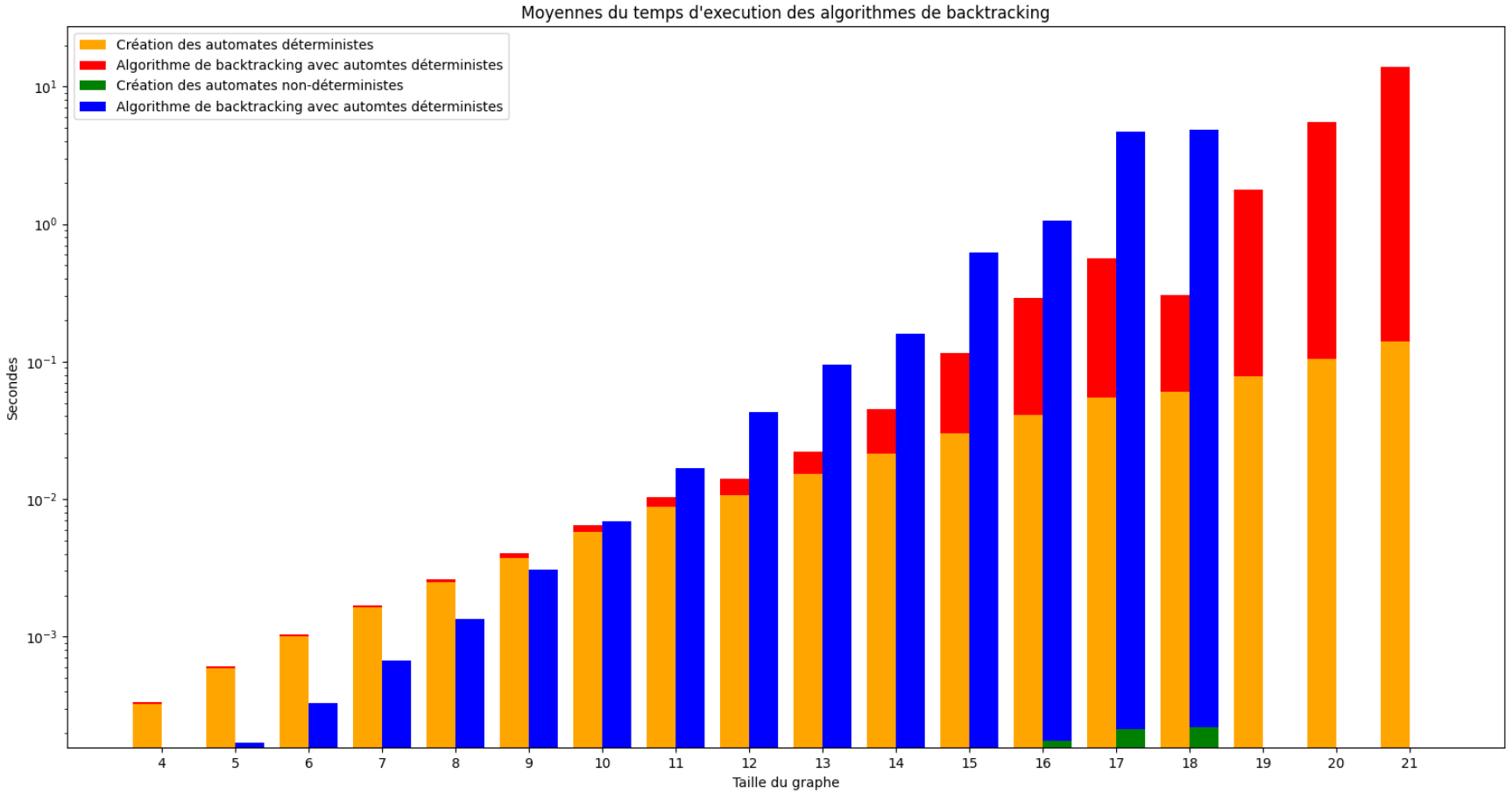

Moyennes des cases trouvées par les règles logiques dans une grille de taille 6
Cases trouvées
Cases trouvées
Probabilité qu'une case soit noire (en %)


Moyennes des cases trouvées par les règles logiques dans une grille de taille 8
Cases trouvées
Cases trouvées
Probabilité qu'une case soit noire (en %)


Moyennes des cases trouvées par les règles logiques dans une grille de taille 12
Cases trouvées
Cases trouvées
Probabilité qu'une case soit noire (en %)


Moyennes des cases trouvées par les règles logiques dans une grille de taille 10
Cases trouvées
Cases trouvées
Probabilité qu'une case soit noire (en %)

Annexe


Moyennes du temps d'execution des algorithmes de backtracking
Création des automates déterministes
Algorithme de backtracking avec automtes déterministes
Création des automates non-déterministes
Algorithme de backtracking avec automtes déterministes
Secondes
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
Taille du graphe

Annexe

Backtrack_50000_11.txt:

….
1715006095 0.007356 0.000000 0.000686 0.008042 -1 1
1715006096 0.008121 0.000000 0.000167 0.008288 -1 1
1715006097 0.005348 0.000000 0.001711 0.007059 -1 1
1715006098 0.007815 0.000000 0.001681 0.009496 -1 1
1715006099 0.007106 0.000000 0.000242 0.007348 -1 1
1715006100 0.007215 0.000000 0.000641 0.007856 -1 1
1715006101 0.009333 0.000000 0.000125 0.009458 -1 1
1715006102 0.004538 0.000000 0.002105 0.006643 -1 1
1715006103 0.008278 0.000000 0.000116 0.008394 -1 1
1715006104 0.009299 0.000000 0.000308 0.009607 -1 1
….

Annexe

```
#ifndef AUTOMATES_H
#define AUTOMATES_H

#include <stdbool.h>
#include "listes.h"
#include "dicts.h"

struct automate_d_s {
    int nb_lettres;
    int nb_etats;
    int depart;
    bool* finaux;
    int** delta;
};
typedef struct automate_d_s automate_d;

struct automate_nd {
    int nb_lettres;
    int nb_etats;
    bool* depart;
    bool* finaux;
    liste** delta;
};
typedef struct automate_nd automate_nd ;

//Initialise un automate deterministe
automate_d* init_automate(int size_alpha,int size_etats);

//Libere un automate
void free_auto(automate_d* a);

//Ajoute une connection dans l'automate
void add_connection_d(automate_d* A,int etat_d,int lettre,int etat_f);

//Delta etoile de l'automate
int delta_etoile_d(automate_d* A,int q,int* input, int size_input);

//Verifie si un mot est reconnu par l'automate
bool reconnu_afd(automate_d* A,int* input, int size_input);

//Affiche l'automate
void print_auto(automate_d* A);

//Initialise un automate non deterministe
automate_nd* init_automate_nd(int size_alpha,int size_etats);

//Ajoute une connection dans l'automate non deterministe
void add_connection_nd(automate_nd *A, int etat_d, int lettre, int etat_f);

//Delta de l'automate non deterministe
bool* delta_nd(automate_nd* A,bool* etats_depart,int lettre);

//Delta etoile de l'Automte non deterministe
bool* delta_etoile_nd(automate_nd* A,bool* etats_depart,int* input,int size_input);

//Verifie si un mot est reconnu par l'automate
bool reconnu_afnd(automate_nd *A, int *input, int size_input);

//Donne tout les etats atteint par un afnd
dict* etats_atteints(automate_nd* A);

//Determinise un automate
automate_d* determiniser(automate_nd* A);
//Affiche l'automate non deterministe
void print_auto_nd(automate_nd* A);

//Libere un automate non det
void free_auto_nd(automate_nd* A);
#endif
```

Annexe

```
#ifndef DICTS_H
#define DICTS_H

#include "listes.h"
#include <stdbool.h>

typedef struct
{   duo_liste* table;
    int length;
} dict ;


dict* create_dict(int n);

duo_liste cons(duo head,duo_liste tail);

void free_duo_liste(duo_liste l);

void add_dict(dict* dico, int key,int val);

bool is_in_dico(dict* dico, int key);

int find_dico(dict* dico, int key);

void remove_dico(dict* dico, int key);

liste all_keys(dict* dico);

void free_dico(dict* dico);

void print_dico(dict* dico);
#endif
```

Annexe

```
#ifndef LISTES_H
#define LISTES_H

//Implemantation basique d'une liste chainee
struct maillon {int val;struct maillon* suivant;};
typedef struct maillon maillon ;
typedef maillon* liste;

typedef struct
{   int x;
    int y;
} duo ;

struct maillon_duo_s
{
    duo val;
    struct maillon_duo_s* suivant;
};

typedef struct maillon_duo_s maillon_duo ;

typedef maillon_duo* duo_liste;


//Libere une liste
void free_liste(liste l);

//Ajoute un element a la liste
liste add_to_liste(int x,liste l);

duo_liste cons(duo head,duo_liste tail);

void free_duo_liste(duo_liste l);

void print_duo(duo d);
//Imprime une listes
void print_liste(liste l);


void print_duo_liste(duo_liste d);
int len_liste(liste l);
int* list_to_tab(liste l);
#endif
```

Annexe

```
#ifndef LOGICRULES_H
#define LOGICRULES_H

#include "listes.h"
#include "picross.h"
typedef struct {
    int nb_blocks;
    duo* est;
}line_est_t;

typedef struct {
    int n;
    line_est_t** lines;
    line_est_t** cols;
} estimation_t;


line_est_t* estimate_line(liste numbers,int n);
estimation_t* full_estimation(picross_numbers* nums);
void print_full_estimation(estimation_t* e);
void print_estimation(line_est_t* e);
void free_estimation(line_est_t* e);
void free_full_estimation(estimation_t* e);
int apply_rules(picross_grid* grille_a_completer,picross_numbers*
nums,estimation_t* est,int k);
#endif
```

