1ère année Informatique - Semestre 5

Rapport

Projet de programmation en langage C

Équipe n°36
Jérémy Bezamat
Camille Plages

Responsable Projet : M. Renault
Encadrant : M. Neggaz

6 janvier 2017

Introduction

Élèves ingénieurs au sein de l'Enseirb-Matmeca en première année d'informatique, nous avons effectué ce semestre le premier projet de notre formation. Ce travail effectué en binôme nous a permis de mettre en pratique nos connaissances théoriques sur le langage de programmation C.

Le but de ce projet est de reproduire le jeu de sociéte "Pelican Cove" (ou "Uluru", les deux jeux étant très similaires et ayant le même principe).

Nous présenterons et expliciterons ce principe dans une première partie. Par la suite nous expliquerons notre démarche en détaillant les structures mises en place afin de pouvoir programmer ce jeu. Nous poursuivrons en expliquant les différentes étapes de notre cheminement et les parties de notre code. Enfin, sera décrit la série de tests mis en place vérifiant la validité de notre code.

Table des matières

1 Présentation du sujet

1.1 Principe de jeu

Le jeu de société "Pelican Cove" contient un plateau avec un certain nombre de cases. Au début du jeu ces cases sont vides. Le jeu contient également plusieurs pions (des pélicans comme l'indique le nom du jeu) de différentes couleurs. Enfin le jeu possède des cartes, décrivant une contrainte pour le pélican auquel elles seront associées : ces cartes seront distribuées aux différents pions. Ainsi chaque pélican possède une contrainte, décrivant la manière dont il souhaite être disposé sur le plateau.

Ces contraintes sont variées et peuvent dépendre seulement du plateau, par exemple la contrainte *Au nord* correspond à quelques cases du plateau définies au préalable. Elles peuvent également dépendre d'un autre pélican, par exemple un pélican peut vouloir *Être en face* d'un autre pélican ou *Être du même côté*.

Avec une contrainte imposée à chaque pélican, le but du jeu sera donc de satisfaire l'ensemble de ces contraintes induisant placement des pélicans sur le plateau (un seul pélican par case). On s'apercevra assez rapidement que satisfaire tout le monde n'est pas toujours possible, dans ce cas il faudra essayer de satisfaire un maximum de pélicans.

FIGURE 1 – Plateau du jeu "Pelican Cove"

Ci-dessus un exemple de placement satisfaisant l'ensemble des contraintes suivantes :

- Le pélican rouge veut être au sud
- Le pélican blanc veut être à l'ouest
- Le pélican vert veut être du même côté que le pélican blanc
- Le pélican jaune veut être au nord
- Le pélican noir veut être en face du pélican rouge
- Le pélican violet veut être à l'est
- Le pélican orange veut être en face du pélican vert
- Le pélican bleu veut être à côté du pélican orange

1.2 Programmation du jeu

Pour récréer un tel jeu, il est nécessaire de définir le problème et expliciter ce dont nous aurons besoin pour le résoudre. Il pourrait se résumer, de façon succincte, à : "Créer un plateau de jeu et placer sur celui ci les différents pélicans en fonction de leurs contraintes."

Pour ce faire, nous aurons besoin de définir un certain nombre d'éléments : le nombre de positions sur le plateau, le nombre de pélicans à placer, les différentes contraintes possibles, etc...

Pour les contraintes dépendantes seulement du plateau il faudra préciser quelle position satisfait quelle contrainte. Des étiquettes (ou tags) seront alors attribuées à chaque position. Par exemple si la position numéro 1 possède l'étiquette "Nord", pour qu'un pélican puisse satisfaire la contrainte *Être au nord* il devra se trouver sur la position numéro 1 (ou sur toutes autres positions possédant l'étiquette "Nord").

Quand aux contraintes dépendantes des pélicans, il faudra introduire une notion de distance : si un pélican veut *Être à côté* d'un autre pélican, il faudra que la distance séparant ces deux pélicans montre qu'ils sont voisins.

Pour représenter les pélicans nous utiliserons des entiers, représentant les couleurs, nous permettant ainsi de gérer le nombre de pélicans souhaité.

Toutes ces structures et fonctions de base nécessaires à la reproduction du jeu seront décrites dans la prochaine partie.

Vous verrez dans cette partie, que toutes nos structures et fonctions sont triées et séparées dans plusieurs fichiers. En vérité nous n'avons effectué cette organisation qu'au bout de quelques séances, car nous avons voulu dans un premier temps poser les bases ensemble. Après réflexion et écriture des premières structures et fonctions associées, nous nous sommes rapidement rendu compte de l'utilité de séparer notre code en plusieurs fichiers. L'intérêt était double : nous permettre de travailler en même temps sur des fichiers séparés, mais aussi rendre notre code plus logique et lisible. Notre travail a alors consisté à réorganiser l'ensemble de notre code et créer un Makefile permettant de l'exécuter.

2 Création des structures de base

Nous avons dû dans un premier temps créer les fondations du jeu Pelican Cove d'après les règles fournies. Le début de l'implémentation en C consistait donc à créer les structures et les fonctions de bases du jeu. Dans cette section nous verrons dans un premier temps comment nous avons implémenté des plateaux de jeux, puis nous verrons les fonctions caractéristiques du plateau et enfin comment se déroule l'affectation des pélicans.

