Facetious pelican

Projet de programmation C

Parpaite Thibault
Menanteau Yoann

Projet de semestre
ENSEIRB-MATMECA
M. Bienvenu Jean-Luc
Semestre 5
06 janvier 2017

Table des matières

1 Introduction **2**
- 1.1 Le pelican cove . . . 2
- 1.2 Organisation . . . 2
- 1.3 Les étapes . . . 3

2 Baseline **4**
- 2.1 Problématiques . . . 4
- 2.2 Choix . . . 4
- 2.3 Génération des éléments de base . . . 5

3 Achievement 1 **11**
- 3.1 Problématique . . . 11
- 3.2 Le solveur . . . 11
- 3.3 Fonctionnement de apply constraint . . . 12

4 Achievement 2 **14**
- 4.1 Problématiques . . . 14
- 4.2 Choix . . . 14
- 4.3 apply_constraint_rec . . . 15
- 4.4 Traitement des dépendances . . . 15
- 4.5 Difficultés . . . 17

5 Achievement 3 **18**
- 5.1 Problématiques . . . 18
- 5.2 Choix . . . 18
- 5.3 Génération du script . . . 19
- 5.4 apply_constraint_z3 . . . 20
- 5.5 Le solveur Z3 . . . 21
- 5.6 Difficultés . . . 22

6 Achievement 4 **24**
- 6.1 Résumé . . . 24
- 6.2 Custom_type . . . 24
- 6.3 Pré-calcule des positions par tag . . . 25
- 6.4 Pré-calcule des couples de positions par contrainte bi-pinguin . . . 26
- 6.5 Difficultés . . . 26

7 Achievement 5 : backtracking ? 27
7.1 Backtracking 27

8 Conclusion 29

1 Introduction

Ce rapport relate notre conception du projet de premier semestre de la filière informatique. Ce projet avait pour thème le jeu du Pelican Cove. Il s'est déroulé en quatre grosses étapes résultant chacune sur un achèvement.

1.1 Le pelican cove

Le pelican cove est un jeu de plateau qui met en scène des ... pélicans. Le plateau contient un certain nombre d'emplacements. Le joueur doit ainsi placer chaque pélican sur le plateau cependant, les pélicans sont facétieux. Chaque pélican énonce des contraintes. Ces contraintes sont matérialisées sous forme de cartes qu'on tire aléatoirement. Il y-a ainsi une contrainte par pélican. Ces contraintes peuvent soit être mono-pélican ou bi-pélican. S'il s'agit d'une contrainte mono-pélican, alors il s'agit d'une contrainte de position ; le pélican souhaite par exemple être au Nord, au Sud ... . S'il s'agit d'une contrainte bi-pélican, alors il peut par exemple vouloir faire face à un autre pélican ou bien être à côté.

1.1.1 Périmètre

Le projet consiste principalement à développer des algorithmes permettant de trouver les meilleurs placements. Cela nécessite donc de modéliser les éléments de base du jeu (le plateau, les pélicans, le tirage aléatoire des cartes, etc ...).

1.2 Organisation

1.2.1 Membres

Ce travail a été effectué par une équipe composée de deux membres : PARPAITE Thibault et MENANTEAU Yoann.

1.2.2 Outils

Les étapes menant au livrable nécessitent certains outils. L'écriture du code n'a fait appel à aucun IDE. Elle s'est faite via l'éditeur de texte Emacs et la compilation avec Gcc et Mingw. Afin de faciliter la compilation, le logiciel Cmake a été utilisé. La documentation, elle, a été générée avec l'outil Doxygen. En ce qui concerne le rapport,

de nombreux schéma ont été produit par Modelio et le rapport lui-même a été écrit en Latex.

1.2.3 Partage du projet

Afin de travailler en collaboration, quelques outils de partage ont été utilisés. Le partage du code s'est fait grâce à l'outil subversion. L'écriture du rapport a été faite en collaboration avec un service en ligne nommé Sharelatex.

1.2.4 Partage des tâches

Souvent, le travail sur les différentes étapes a été fait en parallèle. Lorsqu'une personne terminait l'achèvement un, la seconde avançait sur l'achèvement deux tout en s'accordant sur la procédure à suivre.