Annexe

```
#ifndef PICROSS_H
#define PICROSS_H

#include "automates.h"
#include "listes.h"


struct picross_grid_s {int size; int** grid;};
typedef struct picross_grid_s picross_grid;

struct picross_numbers_s {liste* lig;liste* col;int size;};
typedef struct picross_numbers_s picross_numbers;

//Genere une grille vide
picross_grid* gen_empty_grid(int size);

//Genere une grille inconnue
picross_grid* gen_unk_grid(int size);

//Genere une grille aleatoire avec chance% que chaque case soit noire
picross_grid* gen_random_grid(int size, int chance);

//Fait transposer la grille tel une matrice
picross_grid* tourner_grille(picross_grid* grid);

int* get_col(picross_grid* g, int size,int col_n);

//Affiche une grille de picross
void print_picc(picross_grid* p);

//Affiche les nombres des lignes et colonnes
void print_nums(picross_numbers* nums);

//prend une grille et en tire ses nombres
picross_numbers* gen_numbers_from_grid(picross_grid* grid);

void apply_tab_to_col(picross_grid* g,int* col,int col_n);

//Libere une grille
void free_picross(picross_grid* p);

//Libere les nombres
void free_numbers(picross_numbers* nums);


#endif
```

Annexe

```
#ifndef SOLVER_H
#define SOLVER_H

#include "picross.h"
#include "valideurs.h"

//Verifie si une grille complete est valide O(n^2)
bool est_solution_valide_total(picross_grid* grid,valideur_det* valideur);

//Verifie une seule ligne et colonne O(n)
bool verif_ligne_col(picross_grid* grid,valideur_det* valideur,int ligne,int col);

bool verif_ligne_col_ndet(picross_grid* grid,valideur_ndet* valideur,int ligne,int
col);

//Brute-force O(2^(n^2))
bool bruteforce(picross_grid* grid,valideur_det* valideur,int i,int j);


//Backtracking
bool backtracking(picross_grid* grid,valideur_det* valideur,int i,int j);

bool backtracking_ndet(picross_grid* grid, valideur_ndet* valideur, int i, int j);
#endif
```

Annexe

```
#ifndef UTILS_H
#define UTILS_H

#include "listes.h"
#include <stdbool.h>


#define Fl(n) (int)1 << n


//Libere un tableau de tableau d'entier
void free_int_int(int** t,int n);


//Affiche un tableau d'entier
void print_tab(int* t, int size);

//Affiche un tableau de booleen
void print_bool_tab(bool* t, int size);

//Converti un tableau de bool en un entier unique a ce tableau
int binary_from_bool_int(bool* t,int size);

//Fait l'inverse
bool* bool_arr_from_int(int num,int size);

//Envoie 1 quand envoye 0 et 0 quand envoiye 1
int inverse_valeur(int i);

//Operation binaire AND sur un array
bool and_bool_arr(bool* a1,bool* a2,int size);

bool equal_bool_arr(bool* a1,bool* a2,int size);

void copy_tab(int* dest,int* src,int size);
#endif
```

Annexe

```
#ifndef VALIDEURS_H
#define VALIDEURS_H

#include "picross.h"
#include "automates.h"

struct valideur_d_s {automate_d** ligne;automate_d** col;int size;};
typedef struct valideur_d_s valideur_det;

struct valideur_nd_s {automate_nd** ligne;automate_nd** col;int size;};
typedef struct valideur_nd_s valideur_ndet ;


//Automate qui reconnais une ligne vide
automate_d* auto_de_zeros(void);

//Automate partiel qui recconais une ligne vide
automate_nd* auto_nd_zeros(void);

//Genere un automate qui recconais la ligne donnee
automate_d* generer_automate_ligne(liste ligne);

//Genere tous les automates poutes tt lignes et colonnes
valideur_det* gen_valideur_total(picross_numbers* nums);

//Genere l'automate partiel d'une ligne
automate_nd* generer_automate_partiel_ligne(liste ligne);

//Genere un valideur non det
valideur_ndet* gen_valideur_ndet(picross_numbers* nums);

//Genere le valideur partiel
valideur_det* gen_valideur_partiel(picross_numbers* nums);


//Libere le valideur
void free_valideur_det(valideur_det* A);

//Libere un vlaideur ndet
void free_valideur_ndet(valideur_ndet *A);

#endif
```

Annexe

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>

#include "automates.h"
#include "listes.h"
#include "utils.h"
#include "dicts.h"

automate_d* init_automate(int size_alpha,int size_etats){
    automate_d* res = (automate_d*)malloc(sizeof(automate_d));
    int** delta_res = (int**)malloc(sizeof(int*)*size_etats);
    for (int i=0;i<size_etats;i++){
        int* delta_bis = (int*)malloc(sizeof(int)*size_alpha);
        for (int j=0;j<size_alpha;j++){
            delta_bis[j] = -1;
        }
        delta_res[i] = delta_bis;
    }
    bool* finaux_res = (bool*)calloc(sizeof(bool),size_etats);

    res->nb_etats = size_etats;
    res->nb_lettres = size_alpha;
    res->depart = 0;
    res->finaux = finaux_res;
    res->delta = delta_res;
    return res;
}

void free_auto(automate_d* a){
    free_int_int(a->delta, a->nb_etats);
    free(a->finaux);
    free(a);
}


void add_connection_d(automate_d* A,int etat_d,int lettre,int etat_f){
    A->delta[etat_d][lettre] = etat_f;
}

int delta_etoile_d(automate_d* A,int q,int* input,int size_input){
    int etat_curr = q;
    int i = 0;
    while (i < size_input && etat_curr != -1){
        etat_curr = A->delta[etat_curr][input[i]];
        i++;
    }
    return etat_curr;

}

bool reconnu_afd(automate_d* A,int* input,int size_input){
    int etat_final = delta_etoile_d(A, A->depart,input,size_input);
    //print_tab(input, size_input);
    //printf("ETAT FIN:%d\n",etat_final);
    //print_auto(A);
    if (etat_final == -1){
        return false;
    }
    else
    {
    return A->finaux[etat_final];
    }
}

void print_auto(automate_d* A){
    printf("\n");
    printf("Nb delta: %d\n",A->nb_lettres);
    printf("Nb etats: %d\n",A->nb_etats);

    printf("Etat depart: %d\n",A->depart);
    printf("Etat finaux:");print_bool_tab(A->finaux,A->nb_etats);printf("\n");
    for (int i=0;i<A->nb_etats;i++){
        print_tab(A->delta[i],A->nb_lettres);
    }
    printf("\n");
}

automate_nd* init_automate_nd(int size_alpha,int size_etats){
    automate_nd* res = (automate_nd*)malloc(sizeof(automate_nd));
    liste** delta_res = (liste**)malloc(sizeof(liste*)*size_etats);
    for (int i=0;i<size_etats;i++){
        liste* delta_res_bis = (liste*)calloc(size_alpha,sizeof(liste));
        delta_res[i] = delta_res_bis;
    }
    bool* finaux_res = (bool*)calloc(size_etats,sizeof(bool));
    bool* depart_res = (bool*)calloc(size_etats,sizeof(bool));
    res->nb_etats = size_etats;
    res->nb_lettres = size_alpha;
    res->depart = depart_res;
    res->finaux = finaux_res;
    res->delta = delta_res;
    return res;

}
```

Annexe

```
void add_connection_nd(automate_nd *A, int etat_d, int lettre, int etat_f){
    A->delta[etat_d][lettre] = add_to_liste(etat_f, A->delta[etat_d][lettre]);
}

        }
        printf("]");
    }
    printf("\n");
}

void free_auto_nd(automate_nd* A){
    for(int i=0;i<A->nb_etats;i++){
        for (int j=0;j<A->nb_lettres;j++){
            free_liste(A->delta[i][j]);
        }
        free(A->delta[i]);
    }
    free(A->delta);
    free(A->depart);
    free(A->finaux);
    free(A);
bool* delta_nd(automate_nd* A,bool* etats_depart,int lettre){
    bool* res = (bool*)calloc(A->nb_etats,sizeof(bool));
    for(int i=0;i<A->nb_etats;i++){
        if (etats_depart[i]){
            liste liste_to_parse = A->delta[i][lettre];
            while (liste_to_parse != NULL){
                res[liste_to_parse->val] = true;
                liste_to_parse = liste_to_parse->suivant;
            }
        }
    }
    return res;
}

bool* delta_etoile_nd(automate_nd* A,bool* etats_depart,int* input,int size_input){
    bool* etat_curr = (bool*)malloc(sizeof(bool)*A->nb_etats);
    memcpy(etat_curr,etats_depart, sizeof(bool)*A->nb_etats);
    for (int i=0;i<size_input;i++){
        bool* new_etat = delta_nd(A,etat_curr,input[i]);
        free(etat_curr);
        etat_curr = new_etat;
    }
    return etat_curr;
}

bool reconnu_afnd(automate_nd *A, int *input, int size_input){
    bool* res_tab = delta_etoile_nd(A,A->depart,input,size_input);
    bool res = false;
    for (int i=0;i<A->nb_etats;i++){
        res = res || (A->finaux[i] && res_tab[i]);
    }
    free(res_tab);
    return res;
}
```