2.1 Implémentation des plateaux de jeu

Nous avons tout d'abord implémenté des plateaux de jeu et pour cela il a fallu definir des "Tags" permettant de décrire chaque position :

```
#define TAG_NORTH 1
#define TAG_SOUTH 2
#define TAG_WEST 3
#define TAG_EAST 4
#define TAG_CORNER 5
#define TAG_CERCLE_1 6
#define TAG_CERCLE_2 7
#define TAG_CERCLE_3 8
```

FIGURE 2 – Definition des *Tags* (define.h)

Puis nous avons créé une structure plateau :

```
struct board {
  int num_positions;
  int num_neighbours;
  int id; //Id of board
  int positions[MAX_POS][MAX_TAG];
  int neighbours[MAX_POS][MAX_NEIGHBOURS];
};
```

FIGURE 3 – Code de la structure *board* (board.h)

Cette structure comporte 5 champs. Tout d'abord *num_positions* qui indique le nombre total de positions du plateau, ensuite *num_neighbours* qui indique le nombre de voisins par position. L'entier *id* permet quant à lui de différencier les plateaux.
Le 4eme champ est un tableau de deux dimensions d'entiers nommé *positions* qui permet d'attribuer des tags à chaque position. Le dernier champ est également un tableau d'entiers de deux dimensions nommé *neighbours* qui indique les voisins de chaque position. Nous allons voir les différents plateaux que nous avons créés.

Le premier plateau que nous avons créé comporte seulement 4 cases :

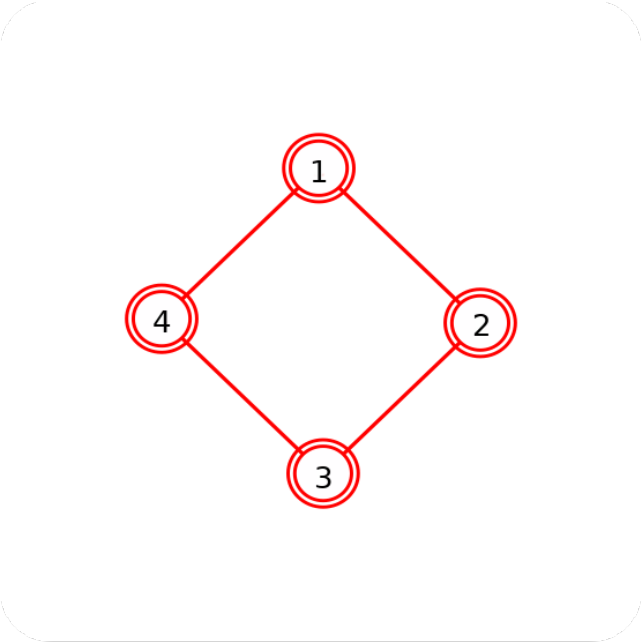

FIGURE 4 – Board 1

Pour jouer avec ce plateau il suffit de l'initialiser grâce à la fonction *init_board_1* :

```
struct board init_board_1(){
  struct board b={.num_positions=4,\
                  .num_neighbours=2,\
                  .id=1,\
                  .neighbours={{1,3},{0,2},{1,3},{0,2}},\
                  .positions={{1,5}, {4,5}, {2,5}, {3,5}}};
  return b;
}
```

FIGURE 5 – Code de la fonction *init_board_1* (board.c)

Cette fonction renvoie l'implémentation du plateau ci-dessus.
Ce plateau contient 4 positions (*num_positions=4*) et chaque positions a 2 voisins (*num_neighbours=2*). On peut également voir que la position 1 possède les tags NORTH (1) et CORNER (5) et que ses voisins sont les positions 2 et 4.

Nous avons ensuite de la même manière créé un deuxième plateau de 8 positions :

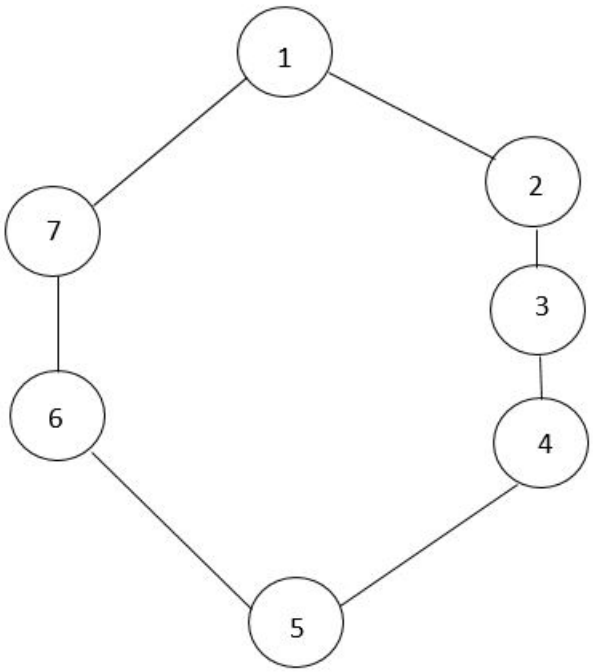

FIGURE 6 – Board 2

Et un troisième de 16 positions :

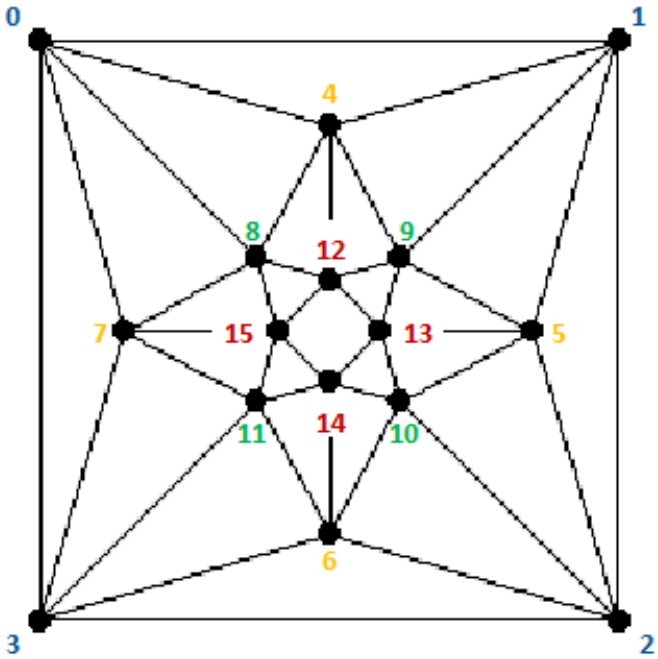

FIGURE 7 – Board 3

Ils sont initialisables respectivement avec les fonctions init_board_2 et init_board_3 disponibles dans le fichier board.c.