1.3 Les étapes

La suite va vous relater les différentes étapes, les problèmes rencontrés ainsi que leur résolution. Le rapport est donc divisé en six grandes parties. Exceptée la première, chacune renvoie à un achèvement. Pour chaque partie, il y aura un préambule qui a pour objectif de résumer les problèmes rencontrés, les choix pris et les solutions trouvées. Ensuite, les sous-parties suivantes détailleront les solutions. Le détail des solutions n'a pas pour vocation de reproduire le code, il y-a pour ça le code lui-même et la documentation Doxygen. Cela a plutôt pour objectif de donner les grandes lignes et les clés pour ensuite comprendre le fonctionnement des algorithmes.

2 Baseline

2.1 Problématiques

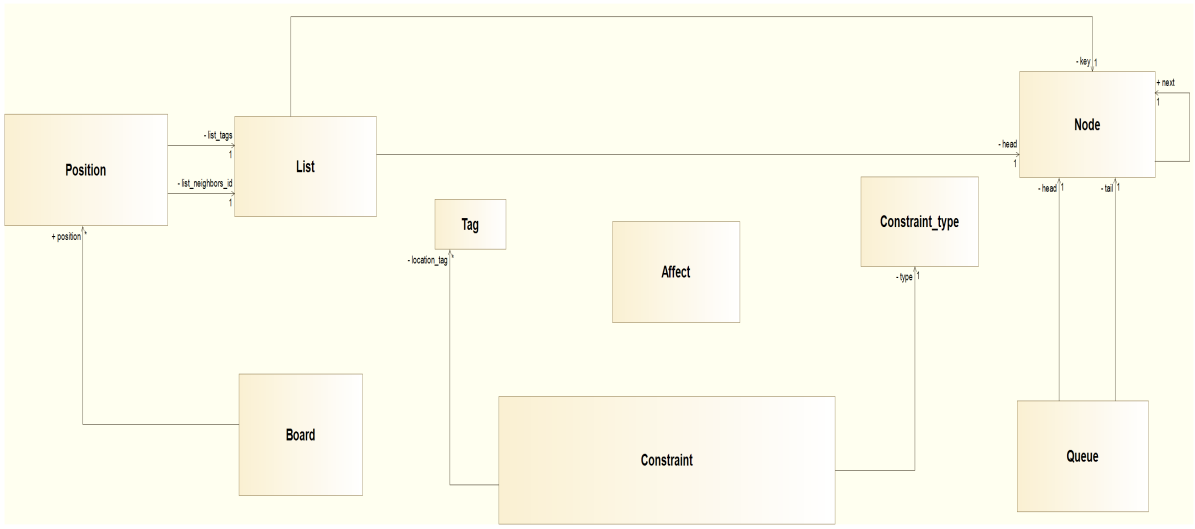
Cette première partie visait à poser les bases du projet. Ainsi, c'était une partie importante voire même la plus importante. Elle posait déjà un certain nombre de questions. Ces questions concernaient en bonne partie l'organisation du code, le choix de telle ou telle structure et de tel ou tel paramètre. Cette partie a fait l'objet de nombreux changements au fil des séances pour déterminer les meilleurs choix. Certains de ces choix concernaient le code mais d'autres portaient plus sur l'interprétation des consignes. Ainsi, on a été amené à se demander quelle serait la board qu'on implémenterait. Allions-nous implémenter le plateau du jeu d'origine ? Ou plutôt l'étendre et essayer de généraliser les règles du jeu à n'importe quel type de plateau. La question des tags a aussi été abordée. Les tags sont un ensemble d'éléments pour caractériser une position. Une position au Nord Est aura par exemple les tags TAG_NORTH et TAG_EAST. Cependant, quels tags choisir ? Pour quel plateau ? Enfin, si l'on choisit un plateau différent de celui d'origine, devrions-nous créer de nouvelles contraintes ? On a aussi été face à des choix comme celui d'astreindre chaque pélican à une seule position et chaque position à un seul pélican. Autant de questions qui ont été la source de réflexions et remises en question tout au long de cette partie mais aussi tout au long du projet.