```
dict* etats_atteints(automate_nd* A){
    int num_etat = 0;
    dict* res = create_dict(A->nb_etats * 5);
    liste a_voir = NULL;
    liste old_liste = NULL;
    a_voir = add_to_liste(binary_from_bool_int(A->depart,A->nb_etats),a_voir);
    while (a_voir != NULL){
        int elem_a_voir = a_voir->val;
        old_liste = a_voir;
        a_voir = old_liste->suivant;
        free(old_liste);
        if (!is_in_dico(res,elem_a_voir)){
            bool* etat_depart = bool_arr_from_int(elem_a_voir,A->nb_etats);
            for (int a = 0;a<A->nb_lettres;a++){
                bool* etat_fin = delta_nd(A,etat_depart,a);
                int etat_fin_int = binary_from_bool_int(etat_fin, A->nb_etats);
                a_voir = add_to_liste(etat_fin_int,a_voir);
                free(etat_fin);

            }
            add_dict(res,elem_a_voir,num_etat);
            num_etat++;
            free(etat_depart);
        }
    }
    free_liste(a_voir);
    return res;
}
```

```
automate_d* determiniser(automate_nd* A){
    dict* atteints = etats_atteints(A);
    liste keys = all_keys(atteints);
    int nb_etats = 0;
    liste keys_copy = keys;
    while (keys_copy != NULL){
        duo_liste dl = atteints->table[keys_copy->val];
        while (dl != NULL){
            nb_etats++;
            dl = dl->suivant;
        }
        keys_copy = keys_copy->suivant;
    }
    automate_d* res = init_automate(A->nb_lettres,nb_etats);
    keys_copy = keys;
    while (keys_copy != NULL){
        duo_liste dl = atteints->table[keys_copy->val];
        while (dl != NULL){
            int etat_depart_int = dl->val.x;
            bool* etat_depart = bool_arr_from_int(etat_depart_int,A->nb_etats);
            int nom_etat_depart = dl->val.y;
            for (int a=0;a<A->nb_lettres;a++){
                bool* etats_atteints = delta_nd(A,etat_depart, a);
                //if (etat_depart_int == 15){
                //    print_dico(atteints);
                //    printf("%d",nb_etats);
                //    print_liste(keys);
                //}
                int etats_atteints_int = binary_from_bool_int(etats_atteints,A->nb_etats);
                int nom_etat_atteints = find_dico(atteints, etats_atteints_int);
                add_connection_d(res, nom_etat_depart, a, nom_etat_atteints);
                free(etats_atteints);
            }
            if (and_bool_arr(etat_depart, A->finaux, A->nb_etats)){
                res->finaux[nom_etat_depart] = true;
            }
            if (equal_bool_arr(etat_depart, A->depart, A->nb_etats)){
                res->depart = nom_etat_depart;
            }
            free(etat_depart);
            dl = dl->suivant;
        }
        keys_copy = keys_copy->suivant;
    }
    free_dico(atteints);
    free_liste(keys);
    return res;


}

void print_auto_nd(automate_nd* A){
    printf("\n");
    printf("Nb delta: %d\n",A->nb_lettres);
    printf("Nb etats: %d\n",A->nb_etats);

    printf("Etat depart:");print_bool_tab(A->depart,A->nb_etats);printf("\n");
    printf("Etat finaux:");print_bool_tab(A->finaux,A->nb_etats);printf("\n");
    for (int i=0;i<A->nb_etats;i++){
        printf("[");
        for (int j=0;j<A->nb_lettres;j++){
            print_liste(A->delta[i][j]);
        }
        printf("]");
    }
    printf("\n");
}

void free_auto_nd(automate_nd* A){
    for(int i=0;i<A->nb_etats;i++){
        for (int j=0;j<A->nb_lettres;j++){
            free_liste(A->delta[i][j]);
        }
        free(A->delta[i]);
    }
    free(A->delta);
    free(A->depart);
    free(A->finaux);
    free(A);
```

Annexe

```
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>

#include "dicts.h"
#include "listes.h"

dict* create_dict(int n){
    dict* res = (dict*)malloc(sizeof(dict));
    duo_liste* table = (duo_liste*)malloc(sizeof(duo_liste)*n);
    for (int i=0;i<n;i++){
        table[i] = NULL;
    }
    res->table = table;
    res->length = n;

    return res;
}




void add_dict(dict* dico, int key,int val){
    int array_index = key % dico->length;
    duo couple;
    couple.x = key;
    couple.y = val;
    dico->table[array_index] = cons(couple,dico->table[array_index]);
}

bool is_in_dico(dict* dico, int key){
    int array_index = key % dico->length;
    duo_liste l = dico->table[array_index];
    while (l != NULL){
        if (l->val.x == key){
            return true;
        }
        l = l->suivant;
    }
    return false;
}

int find_dico(dict* dico, int key){
    int array_index = key % dico->length;
    duo_liste l = dico->table[array_index];
    while (l != NULL){
        if (l->val.x == key){
            return l->val.y;
        }
        l = l->suivant;
    }
    return -1;
}

void remove_dico(dict* dico, int key){
    int array_index = key % dico->length;
    duo_liste l = dico->table[array_index];
    duo_liste prev = l;
    while (l != NULL){
        if (l->val.x == key){
            prev->suivant = l->suivant;
            free(l);
        }
        prev = l;
        l = l->suivant;
    }
    return;
}
```

```
void replace_dico(dict* dico,int key,int val){
    if (is_in_dico(dico,key)){
        remove_dico(dico,key);
    }
    add_dict(dico,key,val);
}

liste all_keys(dict* dico){
    liste res = NULL;
    for (int i=0; i<dico->length;i++){
        if (dico->table[i] != NULL){
            res = add_to_liste(i,res);
        }
    }
    return res;
}

void free_dico(dict* dico){
    for (int i=0;i<dico->length;i++){
        free_duo_liste(dico->table[i]);
    }
    free(dico->table);
    free(dico);
}

void print_dico(dict* dico){
    for (int i=0; i<dico->length;i++){
        printf("%d:",i);
        print_duo_liste(dico->table[i]);
        printf("\n");
    }
}
```

Annexe

```c
#include <stdlib.h>
#include <stdio.h>


#include "listes.h"

void free_liste(liste l){
    if (l != NULL){
        free_liste(l->suivant);
        free(l);
    }
}

liste add_to_liste(int x,liste l){
    liste res = (liste)malloc(sizeof(maillon));
    res->val = x;
    res->suivant = l;
    return res;
}

void print_liste(liste l){
    printf("[");
    liste liste_to_print = l;
    if (l==NULL){
        printf("NULL");
    }
    while (liste_to_print != NULL){
        printf("%d ",liste_to_print->val);
        liste_to_print = liste_to_print->suivant;
    }
    printf("]");
}

duo_liste cons(duo head,duo_liste tail){
    duo_liste res = (duo_liste)malloc(sizeof(maillon_duo));
    res->val = head;
    res->suivant = tail;
    return res;
}

void free_duo_liste(duo_liste l){
    if (l != NULL){
        free_duo_liste(l->suivant);
        free(l);
    }
}

void print_duo_liste(duo_liste d){
    duo_liste d_cpy = d;
    while (d_cpy){
        printf("{%d %d} ",d_cpy->val.x,d_cpy->val.y);
        d_cpy = d_cpy->suivant;
    }
    return;

}

void print_duo(duo d){
    printf("(%d,%d)",d.x,d.y);
}
```

```c
int len_liste(liste l){
    int res =0;
    liste l_c = l;
    while (l_c != NULL){
        res++;
        l_c = l_c->suivant;
    }
    return res;
}