2.2 Fonction du board

Nous avons ensuite créé 3 fonctions qui permettent de manipuler le plateau. Ces fonctions se situent dans le fichier board.c.

2.2.1 Distance

```
unsigned int distance(const struct board *b, unsigned int x, unsigned int y)
```

FIGURE 8 – Prototype de la fonction *distance* (board.c)

Cette fonction permet de calculer la distance entre 2 positions("x" et "y") données en entrée. Pour cela on recherche d'abord "y" dans les voisins de "x" (situé dans le champ *neighbours* de la *struct board*) puis dans les voisins des voisins de "x" puis dans les voisins des voisins des voisins de "x" et s'il n'a toujours pas trouvé "y" on renvoie 4 car c'est la distance maximal de nos plateaux.

2.2.2 Nombre de positions

```
unsigned int num_positions(const struct board *b){
  return (*b).num_positions;
}
```

FIGURE 9 – Code de la fonction *num_positions* (board.c)

Cette fonction renvoie tout simplement le nombre de positions du plateau en cours d'utilisation.

2.2.3 A le tag

```
int has_tag(const struct board *b,unsigned int position, unsigned int tag)
```

FIGURE 10 – Prototype de la fonction *has_tag* (board.c)

Cette fonction indique si une position donnée possède oui ou non le tag passé en argument. Pour cela on parcourt simplement tous les tags associés à la position du plateau (grâce au champ *positions* du plateau).

2.3 Affectation des pélicans

Nous devions ensuite créé une structure permettant de lier les positions et les pélicans ainsi que d'associer des contraintes aux pélicans.

Voilà cette structure :

```
struct affect {
  int num_pelicans;
  int id;
  int pelicans[MAX_POS];
  struct constraint tab_constraints[MAX_POS];
  // Warning ! tab_constraints[n] correspond at the pelican with number n+1
};
```

FIGURE 11 – Code de la structure *affect* (affect.c)

Cette structure comporte 4 champs. Tout d'abord *num_pelicans* qui indique le nombre total de pélicans présents sur le plateau, ensuite l'entier *id* qui comme tout à l'heure permet de différencier les différents plateaux (nous avons besoin de remettre le champ *id* ici pour adapter les contraintes aux différents plateaux).
Le 3ème champ est un tableau d'entiers nommé *pelicans* qui permet d'attribuer à chaque position (indice des cases du tableau) un pélican (numéro dans les cases du tableau).

Enfin un champ du type struct constraint nommé *tab_constraints* qui liste les contraintes que doivent respecter chaque pélican. Struct constraint est juste un triplet d'entiers (n,p1,p2) avec *n* le numéro de la contrainte, *p1* le pélican affecté par la contrainte et *p2* le deuxième pélican en cas de contrainte bi-pélican (que nous détaillerons dans la partie suivante).

2.4 Fonction de la structure affect

Nous avons ensuite créé des fonctions agissant sur la structure affect notamment pour affecter des pélicans.

2.4.1 Nombre de pélicans

La première fonction associée à affect est la fonction suivante :

```
int num_pelicans(const struct affect *a){
  return  (*a).num_pelicans;
```

FIGURE 12 – Code de la fonction *num_pelicans* (affect.c)

Cette fonction renvoie simplement le nombre total de pélicans en jeu.

2.4.2 Initialisation

Il fallait également initialiser l'affectation. Pour cela nous avons créé deux méthodes.

Une première consiste à placer le pélican 1 à la position n+1, le pélican 2 à la position n+2, etc. Avec n=(nombre de positions maximum du plateau)-(nombre de pélicans).
Par exemple le placement de 5 pélicans sur le plateau 2 (8 positions) sera fait comme ça : [0,0,0,1,2,3,4,5] (Le placement est effectué de cette sorte car il est nécessaire que le tableau soit trié dans cette ordre là pour une fonction *next_permutation* que nous verrons ultérieurement.)

```
struct affect init_affect(struct board *b, int num)
```

FIGURE 13 – Code de la fonction *init_affect* (affect.c)

La fonction prend en paramètres le plateau et le nombre de pélicans à affecter (num) puis elle affecte les pélicans dans l'ordre.

2.4.3 Initialisation aléatoire

La deuxième méthode affecte cette fois ci les pélicans au hasard :

```
struct affect init_affect_random(struct board *b, int num)
```

FIGURE 14 – Code de la fonction *init_affect_random* (affect.c)

Pour affecter les pélicans aléatoirement nous stockons le numéro des positions dans un tableau (*t*), puis pour chaque pélican nous effectuons un tirage aléatoire dans ce tableau pour déterminer sa position et nous enlevons cette position du tableau :

```
int pos=0;
int index_pos=0;
for(j=1; j<(num+1); j++){ //j= piaf number
  index_pos = rand_c(n);
  pos=t[index_pos];
  a.pelicans[pos]=j;
  n=n-1;

  //Exchange position of number used and last number of table t
  int tmp=t[n];
  t[n]=t[index_pos];
  t[index_pos]=tmp;
```

FIGURE 15 – Extrait de la fonction *init_affect_random* (affect.c)

3 Fonctions liées aux contraintes

Maintenant que nous disposons d'un plateau de jeu et d'une affectation des pélicans sur celui-ci, nous pouvons nous intéresser aux contraintes à affecter. Le code traité dans cette partie se situe entièrement dans le fichier *constraint.c*.
Nous avons divisé le problème des contraintes en deux sous-problèmes : les contraintes dépendantes d'un seul et de deux pélicans. Ensuite nous avons crée une fonction générale vérifiant si la contrainte donnée d'un pélican est satisfaite.