2.2 Choix

2.2.1 Le découpage

L'organisation du code, bien qu'en C, se rapproche beaucoup de la programmation orientée objet. Ici, il n'est pas question de classes mais de structures. Il n'est pas non plus question de méthodes mais de fonctions. De fait, chaque structure est déclarée dans son fichier source et n'est ainsi accessible par le reste du programme. L'accès à ses membres se fait par un ensemble de fonctions, des accesseurs (getter) disponibles par le reste de l'environnement. La modification lorsqu'elle est nécessaire est possible via des mutateurs (setter). De même, un ensemble d'actions pour chaque "objet" sont possibles via les fonctions qui leur sont associées. Ainsi, le code étant très proche de l'orienté objet, il est possible de le représenter via un diagramme de classe dans lequel chaque structure est une classe et chacun de ses prototypes, des méthodes. Bien sûr, ceci n'est qu'un choix de représentation, le code lui n'est ni du C++, ni du Java. Le graphique suivant donne une vue très globale sur la structuration du code :

FIGURE 2.1 – Représentation de la structure du programme



On y voit la présence d'une structure board. Celle-ci va comme son nom l'indique représenter la board, le plateau. Cette board comprend un tableau de positions dont la taille varie selon la board implémentée. Une positions (elle-même une structure) contient une liste de tags ainsi qu'une liste (liste chaînée) de voisins. Cela sera particulièrement utile pour déterminer les distances. Nous avons fait le choix de représenter une contrainte par une structure. Celle-ci est liée à deux énumérations : "Tag" et "Constraint_type" qui représentent respectivement les tags et les types de contrainte (mono et bi-pingouin). Des piles ont aussi été implémentées puisque utilisées par l'algorithme de calcul des distances. Le choix a été fait de les représenter avec des listes plutôt qu'avec des tableaux. Cela vient du fait que l'on souhaitait gérer n'importe quelle taille de pile. Ceci est un diagramme qui sera détaillé et complété au fil des parties. Le tout sera accompagné d'autres schéma visant plus à représenter le séquençage de certaines fonctions.

2.3 Génération des éléments de base

2.3.1 La board

Fonctionnement

La génération de la board n'est pas aléatoire. Plusieurs boards ont été implémentées, une de taille quatre et une autre de taille huit. On a aussi effectué certains tests avec

une de seize avant de retourner sur des graphes plus petits au vu de la complexité des fonctions. La taille que l'on mentionne renvoie au nombre de noeuds présents sur le graphe. Les boards peuvent être représentées par deux graphes planaires. Les plateaux sont implémentés directement dans le code. Cependant, nous avion mis en place une idée. En fait les boards étaient décrites dans deux fichiers. Ceci étaient présents dans un dossier "graph". Ils comportaient leurs descriptions et devaient respecter un format particulier.

La première ligne comportait la taille du plateau. Ensuite, on avait un ensemble de couples de lignes. Chaque couple pouvait être numéroté de zéro à n (avec n la taille). Le couple i représentait le sommet i du graphe. Pour chaque couple, la première ligne représentait les numéros des sommets voisins, la seconde représentait les tags associés. Il était même possible d'insérer un commentaire entre ces deux ';'. La fonction "board_from_file" permettait ceci. Cependant, le code mis en place était mis à mal dès qu'un fichier créé sous un système Windows devait être utilisé sous Unix. En effet, Windows insère un
r
n à la fin d'une ligne alors que Unix
n. Ceci empêchait le fichier de respecter le bon format. Ainsi, il a été préféré finalement d'implémenter les plateaux dans le code. Il serait tout à fait possible d'ailleurs d'étendre le programme à des graphes autres que ceux utilisés. Les fonctions sont d'ailleurs conçues pour fonctionner avec d'autres plateaux si nécessaire (seuls quelques modifications seraient à ajouter comme l'ajout de tags et les modifications de certaines fonctions). Nos tests finalement fonctionnent avec des plateaux de taille 4 ou 8. Même si la plupart des fonctions ont été testés avec des plateaux de taille 16, les solveurs ne pouvaient être testés que sur des graphes ayant une taille plus faibles.