int* list_to_tab(liste l){
    int n = len_liste(l);
    int* res = (int*)malloc(sizeof(int)*n);
    liste l_c = l;
    for (int i = 0;i<n;i++){
        res[i] = l_c->val;
        l_c =  l_c->suivant;
    }
    return res;
}
```

Annexe

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#include "utils.h"
#include "picross.h"
#include "logicrules.h"
#include "listes.h"


line_est_t* estimate_line(liste numbers,int n){
    if (numbers == NULL){
        return NULL;
    }
    int nb_blocks = len_liste(numbers);
    int* numbers_tab = list_to_tab(numbers);
    duo* blocks = (duo*)malloc(sizeof(duo)*nb_blocks);
    for (int i = 0;i<nb_blocks;i++){
        if (i==0){
            blocks[i].x = 0;
        }
        else{
            int tmp = 0;
            for (int j=0;j<i;j++){
                tmp += numbers_tab[j] +1;
            }
            blocks[i].x = tmp;
        }

        if (i==nb_blocks-1){
            blocks[i].y = n-1;
        }
        else {
            int tmp = 0;
            for (int j=i+1;j<nb_blocks;j++){
                tmp += numbers_tab[j] +1;
            }
            blocks[i].y = (n-1) - tmp;
        }

    }
    line_est_t* res = (line_est_t*)malloc(sizeof(line_est_t));
    res->nb_blocks = nb_blocks;
    res->est = blocks;
    free(numbers_tab);
    return res;
}

estimation_t* full_estimation(picross_numbers* nums){
    int n = nums->size;
    estimation_t* res = (estimation_t*)malloc(sizeof(estimation_t));
    line_est_t** lines = (line_est_t**)malloc(sizeof(line_est_t*)*n);
    line_est_t** cols = (line_est_t**)malloc(sizeof(line_est_t*)*n);
    for (int i =0;i<n;i++){
        line_est_t* line = estimate_line(nums->lig[i],n);
        line_est_t* col = estimate_line(nums->col[i],n);
        lines[i] = line;
        cols[i] = col;
    }
    res->n = n;
    res->cols = cols;
    res->lines = lines;

    return res;
}
```

```
int fill_with_0(int* line,int size){
    int res = 0;
    if (line[0] == 0 && line[size-1] == 0){
        return 0;
    }
    for (int i = 0;i<size;i++){
        if (line[i] != 0){
            res++;
            line[i] = 0;
        }
    }
    return res;
}

int rule1_1_line(int* line,int* nums,line_est_t* est){
    int res = 0;
    int nb_blocks = est->nb_blocks;
    for (int j = 0;j<nb_blocks;j++){
        int start_of_block = est->est[j].x;
        int end_of_block = est->est[j].y;
        int u = (end_of_block - start_of_block + 1) - nums[j];
        for (int i = start_of_block; i<end_of_block+1;i++){
            if ( i >= 0 && line[i] != 2){
                continue;
            }
            if (start_of_block + u <= i && end_of_block - u >= i){
                res++;
                line[i] = 1;
            }
        }
    }
    return res;
}

int rule1_2_line(int* line,int size,line_est_t* est){
    int res = 0;
    int nb_blocks = est->nb_blocks;
    for (int i = 0;i<size;i++){
        if (line[i] != 2){
            continue;
        }
        //printf("%d <",i);print_tab(line,size);printf("\n");
        if (i < est->est[0].x){
            //res++;
            line[i] = 0;
        }
        if (i > est->est[nb_blocks-1].y){
            res++;
            line[i] = 0;
        }
        for (int j = 0;j<nb_blocks -1;j++){
            if (est->est[j].y<i && i< est->est[j+1].x){
                res++;
                line[i] = 0;
            }
        }
    }
    return res;
}
```

```
int rule2_2line(int* line,int size,line_est_t* est){
    int res = 0;
    int nb_blocks = est->nb_blocks;
    for (int j =0;j<nb_blocks;j++){
        if (est->est[j].x-1 >=0 && line[est->est[j].x-1] ==1){
            res++;
            est->est[j].x++;
        }
        if ((est->est[j].y+1 < size )&& (line[est->est[j].y+1] == 1)){
            res++;
            est->est[j].y--;
        }
    }
    return res;
}


int rule1_3line(int* line,int size,int* nums,line_est_t* est){
    int res = 0;
    int nb_blocks = est->nb_blocks;
    for (int j = 0;j<nb_blocks;j++){
        //Start_blocks
        int first = est->est[j].x;
        bool any_one = false;
        bool all_one = true;
        if (line[first] == 1 && (first != 0 && line[first-1] == 2)){
            for (int i = 0;i<nb_blocks;i++){
                if (i == j){
                    continue;
                }
                if (est->est[i].x <= first && est->est[i].y >= first){
                    any_one = true;
                    if (nums[i] == 1){
                        all_one = all_one && true;
                    }
                    else{
                        all_one = false;
                    }
                }
            }
            if (any_one && all_one){
                line[first-1] = 0; //Since we need overlap this can never be the first block
so no error expected
                res++;
            }
        }
        int last = est->est[j].y;
        any_one = false;
        all_one = true;
        if (line[last] == 1 && (last != (size-1) && line[last+1] == 2)){
            for (int i = 0;i<nb_blocks;i++){
                if (i == j){
                    continue;
                }
                if (est->est[i].x <= last && est->est[i].y >= last){
                    any_one = true;
                    if (nums[i] == 1){
                        all_one = all_one && true;
                    }
                    else{
                        all_one = false;
                    }
                }
            }
            if (any_one && all_one){
                line[last+1] = 0; //Since we need overlap this can never be the last block so
no error expected
                res++;
            }}
    }
    return res;
}
```

Annexe

```
int apply_rules(picross_grid* grille_a_completer,picross_numbers* nums,estimation_t* est,int k){
    int res = 0;
    for (int passage = 0;passage<k;passage++){
        for (int i = 0;i<nums->size;i++){
            //Lignes
            int* line = grille_a_completer->grid[i];
            int* num_line = list_to_tab(nums->lig[i]);
            int size = nums->size;
            line_est_t* est_line = est->lines[i];
            if (est_line == NULL){
                //Cas ou la ligne est 0
                res += fill_with_0(line, size);
            }
            else{
                res += rule1_1_line(line, num_line, est_line);
                rule2_2line(line, size, est_line);
                res += rule1_2_line(line, size, est_line);
                res += rule1_3line(line, size, num_line, est_line);
            }
            //Cols
            int* col = get_col(grille_a_completer, size, i);
            int* num_col = list_to_tab(nums->col[i]);
            line_est_t* est_col = est->cols[i];
            if (est_col == NULL){
                res += fill_with_0(col, size);
            }
            else{
                res += rule1_1_line(col, num_col, est_col);
                rule2_2line(col, size, est_col);
                res += rule1_2_line(col, size, est_col);
                res += rule1_3line(col, size, num_col, est_col);
            }
            apply_tab_to_col(grille_a_completer, col, i);
            free(col);
            free(num_col);
            free(num_line);

        }
    }
    return res;
}


void print_estimation(line_est_t* e){
    if (e == NULL){
        printf("(NULL) ");
        return;
    }
    printf("NB_BLOCKS:%d",e->nb_blocks);
    for (int i = 0;i<e->nb_blocks;i++){
        printf("(%d,%d) ",e->est[i].x,e->est[i].y);
    }
    printf("\n");
}

void print_full_estimation(estimation_t* e){
    printf("LINES:\n");
    for (int i = 0;i < e->n;i++){
        print_estimation(e->lines[i]);
    }
    printf("COLS:\n");
    for (int i = 0;i < e->n;i++){
        print_estimation(e->cols[i]);
    }
    printf("\n");
}
```

```
void free_estimation(line_est_t* e){
    if (e == NULL){
        return;
    }
    free(e->est);
    free(e);
}
void free_full_estimation(estimation_t* e){
    for (int i = 0;i < e->n;i++){
        free_estimation(e->lines[i]);
    }
    for (int i = 0;i < e->n;i++){
        free_estimation(e->cols[i]);
    }
    free(e->cols);
    free(e->lines);
    free(e);
}
```

Annexe

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#include "automates.h"
#include "utils.h"
#include "picross.h"
#include "solver.h"
#include "valideurs.h"
#include "logicrules.h"

#define BRUTE Fl(1)
#define BACKTRACK Fl(2)
#define PRINTTIMEVALIDEUR Fl(3)
#define PRINTTIMEALGO Fl(4)
#define QUIET Fl(5)
#define PRINTSEED Fl(6)
#define DEBUG Fl(7)
#define PRINTMODELE Fl(8)
#define PRINTSOL Fl(9)
#define BACKTRACK_ND Fl(10)
#define LOGICRULES Fl(11)
#define GENGRILLE Fl(12)
#define RECORD Fl(13)

#define DEFAULT BACKTRACK|PRINTSEED
#define VERBOSE PRINTSEED | PRINTTIMEALGO | PRINTTIMEVALIDEUR | PRINTMODELE |
PRINTSOL;
#define ALL_TIME PRINTTIMEALGO | PRINTTIMEVALIDEUR

#define RECORD_TO_FILE(file,seed,time_valid,time_lr,time_algo,lr_completed,mode) \
fprintf(file,"%d %f %f %f %f %d %d\n",seed,time_valid,time_lr,time_algo,
(time_algo+time_lr+time_valid),lr_completed,mode)

int main(int argc,char** argv){
    int n = 3;
    int iter = 1;
    int seed = time(NULL);
    int options = 0;//Options par defaut
    char file_name[50] = "";
    FILE* record_file = NULL;
    int chance = 50;
    printf("Hello world!\n");

    for (int i=1;i<argc;i++){
        char* arg = argv[i];
        if (strcmp(arg,"--seed") == 0){
            assert(argc >= i+1);
            seed = atoi(argv[i+1]);
        }
        if (strcmp(arg,"-n") == 0){
            assert(argc >= i+1);
            n = atoi(argv[i+1]);
        }
        if (strcmp(arg, "--iter") ==0){
            assert(argc >= i+1);
            iter = atoi(argv[i+1]);
        }
        if (strcmp(arg, "--chance") ==0){
            assert(argc >= i+1);
            chance = atoi(argv[i+1]);
        }
        if (strcmp(arg, "-r") ==0){
            assert(argc >= i+1);
            options |= RECORD;
            strcpy(file_name, argv[i+1]);
         }
        if (strcmp(arg,"--backtrack")== 0){
            options |= BACKTRACK;
        }
        if (strcmp(arg,"--backtrack-nd")== 0){
            options |= BACKTRACK_ND;
        }
        if (strcmp(arg,"--brute") == 0){
            options |= BRUTE;
        }
        if (strcmp(arg,"--print-time-valideur") == 0){
            options |= PRINTTIMEVALIDEUR;
        }
        if (strcmp(arg,"--print-time-algo") == 0){
            options |= PRINTTIMEALGO;
        }
        if (strcmp(arg,"--print-seed") == 0){
            options |= PRINTSEED;
        }
        if (strcmp(arg,"-q") == 0){
            options |= QUIET;
        }
        if (strcmp(arg,"-d") == 0){
            options |= DEBUG;
        }
        if (strcmp(arg,"--print-model") == 0){
            options |= PRINTMODELE;
        }
        if (strcmp(arg,"--print-sol") == 0){
            options |= PRINTSOL;
        }
        if (strcmp(arg,"-v") == 0){
            options |= VERBOSE;
        }
        if (strcmp(arg,"-t") == 0){
            options |= ALL_TIME;
        }
        if (strcmp(arg,"--default") == 0){
            options |= DEFAULT;
        }
        if (strcmp(arg,"--lr") == 0){
            options |= LOGICRULES;
        }
        if (strcmp(arg,"-g") == 0){
            options |= GENGRILLE;
        }