3.1 Contraintes mono-pélican

Tout d'abord nous allons montrer et expliquer le code de la fonction de base *mono_pelican* afin de pouvoir parler des fonctions qui en découlent.

```
//Function for 1 pelican with only 1 constraint :
int mono_pelican(struct board *b, struct affect *a, int p1, int c){
  int i;
  int pos;
  //Look for the position of the pelican p1 on the board
  for(i=0; i<num_positions(b); i++){
    if( (*a).pelicans[i]==p1){
      pos = i;
      /* Test if the constraint c is respected
         Return True (1) or False (0) */
      return (has_tag(b,pos,c));
    }
  }
  //If the pelican isn't on the board :
  printf("The pelican doesn't exist !\n");
  return 42; //Because 42
}
```

FIGURE 16 – Code de la fonction *mono_pelican* (constraint.c)

Cette fonction prend en paramètres le plateau de jeu, l'affectation de celui-ci ainsi que le numéro du pélican et sa contrainte (correspondant au numéro de son tag). Cette fonction retourne un entier. Il indique si le pélican se trouve sur une position vérifiant la contrainte donnée en entrée.

Cette fonction va donc chercher la position du pélican sur le plateau de jeu en parcourant le tableau de l'affectation de celui-ci. Une fois sa position trouvée, elle va vérifier si cette position possède bien le tag dont le numéro est donné en entrée. Si oui elle retourne 1, sinon 0. En revanche si le pélican ne se trouve pas sur le plateau, un message d'erreur est affiché.

Deux variantes de cette fonction vérifient si un pélican respecte deux contraintes A et B, ou bien deux contraintes A ou B. Nous pouvons ainsi créer par exemple la fonction *in_north_and_corner*.

Une fois ces trois fonctions de base codées, il suffit de les appeler avec le bon tag pour créer toutes les fonctions de contraintes *mono_pelican* que nous voulons. Comme par exemple :

```
int in_corner(struct board *b, struct affect *a, int p1){
  return mono_pelican(b,a,p1,TAG_CORNER);
}
```

FIGURE 17 – Code de la fonction *in_corner* (constraint.c)

Voici donc la liste des contraintes que nous avons crées :

- in_north, in_south, in_east, in_west
- in_corner
- in_cercle 1, in_cercle 2, in_cercle 3
- in_north_or_south
- in_north_and_corner

3.2 Contraintes bi-pélican

Comme pour les contraintes mono-pélican, nous allons dans un premier temps montrer et expliquer le code de la fonction de base *bi_pelican* afin de pouvoir parler des fonctions qui en découlent.

```
//Function for 2 pelicans :
int bi_pelican(struct board *b, struct affect *a, int p1, int p2, int c1, int c2)
```

FIGURE 18 – Prototype de la fonction *bi_pelican* (constraint.c)

Cette fonction ne diffère que peu de la fonction *mono_pelican*. Elle prend deux paramètres de plus en entrée : le deuxième pélican et le deuxième numéro de tag de contrainte. Son rôle est cette fois-ci de vérifier non pas une mais deux positions de pélicans et leurs tags respectifs.

En effet, cette fonction va chercher la position des deux pélicans sur le plateau de jeu puis, une fois leurs positions trouvées, vérifier si ces positions possèdent les tags donnés en entrée. Tout comme la première fois, si c'est le cas la fonction retourne 1, sinon 0. De la même façon, si un des deux pélicans ne se trouve pas sur le plateau, un message d'erreur est affiché.

Deux variantes de cette fonction vérifient si les deux pélicans respectent deux mêmes contraintes, ou bien deux contraintes différentes. Une troisième variante vérifie quant à elle une contrainte commune aux pélicans ainsi qu'une contrainte liée à la distance les séparant.

Une fois ces quatre fonctions de base codées, il suffit, comme pour les contraintes mono-pélican, de les appeler avec le bon tag pour créer toutes les fonctions de contraintes *bi_pelican* que nous voulons. Comme par exemple :

```
int on_same_side(struct board *b, struct affect *a, int p1, int p2){
  return (bi_pelican_same(b,a,p1,p2,TAG_NORTH) || bi_pelican_same(b,a,p1,p2,TAG_SOUTH)
  || bi_pelican_same(b,a,p1,p2,TAG_WEST) || bi_pelican_same(b,a,p1,p2,TAG_EAST) );
}
```

FIGURE 19 – Code de la fonction *on_same_side* (constraint.c)

La fonction *bi_pelican_same* fait partie des variantes de la fonction *bi_pelican*, c'est celle qui l'appelle avec deux fois le même numéro de tag, pour vérifier que deux pélicans respectent la même contrainte.