FIGURE 2.2 – Graphe représentant la board de taille seize

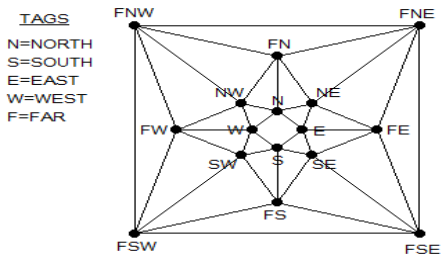
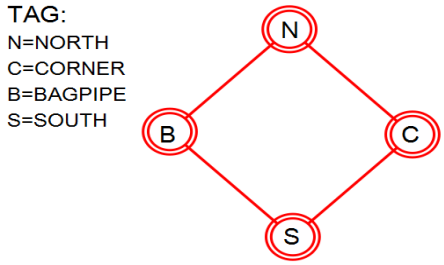


FIGURE 2.3 – Graphe représentant la board de taille quatre



Complexité

La complexité de la création d'une board est constante.

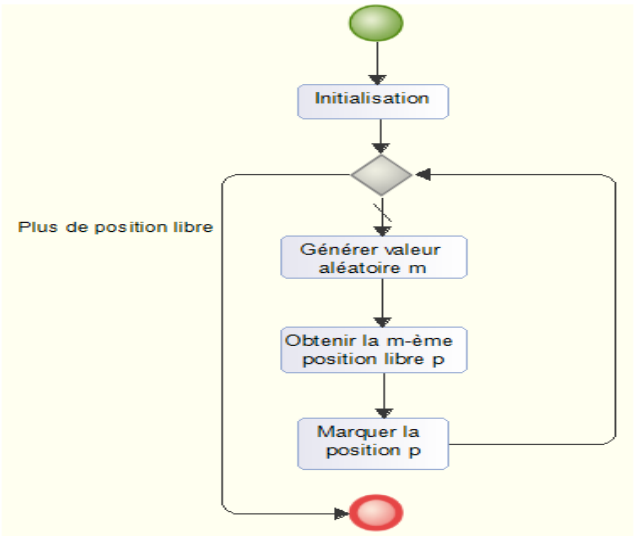
2.3.2 Les affectations

Fonctionnement

L'affectation des pélicans sur le plateau est générée aléatoirement. Ainsi, une fonction retournant un nombre aléatoire a été utilisée. Rand est une fonction bien connue pour ce type d'opération et déjà fournie dans la bibliothèque standard ; c'est celle que nous avons utilisée. Nous avions aussi besoin d'un élément pour marquer les positions du plateau. Il fallait pouvoir déterminer si une position était libre avant de l'affecter.

Pour cela, plusieurs solutions ont été envisagées. La première consistait à utiliser un tableau de booléen. Le problème est que le type booléen n'existe pas de base en C. On avait certes inclus la bibliothèque stdbool permettant d'importer ce type. Cependant, même si le code devenait alors plus lisible, les booléens eux-mêmes n'étaient pas stockés sur un bit mais un octet. En faisant ainsi, on aurait 7/8 de la mémoire réservée par ce tableau qui serait totalement inutile. C'est pourquoi, nous avons cherché d'autres solutions. Parmi ces solutions, on avait celle d'utiliser un entier qu'on traiterait bit à bit. Chaque bit de cet entier représentait alors une position. Cette fois-ci l'inconvénient était la taille limite de l'entier et par extension des plateaux qu'on pouvait utiliser. On a ainsi trouvé une solution à mi-chemin. Notre solution consiste à utiliser une structure qu'on peut utiliser comme un tableau de booléen. En fait, cette structure contient une adresse en mémoire. Cette adresse est le début d'une plage mémoire qu'on réserve à

FIGURE 2.4 – Fonctionnement globale – Affectations



l'initialisation de la structure avec une fonction dédiée. Cette structure qu'on a nommé custom_type permet ensuite au travers de différentes fonctions de traiter chacun des bits de la plage mémoire. Ainsi, en utilisant celle-ci on peut alors traiter n'importe quelle taille de plateau. Le i-ème bit de poids le plus faible représente la position i. Pour chaque pélican, nous générons un nombre aléatoire que nous nommons m. Ce nombre est compris entre 0 et le nombre de positions encore disponibles. Ensuite, nous récupérons la m-ème position encore disponible. Une fois toutes les positions prises, on retourne l'affectation.