        if (strcmp(arg, "--help") == 0){
            printf("Options: -n --seed --iter --chance --backtrack --brute --print-
time-valideur --print-time-algo --print-seed --quiet --debug --print-model --
print-sol\n ");
        }
    }
    printf("Debut du programme...\n");
    if (options & DEBUG){
        printf("No code in debug mode");
        return 0;
    }
    if (options & RECORD){
        record_file = fopen(file_name, "w");
    }
    if (options & PRINTSEED){
        printf("seed:%d\n",seed);
    }
    if (options & GENGRILLE){
        printf("Generation de la grille %d",seed);
        srand(seed);
        picross_grid* grille = gen_random_grid(n, chance);
        picross_grid* lr_grid = gen_unk_grid(n);
        picross_numbers* nums = gen_numbers_from_grid(grille);
        estimation_t* estimation = full_estimation(nums);
        int nb_lr = apply_rules(lr_grid, nums, estimation, 5);
        print_picc(grille);
        print_nums(nums);
        print_full_estimation(estimation);
        printf("LR solved:%d\n",nb_lr);
        print_picc(lr_grid);
        free_picross(lr_grid);
        free_picross(grille);
        free_numbers(nums);
        free_full_estimation(estimation);
        return 0;
    }
    for (int boucle = 1;boucle < iter +1;boucle++){
        clock_t t1_valid = 0;
        clock_t t2_valid = 0;
        double delta_valid = 0;
        clock_t t1_algo = 0;
        clock_t t2_algo = 0;
        double delta_algo = 0;
        clock_t t1_lr = 0;
        clock_t t2_lr = 0;
        double delta_lr = 0;
        int logic_rules_completed = -1;
        int mode = 0;

        srand(seed+boucle);
        picross_grid* grille_a_trouver = gen_random_grid(n,chance);
        picross_numbers* numeros = gen_numbers_from_grid(grille_a_trouver);
        if(options & PRINTMODELE){
            print_picc(grille_a_trouver);
            print_nums(numeros);
        }

        if (BRUTE & options){
            mode = 0;
            picross_grid* grille_vide = gen_empty_grid(n);
            t1_valid = clock();
            valideur_det* valideur_complet = gen_valideur_total(numeros);
            t2_valid = clock();

            t1_algo = clock();
            bool res = bruteforce(grille_vide,valideur_complet,0,0);
            t2_algo = clock();

            delta_valid = (double)(t2_valid - t1_valid) / CLOCKS_PER_SEC;
            delta_algo  = (double)(t2_algo - t1_algo) / CLOCKS_PER_SEC;

            if (options & PRINTSOL){
                print_picc(grille_vide);
            }
            //for (int i=0;i<3;i++){
            //    printf("%d\n",i);
            //    print_auto(valideur_complet->ligne[i]);
            //    print_auto(valideur_complet->col[i]);
            //
            //}
            free_picross(grille_vide);
            free_valideur_det(valideur_complet);


            //En cas d'erreur (ce qui est normalment impossible)
            if (!res){
                printf("Erreur de brute-force seed:%d\n",seed+boucle);
                free_picross(grille_a_trouver);
                free_numbers(numeros);
                return -1;
            }

            if (!(options & QUIET)){
```

Annexe

```
void add_connection_nd(automate_nd *A, int etat_d, int lettre, int etat_f){
    A->delta[etat_d][lettre] = add_to_liste(etat_f, A->delta[etat_d][lettre]);
}

        }
        printf("]");
    }
    printf("\n");
}

void free_auto_nd(automate_nd* A){
    for(int i=0;i<A->nb_etats;i++){
        for (int j=0;j<A->nb_lettres;j++){
            free_liste(A->delta[i][j]);
        }
        free(A->delta[i]);
    }
    free(A->delta);
    free(A->depart);
    free(A->finaux);
    free(A);
bool* delta_nd(automate_nd* A,bool* etats_depart,int lettre){
    bool* res = (bool*)calloc(A->nb_etats,sizeof(bool));
    for(int i=0;i<A->nb_etats;i++){
        if (etats_depart[i]){
            liste liste_to_parse = A->delta[i][lettre];
            while (liste_to_parse != NULL){
                res[liste_to_parse->val] = true;
                liste_to_parse = liste_to_parse->suivant;
            }
        }
    }
    return res;
}

bool* delta_etoile_nd(automate_nd* A,bool* etats_depart,int* input,int size_input){
    bool* etat_curr = (bool*)malloc(sizeof(bool)*A->nb_etats);
    memcpy(etat_curr,etats_depart, sizeof(bool)*A->nb_etats);
    for (int i=0;i<size_input;i++){
        bool* new_etat = delta_nd(A,etat_curr,input[i]);
        free(etat_curr);
        etat_curr = new_etat;
    }
    return etat_curr;
}

bool reconnu_afnd(automate_nd *A, int *input, int size_input){
    bool* res_tab = delta_etoile_nd(A,A->depart,input,size_input);
    bool res = false;
    for (int i=0;i<A->nb_etats;i++){
        res = res || (A->finaux[i] && res_tab[i]);
    }
    free(res_tab);
    return res;
}
```

```
dict* etats_atteints(automate_nd* A){
    int num_etat = 0;
    dict* res = create_dict(A->nb_etats * 5);
    liste a_voir = NULL;
    liste old_liste = NULL;
    a_voir = add_to_liste(binary_from_bool_int(A->depart,A->nb_etats),a_voir);
    while (a_voir != NULL){
        int elem_a_voir = a_voir->val;
        old_liste = a_voir;
        a_voir = old_liste->suivant;
        free(old_liste);
        if (!is_in_dico(res,elem_a_voir)){
            bool* etat_depart = bool_arr_from_int(elem_a_voir,A->nb_etats);
            for (int a = 0;a<A->nb_lettres;a++){
                bool* etat_fin = delta_nd(A,etat_depart,a);
                int etat_fin_int = binary_from_bool_int(etat_fin, A->nb_etats);
                a_voir = add_to_liste(etat_fin_int,a_voir);
                free(etat_fin);

            }
            add_dict(res,elem_a_voir,num_etat);
            num_etat++;
            free(etat_depart);
        }
    }
    free_liste(a_voir);
    return res;
}
```

```
automate_d* determiniser(automate_nd* A){
    dict* atteints = etats_atteints(A);
    liste keys = all_keys(atteints);
    int nb_etats = 0;
    liste keys_copy = keys;
    while (keys_copy != NULL){
        duo_liste dl = atteints->table[keys_copy->val];
        while (dl != NULL){
            nb_etats++;
            dl = dl->suivant;
        }
        keys_copy = keys_copy->suivant;
    }
    automate_d* res = init_automate(A->nb_lettres,nb_etats);
    keys_copy = keys;
    while (keys_copy != NULL){
        duo_liste dl = atteints->table[keys_copy->val];
        while (dl != NULL){
            int etat_depart_int = dl->val.x;
            bool* etat_depart = bool_arr_from_int(etat_depart_int,A->nb_etats);
            int nom_etat_depart = dl->val.y;
            for (int a=0;a<A->nb_lettres;a++){
                bool* etats_atteints = delta_nd(A,etat_depart, a);
                //if (etat_depart_int == 15){
                //    print_dico(atteints);
                //    printf("%d",nb_etats);
                //    print_liste(keys);
                //}
                int etats_atteints_int = binary_from_bool_int(etats_atteints,A->nb_etats);
                int nom_etat_atteints = find_dico(atteints, etats_atteints_int);
                add_connection_d(res, nom_etat_depart, a, nom_etat_atteints);
                free(etats_atteints);
            }
            if (and_bool_arr(etat_depart, A->finaux, A->nb_etats)){
                res->finaux[nom_etat_depart] = true;
            }
            if (equal_bool_arr(etat_depart, A->depart, A->nb_etats)){
                res->depart = nom_etat_depart;
            }
            free(etat_depart);
            dl = dl->suivant;
        }
        keys_copy = keys_copy->suivant;
    }
    free_dico(atteints);
    free_liste(keys);
    return res;


}

void print_auto_nd(automate_nd* A){
    printf("\n");
    printf("Nb delta: %d\n",A->nb_lettres);
    printf("Nb etats: %d\n",A->nb_etats);

    printf("Etat depart:");print_bool_tab(A->depart,A->nb_etats);printf("\n");
    printf("Etat finaux:");print_bool_tab(A->finaux,A->nb_etats);printf("\n");
    for (int i=0;i<A->nb_etats;i++){
        printf("[");
        for (int j=0;j<A->nb_lettres;j++){
            print_liste(A->delta[i][j]);
        }
        printf("]");
    }
    printf("\n");
}

void free_auto_nd(automate_nd* A){
    for(int i=0;i<A->nb_etats;i++){
        for (int j=0;j<A->nb_lettres;j++){
            free_liste(A->delta[i][j]);
        }
        free(A->delta[i]);
    }
    free(A->delta);
    free(A->depart);
    free(A->finaux);
    free(A);
```

Annexe

```
            printf("%d/%d resolu brute ",boucle,iter);
            if (options & PRINTTIMEVALIDEUR){
                printf("Temps valideur: %f sec ",delta_valid);
            }
            if (options & PRINTTIMEALGO){
                printf("Temps algo: %f sec ",delta_algo);
            }
            printf("\n");
        }
        if (options & RECORD){
            RECORD_TO_FILE(record_file, seed+boucle, delta_valid, delta_lr, delta_algo,
logic_rules_completed, mode);
        }

    }
    if (BACKTRACK & options){
        mode = 1;
        picross_grid* grille_inconnue = gen_unk_grid(n);
        t1_valid = clock();
        valideur_det* valideur_partiel = gen_valideur_partiel(numeros);
        t2_valid = clock();
        if (options & LOGICRULES){
            mode = 3;
            t1_lr = clock();
            estimation_t* estimation = full_estimation(numeros);
            logic_rules_completed = apply_rules(grille_inconnue, numeros, estimation, 5);
            t2_lr = clock();
            free_full_estimation(estimation);
            delta_lr = (double)(t2_lr - t1_lr) / CLOCKS_PER_SEC;
        }

        t1_algo = clock();
        bool res = backtracking(grille_inconnue,valideur_partiel,0,0);
        t2_algo = clock();

        delta_valid = (double)(t2_valid - t1_valid) / CLOCKS_PER_SEC;
        delta_algo  = (double)(t2_algo - t1_algo) / CLOCKS_PER_SEC;

        if (options & PRINTSOL){
            print_picc(grille_inconnue);
        }

        free_picross(grille_inconnue);
        free_valideur_det(valideur_partiel);