Voici donc la liste des contraintes que nous avons créées ainsi que leur explication si nécessaire :

- in_front_of
- on_same_side
- same_angle (pélicans voisins et dans un coin)
- same_cercle

3.3 Vérification de la satisfaction d'une contrainte

Maintenant que toutes les fonctions de contraintes ont été codées, nous pouvons imaginer une fonction générale vérifiant pour n'importe quel pélican la satisfaction d'une contrainte donnée.

```
int apply_constraint(struct board *b, struct affect *a, int n, int p1, int p2){
    switch (n) {
    case 0:
      return 1;
    case 1 :
      return in_north(b,a,p1);
    case 2 :
      return in_south(b,a,p1);
    case 3 :
      return in_west(b,a,p1);
    case 4 :
      return in_east(b,a,p1);
    case 5 :
      return in_corner(b,a,p1);
    case 6 :
      return in_north_or_south(b,a,p1);
    case 7 :
      return in_north_and_corner(b,a,p1);
    case 8 :
      return in_front_of(b,a,p1,p2);
    //Constraints for board 2 :
    case 9 :
      return on_same_side(b,a,p1,p2);
    case 10 :
      return same_angle(b,a,p1,p2);
    //Constraints for board 2 & 3 :
    case 11 :
      return same_cercle(b,a,p1,p2);
    case 12 :
      return in_cercle_1(b,a,p1);
    case 13 :
      return in_cercle_2(b,a,p1);
    case 14 :
      return in_cercle_3(b,a,p1);
    }
    printf("Warning : You ask a contraint mono-pelican with two pelicans in argument !");
  return 42;
}
```

FIGURE 20 – Code de la fonction *apply_constraint* (constraint.c)

Nous avons donc séparé en plusieurs cas les différentes contraintes, associant ainsi un numéro à chaque contrainte : la contrainte n°11 correspond dorénavant à la contrainte *in_cercle_1*.

Le principe de cette fonction est très simple, en prenant en entrée un entier *n* correspondant au numéro de la contrainte, la fonction va appeler la fonction de contrainte correspondante, codée auparavant. La fonction doit donc également prendre en entrée les paramètres nécessaires à cette fonction auxiliaire. De la même manière que celle-ci, la fonction *apply_constraint* renvoie un entier (0 ou 1) indiquant si la contrainte est vérifiée.

Par la suite, nous avons rajouté deux nouvelles contraintes importantes mais nécessitant une modification de notre fonction *apply_constraint*. Ces contraintes sont les suivantes : "Avoir la même contrainte que tel pélican" et "Avoir la contrainte opposée à tel pélican".

Afin d'implémenter ces nouvelles contraintes nous avons modifié la fonction *apply_constraint* et fait en sorte qu'elle devienne récursive afin de manipuler l'ensemble des contraintes. Nous avons également crée des fonctions nous permettant de récupérer le numéro de contrainte (et éventuellement le deuxième pélican associé à cette contrainte) de n'importe quel pélican. Ceci étant fait, il a suffi de rajouter deux cas à notre fonction *apply_constraint* :

```
case 8 : //p1 wants the same constraint than p2
  return (apply_constraint_rec(b,a,recover_constraint(a,p2),p1,recover_p2(a,p2),
  tab, size));
case 9: //p1 wants the opposite constraint than p2
  return (!(apply_constraint_rec(b,a,recover_constraint(a,p2),p1,recover_p2(a,p2),
  tab, size)));
```

FIGURE 21 – Extrait du code de la nouvelle fonction *apply_constraint_rec* (constraint.c)

Le problème posé par ces nouvelles contraintes est l'apparition de cycle dans l'appel de fonction et donc de boucles infinies. Afin de détecter et d'éviter ceci, la fonction *apply_constraint_rec* prend deux nouveaux paramètres en entrée : un tableau et un entier. Ils vont servir à gérer ces cycles. Le tableau stockera le numéro du pélican sur lequel est appliqué la contrainte. Si l'on retrouve deux fois le même pélican dans ce tableau au cours des appels de fonction, cela signifie que nous sommes en train de boucler. La fonction retournera alors l'entier 1 signifiant que la contrainte est respectée.

En effet, si un pélican p1 veut la même contrainte que le pélican p2, p2 veut la même que p3 et p3 la même que p1, nous nous trouvons bien dans le cas d'un cycle. Selon nous cela revient à dire que ces trois pélicans veulent exactement la même contrainte, c'est le cas, leurs contraintes sont donc satisfaites.

4 Premier solveur

Nous nous sommes ensuite intéressés à la création d'une fonction solver permettant de renvoyer l'ensemble des placements possibles des pélicans qui respectent leurs contraintes sur un plateau. On stocke l'ensemble des solutions dans un tableau (*tab_rec*).

```
void solver(struct board *b, struct affect *a, int resultats[][MAX_POS])
```

FIGURE 22 – Prototype de la fonction *solver* (solver.c)

Le principe de l'algorithme est simple. Nous parcourons l'ensemble des placements possibles et vérifions à chaque fois si les contraintes sont respectées, si tel est le cas on ajoute la solution au tableau, sinon on passe à la permutation suivante grâce à la fonction *next_permutation*.

La fonction *next_permutation* permet de renvoyer la permutation suivante d'un tableau et nous sert donc à avoir toutes les permutations des placements des pélicans sur le plateau dans l'ordre. Nous nous sommes inspirés d'un algorithme trouvé sur *www.nayuki.io*

```
int next_permutation(int *tab, int l){
  //Find longest non-increasing suffix
  int i = l-1; //l=lenght of tab
  while( (i>0) && tab[i-1]>=tab[i])
    i--;
  //Now i is the head index of the suffix
  //Are we at the last permutation already ?
  if (i <= 0)
    return 0;
  //Let tab[i-1] be the pivot
  //Find rightmost element that exceeds the pivot
  int j = l-1;
  while (tab[j]<=tab[i-1])
    j--;
  //Now the value tab[j] will become the new pivot
  //Assertion: j>=i
  //Swap the pivot with j
  int tmp = tab[i-1];
  tab[i-1]=tab[j];
  tab[j]=tmp;
  //Reverse the suffix
  j=l-1;
  while (i<j) {
    tmp = tab[i];
    tab[i]=tab[j];
    tab[j]=tmp;
    i++;
    j--;
  return 1;
```