Complexité

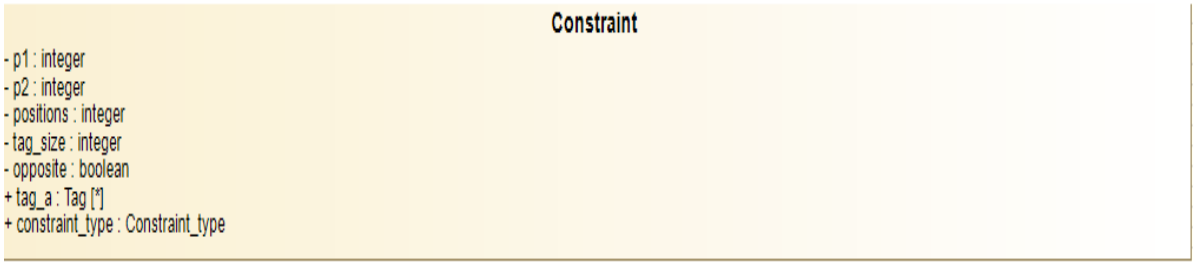
La génération des positions nécessite de définir une position pour chaque pélican. Une boucle était donc nécessaire. Cette boucle fait appel à une autre pour rechercher la m-ème position libre. Si on envisage le pire des cas où m serait généré systématiquement à sa valeur maximale (soit 15,14,13,...), on aurait alors la complexité suivante :

$\sum_{i=0}^{n-1} i$ où n est la taille de l'affectation

2.3.3 Génération des contraintes

Fonctionnement

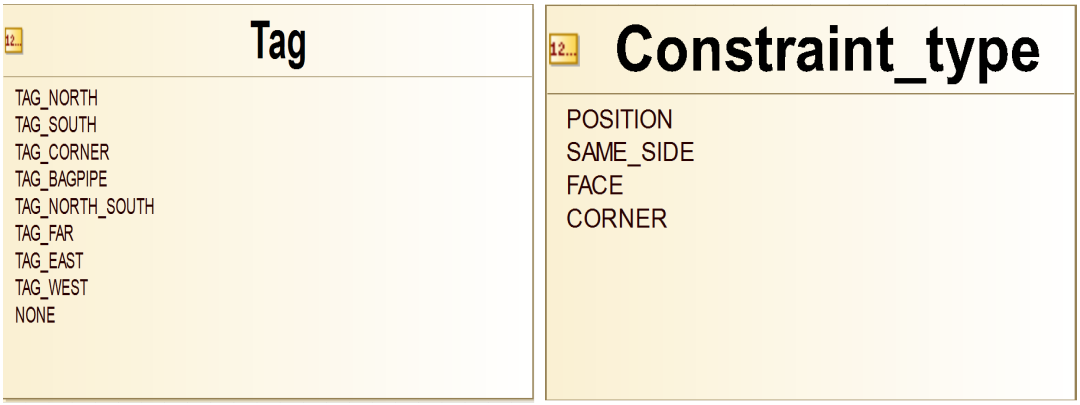
FIGURE 2.5 – Structure Constraint



Pour comprendre la génération des contraintes, il faut étudier la structure "constraint". Cette structure contient sept paramètres et de nombreuses fonctions lui sont consacrées. Les deux premiers paramètres, p1 et p2, représentent les couleurs des pélicans. Le paramètre "constraint_type" correspond au type de contrainte. Si la contrainte est de type POSITION (contrainte mono-pélican) alors la couleur du pélican 2 sera par défaut NO_COLOR. Le booléen opposite signifie, s'il est à true, que le pélican veut la négation