        //En cas d'erreur (ce qui est normalment impossible)
        if (!res){
            printf("Erreur de backtrack seed:%d\n",seed+boucle);
            free_picross(grille_a_trouver);
            free_numbers(numeros);
            return -1;
        }

        if (!(options & QUIET)){
            printf("%d/%d resolu backtrack ",boucle,iter);
            if (options & PRINTTIMEVALIDEUR){
                printf("Temps valideur: %f sec ",delta_valid);
                printf("Temps LR: %f sec ",delta_lr);
                printf("Nb fait:%d ",logic_rules_completed);
            }
            if (options & PRINTTIMEALGO){
                printf("Temps algo: %f sec ",delta_algo);
            }
            printf("\n");
        }
        if (options & RECORD){
            RECORD_TO_FILE(record_file, seed+boucle, delta_valid, delta_lr, delta_algo,
logic_rules_completed, mode);
        }
    }
    if (BACKTRACK_ND & options){
        mode = 2;
        picross_grid* grille_inconnue = gen_unk_grid(n);
        t1_valid = clock();
        valideur_ndet* valideur_partiel_ndet = gen_valideur_ndet(numeros);
        t2_valid = clock();

        if (options & LOGICRULES){
            mode = 4;
            t1_lr = clock();
            estimation_t* estimation = full_estimation(numeros);
            logic_rules_completed = apply_rules(grille_inconnue, numeros,
estimation, 5);
            t2_lr = clock();
            free_full_estimation(estimation);
            delta_lr = (double)(t2_lr - t1_lr) / CLOCKS_PER_SEC;
        }

        t1_algo = clock();
        bool res =
backtracking_ndet(grille_inconnue,valideur_partiel_ndet,0,0);
        t2_algo = clock();

        delta_valid = (double)(t2_valid - t1_valid) / CLOCKS_PER_SEC;
        delta_algo  = (double)(t2_algo - t1_algo) / CLOCKS_PER_SEC;

        if (options & PRINTSOL){
            print_picc(grille_inconnue);
        }
        free_picross(grille_inconnue);
        free_valideur_ndet(valideur_partiel_ndet);


        //En cas d'erreur (ce qui est normalment impossible)
        if (!res){
            printf("Erreur de backtrack seed:%d\n",seed+boucle);
            free_picross(grille_a_trouver);
            free_numbers(numeros);
            return -1;
        }

        if (!(options & QUIET)){
            printf("%d/%d resolu backtrack-nd ",boucle,iter);
            if (options & PRINTTIMEVALIDEUR){
                printf("Temps valideur: %f sec ",delta_valid);
                printf("Temps LR: %f sec",delta_lr);
                printf("Nb fait: %d ",logic_rules_completed);
            }
            if (options & PRINTTIMEALGO){
                printf("Temps algo: %f sec ",delta_algo);
            }
            printf("\n");
        }
        if (options & RECORD){
            RECORD_TO_FILE(record_file, seed+boucle, delta_valid, delta_lr,
delta_algo, logic_rules_completed, mode);
        }

    }
    free_picross(grille_a_trouver);
    free_numbers(numeros);

  }
  if (options & RECORD){
    fclose(record_file);
  }

  printf("Fin programme\n");
  return 0;
}
```

Annexe

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "picross.h"
#include "utils.h"
#include "automates.h"
#include "listes.h"

picross_grid* gen_empty_grid(int size){
    int** res_grid = (int**)malloc(sizeof(int*)*size);
    for (int i=0;i<size;i++){
        int* res_grid_bis = (int*)calloc(size,sizeof(int));
        res_grid[i] = res_grid_bis;
    }
    picross_grid* res = (picross_grid*)malloc(sizeof(picross_grid));
    res->size = size;
    res->grid = res_grid;
    return res;
}

picross_grid* gen_unk_grid(int size){
    int** res_grid = (int**)malloc(sizeof(int*)*size);
    for (int i=0;i<size;i++){
        int* res_grid_bis = (int*)malloc(size*sizeof(int));
        for (int j =0;j<size;j++){
            res_grid_bis[j] = 2;
        }
        res_grid[i] = res_grid_bis;
    }
    picross_grid* res = (picross_grid*)malloc(sizeof(picross_grid));
    res->size = size;
    res->grid = res_grid;
    return res;
}

picross_grid* gen_random_grid(int size, int chance){
    int** res = (int**)malloc(sizeof(int*)*size);
    for (int i = 0;i<size;i++){
        int* tab_bis = (int*)malloc(sizeof(int)*size);
        for (int j=0;j<size;j++){
            if ((rand() % 100) > chance){
                tab_bis[j] = 0;
            }else{
                tab_bis[j] = 1;
            }
        }
        res[i] = tab_bis;
    }
    picross_grid* res_piccross = (picross_grid*)malloc(sizeof(picross_grid));
    res_piccross->size = size;
    res_piccross->grid = res;
    return res_piccross;
}

picross_numbers* gen_numbers_from_grid(picross_grid* grid){
    int size = grid->size;
    //Lines
    liste* res_ligne = (liste*)malloc(sizeof(liste)*size);
    liste* res_cols = (liste*)malloc(sizeof(liste)*size);
    for (int i =0;i<size;i++){
        liste res_ligne_bis = NULL;
        liste res_cols_bis = NULL;
        int size_chunk_ligne = 0;
        int size_chunk_col =0;
        for (int j=size-1;j>-1;j--){
            //Ligne
            if (grid->grid[i][j] == 1){
                size_chunk_ligne++;
            }
            else{
                if (size_chunk_ligne != 0){
                    res_ligne_bis =
add_to_liste(size_chunk_ligne,res_ligne_bis);
                    size_chunk_ligne = 0;
                }
            }
            //Colonnes
            if (grid->grid[j][i] == 1){
                size_chunk_col++;
            }
            else{
                if (size_chunk_col != 0){
                    res_cols_bis = add_to_liste(size_chunk_col,res_cols_bis);
                    size_chunk_col = 0;
                }
            }

        }
        if (size_chunk_ligne != 0){
            res_ligne_bis = add_to_liste(size_chunk_ligne,res_ligne_bis);
            size_chunk_ligne = 0;
        }
        if (size_chunk_col != 0){
            res_cols_bis = add_to_liste(size_chunk_col,res_cols_bis);
            size_chunk_col = 0;
        }
        res_ligne[i] = res_ligne_bis;
        res_cols[i] = res_cols_bis;

    }
    picross_numbers* res =
(picross_numbers*)malloc(sizeof(picross_numbers));
    res->size = size;
    res->lig = res_ligne;
    res->col = res_cols;
    return res;

}

picross_grid* tourner_grille(picross_grid* grid){
    int** res_grid = (int**)malloc(sizeof(int*)*grid->size);
    for (int i=0;i<grid->size;i++){
        int* res_grid_bis = (int*)malloc(sizeof(int)*grid->size);
        for (int j=0;j<grid->size;j++){
            res_grid_bis[j] = grid->grid[j][i];
        }
        res_grid[i] = res_grid_bis;
    }
    picross_grid* res = (picross_grid*)malloc(sizeof(picross_grid));
    res->size = grid->size;
    res->grid = res_grid;
    return res;

}

int* get_col(picross_grid* g, int size,int col_n){
    int* res = (int*)malloc(sizeof(int)*size);
    for (int i = 0;i<size;i++){
        res[i] = g->grid[i][col_n];
    }
    return res;
}

void print_picc(picross_grid* p){
    printf("\n\n");
    printf("--------------\n");
    for(int i=0;i<p->size;i++){
        for (int j=0;j<p->size;j++){
            if (p->grid[i][j] == 1){
                printf("#");
            }else if (p->grid[i][j] == 0){
                printf(" ");
            }else {
                printf("?");
            }
        }
        printf("\n");
    }
    printf("--------------\n");
}

void print_nums(picross_numbers* nums){
    printf("\n");
    printf("Ligne:[");
    for (int i=0;i<nums->size;i++){
        print_liste(nums->lig[i]);printf(" ");
    }
    printf("]\nCols:[");
    for (int i=0;i<nums->size;i++){
        print_liste(nums->col[i]);printf(" ");
    }
    printf("]\n");
}



/*bool cmp_grid(picross_grid* g1,picross_grid* g2){
    bool res = true;
}*/

void apply_tab_to_col(picross_grid* g,int* col,int col_n){
    for (int i = 0;i<g->size;i++){
        g->grid[i][col_n] = col[i];
    }
}

void free_picross(picross_grid* p){
    free_int_int(p->grid,p->size);
    free(p);
}

void free_numbers(picross_numbers* nums){
    for(int i=0;i<nums->size;i++){
        free_liste(nums->lig[i]);
        free_liste(nums->col[i]);

    }
    free(nums->lig);
    free(nums->col);
    free(nums);
}
```