FIGURE 23 – Code de la fonction *next_permutation* (function.c)

Voilà comment fonctionne l'algorithme :

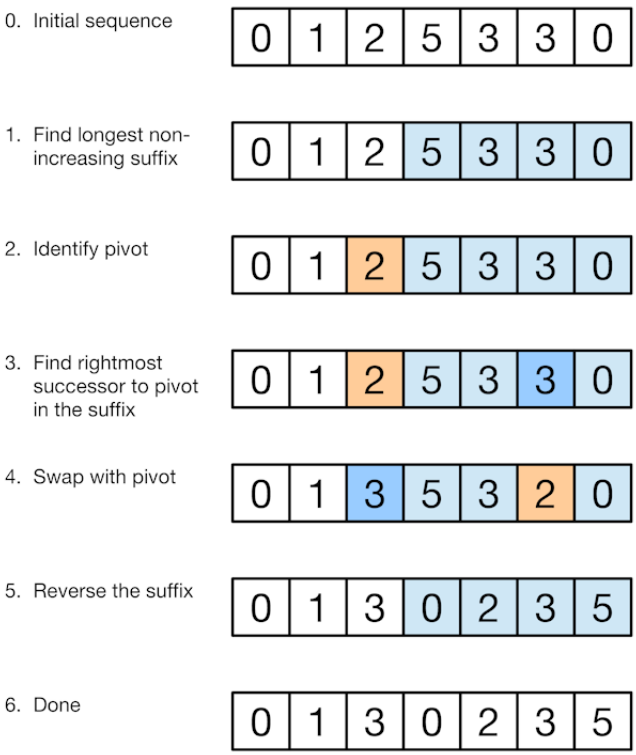

FIGURE 24 – Explication du principe (source : www.nayuki.io)

Par exemple si nous partons du tableau [1,2,3]

-Après 1 application de next_permutation le tableau devient : [1,3,2]

-Après 2 applications de next_permutation le tableau devient : [2,1,3]

-Après 3 applications de next_permutation le tableau devient : [2,3,1]

-Après 4 applications de next_permutation le tableau devient : [3,1,2]

-Après 5 applications de next_permutation le tableau devient : [3,2,1]

Nous obtenons donc bien au final toutes les permutations du tableau.

Ce solver est malheureusement très coûteux en temps avec sa complexité en o(n !), n étant le nombre de positions du plateau. En effet pour le plateau de 4 cases il faut tester 4 !=24 placements, pour celui de 8 il faut tester 8 !=40320 placements et pour celui de 16 il faut tester 16 ! possibilités ! ! Nous ne verrons donc jamais la fin de cette algorithme pour le plateau de 16.
C'est pour cela qu'il nous a fallu trouver une autre solution moins gourmande en temps.

5 Deuxième solveur

Le code traité dans cette partie se situe dans deux fichiers : *constraint.c*, correspondant à la première sous-partie, et *z3solver.c* correspondant à la seconde.

5.1 Principe de sur-approximation

Notre premier solveur fonctionne correctement, mais comme dit précédemment sa complexité en temps est très importante.
Cependant il est inutile de chercher à savoir si le pélican voulant être placé *Au nord* satisfait sa contrainte en étant sur une case possédant le tag *Sud*. C'est une perte de temps. Une façon de l'éviter est d'effectuer une sur-approximation de la solution.

Explicitons un peu nos propos. En quoi va consister cette sur-approximation ? Au lieu d'explorer l'ensemble des contraintes, nous allons réduire notre recherche. Nous allons associer à chaque contrainte l'ensemble de ses positions possibles. C'est ce qu'effectue la fonction *over approximation*.

Nous n'allons pas inclure directement le code de cette fonction dans ce rapport, beaucoup trop long et fastidieux, mais plutôt en expliquer le principe.

Nous avons décidé de stocker dans des tableaux les positions associées à chaque contrainte, dans deux tableaux plus exactement. Deux tableaux correspondant à deux catégories.
Prenons par exemple la contrainte *in west*, le pélican ne peut se trouver que sur un certain nombre de positions, celles possédant le tag *west*, pour satisfaire la contrainte. En revanche la contrainte *in front of* dépend d'un autre pélican, donc d'une deuxième position. Il y aura alors plusieurs listes de positions possibles pour satisfaire cette contrainte, dépendant de la position du deuxième pélican.

En bref, ce sont les deux mêmes catégories que pour les fonctions liées aux contraintes : mono et bi-pélican. Le premier tableau sera donc un tableau à deux dimensions tandis que le second tableau en aura trois.

Voici un schéma illustrant le stockage des positions de chaque contrainte dans ces tableaux.

Contraintes	in_north	in_south	in_weast	etc...
Positions associées	0	2	4	...
	1	5	7	..
		6	8	...

Table 1 – Tableau stockant la sur-approximation des contraintes mono-pélican

Dans ce tableau, on voit qu'à chaque contrainte est associée la liste des positions la satisfaisant. Par exemple si un pélican veut être au sud, il devra se placer sur la case 2, 5 ou 6.

Contraintes / Position du pélican	in_front_of	same_cercle	etc...
0	[3,4,6]	[4,7,10]	...
1	[5,8,13]	[ ]	...
2	[6,9,11,14]	[3,6,8]	...
...	...	...	...

Table 2 – Tableau stockant la sur-approximation des contraintes bi-pélican

Dans ce tableau, à chaque position du pélican sur lequel est appliqué la contrainte (1^{re} colonne de gauche), correspond la liste des positions satisfaisant cette contrainte. Pour une contrainte il y aura donc n listes possible, avec nn le nombre de cases du plateau. Par exemple ici, pour qu'un pélican soit en face d'un pélican se trouvant sur la case 1, il devra se placer sur la case 5, 8 ou 13.