Annexe

```
#include <assert.h>
#include <ctype.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>

#include "automates.h"
#include "picross.h"
#include "utils.h"
#include "valideurs.h"

bool est_solution_valide_total(picross_grid* grid,valideur_det* valideur){
    bool res = true;
    picross_grid* grid_turned = tourner_grille(grid);
    for (int i=0;i<grid->size;i++){
        res = res && reconnu_afd(valideur->ligne[i], grid->grid[i],grid->size);
        res = res && reconnu_afd(valideur->col[i], grid_turned->grid[i],grid->size);
    }
    free_picross(grid_turned);
    return res;
}

bool verif_ligne_col(picross_grid* grid,valideur_det* valideur,int ligne,int col){
    int* col_arr = get_col(grid, grid->size, col);
    bool res =
        reconnu_afd(valideur->ligne[ligne],grid->grid[ligne],grid->size)
        &&
        reconnu_afd(valideur->col[col], col_arr, grid->size);
    free(col_arr);
    return res;


}

bool verif_ligne_col_ndet(picross_grid* grid,valideur_ndet* valideur,int ligne,int col){
    int* col_arr = (int*)malloc(sizeof(int)*grid->size);
    for (int i = 0; i < grid->size; i++){
        col_arr[i] = grid->grid[i][col];
    }
    bool res =
        reconnu_afnd(valideur->ligne[ligne],grid->grid[ligne],grid->size)
        &&
        reconnu_afnd(valideur->col[col], col_arr, grid->size);
    free(col_arr);
    return res;


}

bool bruteforce(picross_grid* grid,valideur_det* valideur,int i,int j){
    if (i == grid->size-1 && j==grid->size-1){
        //print_picc(grid);
        if (est_solution_valide_total(grid,valideur)){
            return true;
        }
        grid->grid[i][j] = 1;
        if (est_solution_valide_total(grid,valideur)){
            return true;
        }
        grid->grid[i][j] = 0;
        return false;
    }
    int j_next = (j+1) % grid->size;
    int i_next;
    if (j == grid->size -1){
        i_next = i+1;
    }
    else{
        i_next = i;
    }

    if (bruteforce(grid,valideur, i_next, j_next)){
        return true;
    }
    grid->grid[i][j] = 1;
    if (bruteforce(grid,valideur, i_next, j_next)){
        return true;
    }
    grid->grid[i][j] = 0;
    return false;
}


bool backtracking(picross_grid* grid, valideur_det* valideur, int i, int j){
    if (i == grid->size -1 && j == grid->size-1){
        if (grid->grid[i][j] != 2){
            return true;
        }
        grid->grid[i][j] = 0;
        if (verif_ligne_col(grid,valideur,i,j)){
            return true;
        }
        grid->grid[i][j] = 1;
        if (verif_ligne_col(grid,valideur,i,j)){
            return true;
        }
        grid->grid[i][j] = 2;
        return false;
    }
    int j_next = (j+1) % grid->size;
    int i_next;
    if (j == grid->size -1){
        i_next = i+1;
    }
    else{
        i_next = i;
    }
    if (grid->grid[i][j] != 2){
        return backtracking(grid,valideur,i_next,j_next);
    }
    grid->grid[i][j] = 0;
    if (verif_ligne_col(grid,valideur,i,j)){
        if (backtracking(grid,valideur,i_next,j_next)){
            return true;
        }
    }
    grid->grid[i][j] = 1;
    if (verif_ligne_col(grid,valideur,i,j)){
        if (backtracking(grid,valideur,i_next,j_next)){
            return true;
        }
    }
    grid->grid[i][j] = 2;
    return false;

}

bool backtracking_ndet(picross_grid* grid, valideur_ndet* valideur, int i,
int j){
    if (i == grid->size -1 && j == grid->size-1){
        if (grid->grid[i][j] != 2){
            return true;
        }
        grid->grid[i][j] = 0;
        if (verif_ligne_col_ndet(grid,valideur,i,j)){
            return true;
        }
        grid->grid[i][j] = 1;
        if (verif_ligne_col_ndet(grid,valideur,i,j)){
            return true;
        }
        grid->grid[i][j] = 2;
        return false;
    }
    int j_next = (j+1) % grid->size;
    int i_next;
    if (j == grid->size -1){
        i_next = i+1;
    }
    else{
        i_next = i;
    }
    //assert(i_next < grid->size);
    if (grid->grid[i][j] != 2){
        return backtracking_ndet(grid,valideur,i_next,j_next);
    }
    grid->grid[i][j] = 0;
    if (verif_ligne_col_ndet(grid,valideur,i,j)){
        if (backtracking_ndet(grid,valideur,i_next,j_next)){
            return true;
        }
    }
    //printf("Zero raté:%d %d\n",i,j);
    grid->grid[i][j] = 1;
    if (verif_ligne_col_ndet(grid,valideur,i,j)){
        if (backtracking_ndet(grid,valideur,i_next,j_next)){
            return true;
        }
    }
    //printf("Tout raté:%d %d\n",i,j);
    grid->grid[i][j] = 2;
    return false;

}
```

Annexe

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>



void free_int_int(int** t,int n){
    for(int i =0;i<n;i++){
        free(t[i]);
    }
    free(t);
}



void print_tab(int* t, int size){
    printf("[");
    for (int i = 0;i<size;i++){
        printf("%d ",t[i]);
    }
    printf("]");
}

void print_bool_tab(bool* t, int size){
    printf("[");
    for (int i = 0;i<size;i++){
        if (t[i]){
            printf("true ");
        }else{
            printf("false ");
        }
    }
    printf("]");

}


int binary_from_bool_int(bool* t,int size){
    int res =0;
    for (int i=0;i<size;i++){
        if (t[i]){
            res += pow(2,i);
        }
    }
    return res;
}

bool* bool_arr_from_int(int num,int size){
    bool* res = calloc(size, sizeof(bool));
    int num_now = num;
    for (int i = size-1;i>-1;i--){
        if ((num_now>=pow(2,i))){
            res[i] = true;
            num_now -= pow(2,i);
        }
    }
    return res;
}

int inverse_valeur(int i){
    if (i==0){
        return 1;
    }else {
    return 0;
    }
}

bool and_bool_arr(bool* a1,bool* a2,int size){
    for (int i=0;i<size;i++){
        if (a1[i] && a2[i]){
            return true;
        }
    }
    return false;
}

bool equal_bool_arr(bool* a1,bool* a2,int size){
    bool res = true;
    for (int i=0;i<size;i++){
        res = res && ((a1[i] && a2[i]) || (!a1[i] && !a2[i]));
    }
    return res;
}

void copy_tab(int* dest,int* src,int size){
    for (int i  = 0;i<size;i++){
        dest[i] = src[i];
    }
}
```

Annexe

```
#include <stdbool.h>
#include <stdlib.h>

#include "automates.h"
#include "picross.h"
#include "valideurs.h"
#include "listes.h"
#include "utils.h"

automate_d* auto_de_zeros(void){
    automate_d* res = init_automate(2,1);
    res->depart = 0;
    res->finaux[0] = true;
    add_connection_d(res,0,0,0);
    return res;
}

automate_nd* auto_nd_zeros(void){
    automate_nd* res = init_automate_nd(3,1);
    res->depart[0] = true;
    res->finaux[0] = true;
    add_connection_nd(res,0,0,0);
    add_connection_nd(res, 0, 2, 0);
    return res;
}

automate_d* generer_automate_ligne(liste ligne){
    if (ligne == NULL){
        return auto_de_zeros();
    }
    int nb_of_states = 0; //L'etat vide
    liste ligne_to_parse = ligne;
    while (ligne_to_parse != NULL){
        nb_of_states += 1 + ligne_to_parse->val;
        ligne_to_parse = ligne_to_parse->suivant;
    }
    automate_d* res = init_automate(2, nb_of_states);

    ligne_to_parse = ligne;
    int state_index =0;
    while (ligne_to_parse != NULL){
        //Connect to itself
        add_connection_d(res, state_index,0,state_index);
        //Fait une chaine de 1
        for (int j = 0;j<ligne_to_parse->val;j++){
            add_connection_d(res, state_index,1,state_index +1);
            state_index++;
        }
        //On DOIT finir par un zero si on est pas le dernier nombre
        if (ligne_to_parse->suivant != NULL){
            add_connection_d(res, state_index, 0, state_index+1);
            state_index++;
        }
        //Sinon on boucle en 0 sur le dernier
        else{
            add_connection_d(res,state_index,0,state_index);
            state_index++;
        }
        ligne_to_parse = ligne_to_parse->suivant;
    }
    res->depart = 0;
    res->finaux[res->nb_etats -1] = true;

    return res;

}

valideur_det* gen_valideur_total(picross_numbers* nums){
    automate_d** ligne = (automate_d**)malloc(sizeof(automate_d*)*nums->size);
    automate_d** cols = (automate_d**)malloc(sizeof(automate_d*)*nums->size);
    for(int i=0;i<nums->size;i++){
        ligne[i] = generer_automate_ligne(nums->lig[i]);
        cols[i] = generer_automate_ligne(nums->col[i]);
    }
    valideur_det* res = (valideur_det*)malloc(sizeof(valideur_det));
    res->size = nums->size;
    res->ligne = ligne;
    res->col = cols;
    return res;
}


automate_nd* generer_automate_partiel_ligne(liste ligne){
    if (ligne == NULL){
        return auto_nd_zeros();
    }
    int nb_of_states = 0; //L'etat vide
    liste ligne_to_parse = ligne;
    while (ligne_to_parse != NULL){
        nb_of_states += 1 + ligne_to_parse->val;
        ligne_to_parse = ligne_to_parse->suivant;
    }
    automate_nd* res = init_automate_nd(3, nb_of_states);
    //print_auto_nd(res);

    ligne_to_parse = ligne;
    int state_index =0;
    while (ligne_to_parse != NULL){
        //Connect to itself
        add_connection_nd(res, state_index,0,state_index);
        add_connection_nd(res, state_index,2,state_index);
        //Fait une chaine de 1
        for (int j = 0;j<ligne_to_parse->val;j++){
            add_connection_nd(res, state_index,1,state_index +1);
            add_connection_nd(res, state_index,2,state_index +1);
            state_index++;
        }
        //On DOIT finir par un zero si on est pas le dernier nombre
        if (ligne_to_parse->suivant != NULL){
            add_connection_nd(res, state_index, 0, state_index+1);
            add_connection_nd(res, state_index, 2, state_index+1);
            state_index++;
        }
        //Sinon on boucle en 0 sur le dernier
        else{
            add_connection_nd(res,state_index,0,state_index);
            add_connection_nd(res,state_index,2,state_index);
            state_index++;
        }
        ligne_to_parse = ligne_to_parse->suivant;
    }
    res->depart[0] = true;
    res->finaux[res->nb_etats -1] = true;

    return res;