Maintenant que nous disposons pour chaque contrainte d'une ou plusieurs liste(s) de positions associées, nous allons pouvoir les utiliser avec le SAT-solver z3.

5.2 Utilisation du solveur Z3

5.2.1 Principe d'utilisation et écriture du fichier.smt2

Le solveur z3 (programme écrit pour résoudre efficacement des problèmes de logique), va nous permettre de trouver, en un temps très convenable, une solution. Mais pour cela nous devons rédiger un document dans un format accepté par le solveur. Ce format utilise des formules logiques du 1^{er} ordre. Nous allons donc représenter les contraintes avec un ensemble fini de variables pi_cj signifiant "Pélican i sur la case j".

Voici un exemple de déclaration de contrainte au format accepté par z3 :

```
(assert (or p1_c1 p1_c2))
```

Afin d'utiliser le solveur z3, nous devons donc rédiger au préalable un document (au format .smt2) comportant la définition de l'ensemble des variables et des contraintes associées.

Une fonction déclarera l'ensemble des variables de cette manière :

```
(declare-const pi_cj Bool)
```

Deux fonctions vérifieront qu'il y ait au moins et au maximum une case pour chaque pélican.

Nous pouvons ensuite passer à l'écriture des contraintes. Nous avons crée pour cela une fonction dépendant d'un entier *n* indiquant le numéro de la contrainte, tout comme la fonction *apply constraint*. Nous appelerons donc cette fonction en récupérant le numéro de la contrainte de chaque pélican stocké dans le struct constraint du struct affect.

Encore une fois pour chaque numéro de contrainte, une fonction auxiliaire est appelée. Cette fonction traduit en logique du 1^{er} ordre la signification de la contrainte. Pour ce faire, elle cherche dans le tableau de sur-approximation, les positions possibles pour la dite contrainte. Une fois de plus, une distinction est faite entre les contraintes mono et bi-pélican. Vous pouvez observer la différence d'écriture entre ces deux types de contraintes dans l'exemple ci-dessous.

```
//Contrainte mono-pelican
(assert (or p3_c0 p3_c1))
//Contrainte bi-pelican
(assert (and (implies p4_c4 (or p1_c4 p1_c5 p1_c6 p1_c7))
             (implies p4_c5 (or p1_c4 p1_c5 p1_c6 p1_c7))))
```

Une fois le fichier.smt2 écrit, nous pouvons appeler le solveur z3 et écrire sa réponse dans un fichier au format texte.

5.2.2 Restitution de la solution

A présent c'est au tour de la fonction *aff res* de nous dire si une solution a été trouvée, et le cas échéant l'afficher.

Pour y parvenir, cette fonction parcourt le fichier.txt contenant la réponse du solveur z3. Si aucune solution n'a été trouvée, la première ligne du fichier contiendra "unsat", auquel cas la fonction renvoie 0.
En revanche si la première ligne contient le mot "sat" alors un modèle répondant aux différentes assertions a été trouvé. Il ne reste plus qu'à en extraire les informations.

Si un modèle a été trouvé, le fichier de sortie est de la forme suivante :

```
sat
(model
  (define-fun p4_c8 () Bool
    false)
  (define-fun p3_c15 () Bool
    false)
  (define-fun p1_c14 () Bool
    true)
  (define-fun p3_c0 () Bool
    false)
  (define-fun p4_c5 () Bool
    true)
  (define-fun p1_c11 () Bool
    false)
    ...
```

La fonction *aff res* recherche donc les lignes contenant "true" afin de remonter aux pélicans et cases associées une ligne plus haut. Une fois tous les pélicans trouvés, elle les affiche et retourne l'entier 1.

Il peut donc arriver qu'aucun modèle ne soit trouvé et que la fonction renvoie 0. Dans ce cas, nous tentons une deuxième approche...

5.2.3 Recherche d'une seconde solution maximisant les contraintes

Si aucune solution n'est possible pour satisfaire toutes les contraintes, nous allons en chercher une autre satisfaisant le maximum de contraintes.

Pour obtenir ce résultat, nous allons recommencer les mêmes démarches que précédemment, à une différence près. Si aucune solution n'est trouvée pour n contraintes, alors nous essaierons d'en trouver pour n-1, et ainsi de suite.

Afin d'y parvenir, à chaque nombre de contraintes à satisfaire réduit, nous écrirons plusieurs fichiers.smt2 correspondant à l'ensemble des associations de contraintes possibles. Par exemple, si nous avons 4 pélicans numérotés 1,2,3,4 et qu'il est impossible de satisfaire toutes leurs contraintes. Alors nous tenterons de satisfaire les contraintes des pélicans 1,2,3 puis 1,2,4 puis 1,3,4 puis 2,3,4. En bref nous appellerons z3 pour toutes les séquences existantes avec $n-1$ pélicans parmi les n pélicans initiaux. Si cela ne suffit pas, nous ferons de même avec $n-2$ pélicans, $n-3$, etc...

Nous effectuons cette démarche tant que le nombre de contraintes enlevées est inférieur ou égal au nombre total de contraintes. Si une solution est trouvée avant, le processus s'arrête et la solution s'affiche.

6 Tests

Nous avons également créé des tests pour voir si notre code fonctionnait correctement. Toutes les fonctions décrites dans cette partie se situent dans le fichier *test_function.c*. La première fonction que nous avons testé est la fonction *distances*.