}

valideur_ndet* gen_valideur_ndet(picross_numbers* nums){
    automate_nd** ligne =
(automate_nd**)malloc(sizeof(automate_nd*)*nums->size);
    automate_nd** cols =
(automate_nd**)malloc(sizeof(automate_nd*)*nums->size);
    for(int i=0;i<nums->size;i++){
        ligne[i] = generer_automate_partiel_ligne(nums->lig[i]);
        cols[i] = generer_automate_partiel_ligne(nums->col[i]);
    }
    valideur_ndet* res = (valideur_ndet*)malloc(sizeof(valideur_ndet));
    res->size = nums->size;
    res->ligne = ligne;
    res->col = cols;
    return res;
}

valideur_det* gen_valideur_partiel(picross_numbers* nums){
    automate_d** ligne = (automate_d**)malloc(sizeof(automate_d*)*nums->size);
    automate_d** cols = (automate_d**)malloc(sizeof(automate_d*)*nums->size);
    for(int i=0;i<nums->size;i++){
        automate_nd* tmp_ligne = generer_automate_partiel_ligne(nums->lig[i]);
        automate_nd* tmp_col = generer_automate_partiel_ligne(nums->col[i]);
        ligne[i] = determiniser(tmp_ligne);
        cols[i] = determiniser(tmp_col);
        free_auto_nd(tmp_ligne);
        free_auto_nd(tmp_col);
    }
    valideur_det* res = (valideur_det*)malloc(sizeof(valideur_det));
    res->size = nums->size;
    res->ligne = ligne;
    res->col = cols;
    return res;
}

void free_valideur_det(valideur_det* A){
    for (int i = 0;i<A->size;i++){
        free_auto(A->ligne[i]);
        free_auto(A->col[i]);
    }
    free(A->ligne);
    free(A->col);
    free(A);
}

void free_valideur_ndet(valideur_ndet *A){
    for (int i = 0;i<A->size;i++){
        free_auto_nd(A->ligne[i]);
        free_auto_nd(A->col[i]);
    }
    free(A->ligne);
    free(A->col);
    free(A);
}
```

Annexe

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#include "automates.h"
#include "utils.h"
#include "picross.h"
#include "solver.h"
#include "valideurs.h"
#include "logicrules.h"

#define BRUTE Fl(1)
#define BACKTRACK Fl(2)
#define PRINTTIMEVALIDEUR Fl(3)
#define PRINTTIMEALGO Fl(4)
#define QUIET Fl(5)
#define PRINTSEED Fl(6)
#define DEBUG Fl(7)
#define PRINTMODELE Fl(8)
#define PRINTSOL Fl(9)
#define BACKTRACK_ND Fl(10)
#define LOGICRULES Fl(11)
#define GENGRILLE Fl(12)
#define RECORD Fl(13)

#define DEFAULT BACKTRACK|PRINTSEED
#define VERBOSE PRINTSEED | PRINTTIMEALGO | PRINTTIMEVALIDEUR | PRINTMODELE |
PRINTSOL;
#define ALL_TIME PRINTTIMEALGO | PRINTTIMEVALIDEUR

#define RECORD_TO_FILE(file,seed,time_valid,time_lr,time_algo,lr_completed,mode) \
fprintf(file,"%d %f %f %f %f %d %d\n",seed,time_valid,time_lr,time_algo,
(time_algo+time_lr+time_valid),lr_completed,mode)

int main(int argc,char** argv){
    int n = 3;
    int iter = 1;
    int seed = time(NULL);
    int options = 0;//Options par defaut
    char file_name[50] = "";
    FILE* record_file = NULL;
    int chance = 50;
    printf("Hello world!\n");

    for (int i=1;i<argc;i++){
        char* arg = argv[i];
        if (strcmp(arg,"--seed") == 0){
            assert(argc >= i+1);
            seed = atoi(argv[i+1]);
        }
        if (strcmp(arg,"-n") == 0){
            assert(argc >= i+1);
            n = atoi(argv[i+1]);
        }
        if (strcmp(arg, "--iter") ==0){
            assert(argc >= i+1);
            iter = atoi(argv[i+1]);
        }
        if (strcmp(arg, "--chance") ==0){
            assert(argc >= i+1);
            chance = atoi(argv[i+1]);
        }
        if (strcmp(arg, "-r") ==0){
            assert(argc >= i+1);
            options |= RECORD;
            strcpy(file_name, argv[i+1]);
         }
        if (strcmp(arg,"--backtrack")== 0){
            options |= BACKTRACK;
        }
        if (strcmp(arg,"--backtrack-nd")== 0){
            options |= BACKTRACK_ND;
        }
        if (strcmp(arg,"--brute") == 0){
            options |= BRUTE;
        }
        if (strcmp(arg,"--print-time-valideur") == 0){
            options |= PRINTTIMEVALIDEUR;
        }
        if (strcmp(arg,"--print-time-algo") == 0){
            options |= PRINTTIMEALGO;
        }
        if (strcmp(arg,"--print-seed") == 0){
            options |= PRINTSEED;
        }
        if (strcmp(arg,"-q") == 0){
            options |= QUIET;
        }
        if (strcmp(arg,"-d") == 0){
            options |= DEBUG;
        }
        if (strcmp(arg,"--print-model") == 0){
            options |= PRINTMODELE;
        }
        if (strcmp(arg,"--print-sol") == 0){
            options |= PRINTSOL;
        }
        if (strcmp(arg,"-v") == 0){
            options |= VERBOSE;
        }
        if (strcmp(arg,"-t") == 0){
            options |= ALL_TIME;
        }
        if (strcmp(arg,"--default") == 0){
            options |= DEFAULT;
        }
        if (strcmp(arg,"--lr") == 0){
            options |= LOGICRULES;
        }
        if (strcmp(arg,"-g") == 0){
            options |= GENGRILLE;
        }

        if (strcmp(arg, "--help") == 0){
            printf("Options: -n --seed --iter --chance --backtrack --brute --print-
time-valideur --print-time-algo --print-seed --quiet --debug --print-model --
print-sol\n ");
        }
    }
    printf("Debut du programme...\n");
    if (options & DEBUG){
        printf("No code in debug mode");
        return 0;
    }
    if (options & RECORD){
        record_file = fopen(file_name, "w");
    }
    if (options & PRINTSEED){
        printf("seed:%d\n",seed);
    }
    if (options & GENGRILLE){
        printf("Generation de la grille %d",seed);
        srand(seed);
        picross_grid* grille = gen_random_grid(n, chance);
        picross_grid* lr_grid = gen_unk_grid(n);
        picross_numbers* nums = gen_numbers_from_grid(grille);
        estimation_t* estimation = full_estimation(nums);
        int nb_lr = apply_rules(lr_grid, nums, estimation, 5);
        print_picc(grille);
        print_nums(nums);
        print_full_estimation(estimation);
        printf("LR solved:%d\n",nb_lr);
        print_picc(lr_grid);
        free_picross(lr_grid);
        free_picross(grille);
        free_numbers(nums);
        free_full_estimation(estimation);
        return 0;
    }
    for (int boucle = 1;boucle < iter +1;boucle++){
        clock_t t1_valid = 0;
        clock_t t2_valid = 0;
        double delta_valid = 0;
        clock_t t1_algo = 0;
        clock_t t2_algo = 0;
        double delta_algo = 0;
        clock_t t1_lr = 0;
        clock_t t2_lr = 0;
        double delta_lr = 0;
        int logic_rules_completed = -1;
        int mode = 0;

        srand(seed+boucle);
        picross_grid* grille_a_trouver = gen_random_grid(n,chance);
        picross_numbers* numeros = gen_numbers_from_grid(grille_a_trouver);
        if(options & PRINTMODELE){
            print_picc(grille_a_trouver);
            print_nums(numeros);
        }

        if (BRUTE & options){
            mode = 0;
            picross_grid* grille_vide = gen_empty_grid(n);
            t1_valid = clock();
            valideur_det* valideur_complet = gen_valideur_total(numeros);
            t2_valid = clock();

            t1_algo = clock();
            bool res = bruteforce(grille_vide,valideur_complet,0,0);
            t2_algo = clock();

            delta_valid = (double)(t2_valid - t1_valid) / CLOCKS_PER_SEC;
            delta_algo  = (double)(t2_algo - t1_algo) / CLOCKS_PER_SEC;

            if (options & PRINTSOL){
                print_picc(grille_vide);
            }
            //for (int i=0;i<3;i++){
            //    printf("%d\n",i);
            //    print_auto(valideur_complet->ligne[i]);
            //    print_auto(valideur_complet->col[i]);
            //
            //}
            free_picross(grille_vide);
            free_valideur_det(valideur_complet);


            //En cas d'erreur (ce qui est normalment impossible)
            if (!res){
                printf("Erreur de brute-force seed:%d\n",seed+boucle);
                free_picross(grille_a_trouver);
                free_numbers(numeros);
                return -1;
            }

            if (!(options & QUIET)){
```