6.1 Distances

```
int test_distance(const struct board *b3, const struct board *b2)
```

FIGURE 25 – Prototype de la fonction *test_distance* (test_function.c)

Pour ce test nous avons utilisé les plateaux 2 et 3 et codé en dur pour chacun d'entre eux les distances entre chaque case. Par exemple avec le plateau 3, comportant pour rappel 16 cases :

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	1	2	1	1	2	2	1	1	2	3	2	2	3	3	2
2	4	0	1	2	1	1	2	2	2	1	2	3	2	2	3	3
3	4	4	0	1	2	1	1	2	3	2	1	2	3	2	2	3
4	4	4	4	0	2	2	1	1	2	3	2	1	3	3	2	2
5	4	4	4	4	0	2	3	2	1	1	3	3	1	2	3	2
6	4	4	4	4	4	0	2	3	3	1	1	3	2	1	2	3
7	4	4	4	4	4	4	0	2	3	3	1	1	3	2	1	2
8	4	4	4	4	4	4	4	0	1	3	3	1	2	3	2	1
9	4	4	4	4	4	4	4	4	0	2	3	2	1	2	2	1
10	4	4	4	4	4	4	4	4	4	0	2	3	1	1	2	2
11	4	4	4	4	4	4	4	4	4	4	0	2	2	1	1	2
12	4	4	4	4	4	4	4	4	4	4	4	0	2	2	1	1
13	4	4	4	4	4	4	4	4	4	4	4	4	0	1	2	1
14	4	4	4	4	4	4	4	4	4	4	4	4	4	0	1	2
15	4	4	4	4	4	4	4	4	4	4	4	4	4	4	0	1
16	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	0

FIGURE 26 – Distances entre chaque case

Nous parcourons ensuite ce tableau et nous le comparons avec les résultats renvoyés par la fonction *distances*. Nous avons mis des 4 en dessous de la diagonale car pour le plateau 3 la distance max vaut 3. Ceci permet de voir si l'on parcourt bien que la bonne partie du tableau.

Si toutes les distances correspondent pour le plateau 2 et 3 nous affichons :

```
Test distance:
                        -Sub test 1   : OK
                        -Sub test 2   : OK
*Test distance total: OK
```

FIGURE 27 – Affichage des tests de la fonction *distances*

Sub test 1 correspond au test sur le board de 16 et Sub test 2 sur celui de 8.

```
Test distance:
                        -Sub test 1   : FAILED
                        -Sub test 2   : OK
*Test distance total: FAILED    (1 error)
```

FIGURE 28 – Affichage d'un test raté de la fonction *distances*

Mais en cas d'erreur on affiche une erreur, comme on peut le voir ci-dessus avec le board 3.

6.2 A le tag

Nous avons ensuite testé la fonction *has_tag* :

```
int test_has_tag(const struct board *b)
```

FIGURE 29 – Prototype de la fonction *test_has_tag* (test_function.c)

Cette fonction vérifie simplement que *has_tag* renvoie les bons tags sur le plateau 1.

6.3 Permutation suivante

Nous avons également fait des tests sur la fonction next_permutation dont le fonctionnement est décrit dans la partie 4 - Premier Solveur.

```
int test_next_permutation()
```

FIGURE 30 – Prototype de la fonction *test_next_permutation* (test_function.c)

Pour ce test nous essayons simplement plusieurs tableaux différents et regardons si le résultat retourné par *next_permutation* est celui attendu.

6.4 Solver

Nous avons évidemment testé notre fonction solver :

```
int test_solver(struct board *b,struct affect *a,struct board *b2,struct affect *a2)
```

FIGURE 31 – Prototype de la fonction *test_solver* (test_function.c)

Nous avons d'abord effectué plusieurs tests du solver sur le plateau 1 en faisant varier les contraintes. Puis des tests sur le plateau 2. Nous n'avons pas fait de test sur le plateau 3 car comme vu précédemment, l'exécution du solveur prendrait trop de temps.

6.5 Z3 solver

Notre dernier test est celui du solver z3 :

```
int test_z3solver(struct board *b,struct affect *a,struct board *b2,struct affect *a2)
```

FIGURE 32 – Prototype de la fonction *test_z3solver* (test_function.c)

Il fonctionne de la même manière que le *test_solver*.

L'ensemble de ces tests nous a permis de détecter quelques bugs et oublis dans notre code. Par exemple la fonction *distance* ne fonctionnait pas pour le plateau 2. Nous avions également oublié des cas dans l'application de la contrainte *in_front_of*. Nous avons aussi détecté un problème avec la fonction *next_permutation* lorsque que le tableau n'était pas entièrement rempli de pélicans.

Conclusion

Au terme de ce projet, nous avons réussi à reproduire le principe du jeu "Pelican Cove". Nous avons pour cela dû apprendre à travailler en équipe. C'est à dire se répartir équitablement les tâches, savoir communiquer afin de discuter des points théoriques mais aussi se tenir au courant de nos avancées respectives afin de faciliter le travail de l'autre.

D'un point de vue technique, ce projet nous aura bien entendu permis de mieux maîtriser le langage C, mais aussi d'apprendre à rebondir. En effet au cours de notre démarche, nous avons constaté que notre première idée de solveur, bien que correcte, n'était pas forcément la plus efficace. Nous avons alors réussi à reprogrammer, d'autant plus vite grâce au travail fourni sur le précédent, un nouveau solveur dont la complexité en temps était bien plus avantageuse. La série de test mise en place nous aura également permis de nous rendre compte parfois, de manquements voire d'erreurs dans notre code, que nous n'aurions pas forcément remarqués sans cette vérification.

Enfin d'un point de vue plus personnel, nous pensons que ce genre de projet nous apprenant à gérer un travail sur le long terme est un avant-goût de nos métiers futurs. Nous avons trouvé ce projet enrichissant, et même si parfois il a pu s'avérer ardu, il nous a permis de gagner en réflexion et maturité.