de la contrainte (cela sera utilisé lors de l'achèvement deux). "Positions" correspond aux positions qui respectent cette contrainte (idée qu'on a eu pour l'achèvement quatre). Le paramètre tag_a correspond à un tableau de tags. Ce dernier est NULL par défaut sauf s'il s'agit d'une contrainte de position auquel cas ce tableau correspond à l'ensemble des tags pouvant satisfaire le pélican. Un pélican voulant être au nord OU au sud aura les tags TAG_NORTH et TAG_SOUTH. A noter qu'ici le tableau de tags [A,B,C] signifie que le pélican souhaite A ou B ou C.
Pour créer une contrainte aléatoirement, la classe de contrainte (mono ou bi-pélican) est définie aléatoirement, puis s'il s'agit d'une contrainte bi-pélican, le type de contrainte est généré ainsi que le pélican associé "p2". Sinon, s'il s'agit d'une contrainte mono-pélican (donc de type position), le tableau de tag "tag_a" est généré. Tout est généré aléatoirement utilisant rand.

FIGURE 2.6 – Les énumérations Tag et Constraint_type



La figure ci-dessus représente notre choix de tags et de contraintes. Il serait possible de faire fonctionner le programme en y ajoutant des tags et de nouveaux types de contraintes. Seulement, cela nécessiterait de les ajouter à la fonction de génération des contraintes.

Complexité

Ces actions sont effectuées pour chaque pélican. La complexité dans tout les cas est de la taille de l'affectation, elle-même de la taille de la board. Nous avons donc :

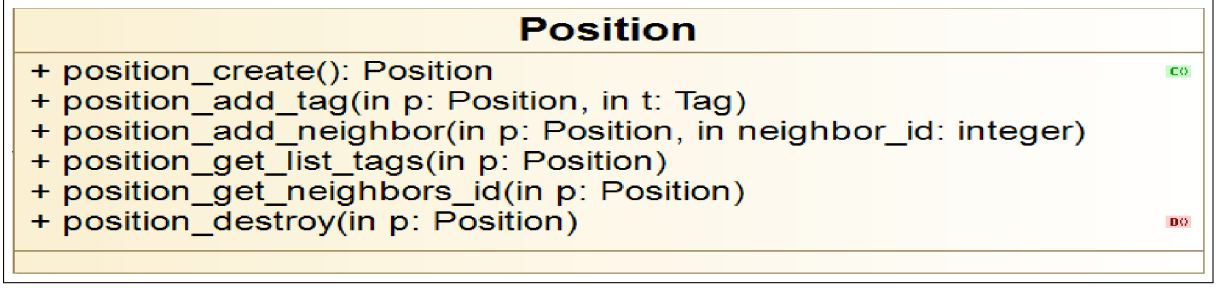
$\theta(n)$ où n est le nombre de pélicans.

2.3.4 Les distances

Fonctionnement

Pour calculer les distances, l'algorithme de Dijkstra a été utilisé. Ici, on considère que le poids d'une arrête est de un. A partir d'un point de départ, on marque chaque voisin et on recommence avec chacun des voisins déjà marqués jusqu'à arriver à la destination. La distance correspond au nombre de noeuds qu'il a fallu traiter pour arriver à destination. La liste des voisins est stockée dans la structure 'position'. Un tableau de positions étant présent dans la structure board, les positions sont créées à la génération de la board.

FIGURE 2.7 – Les fonctions dédiées à la structure Position



Complexité

Ici, le graphe est représenté par une liste d'adjacence, la complexité serait alors : $O((A+S)log(S))$ avec A le nombre d'arrêtes et S le nombre de sommets.

2.3.5 Has tag

Fonctionnement

La fonction has_tag a pour but de déterminer si une position possède ou non un tag. Une simple boucle dans laquelle chaque tag de la position est comparé au tag passé en paramètre permet de résoudre le problème. Puis, il est retourné vrai ou faux selon les résultats du test.

Complexité

La complexité est donc linéaire en le nombre de tags présents sur la position.

3 Achievement 1

3.1 Problématique

Le premier solveur qui a été implémenté est un solveur "bruteforce" qui a pour objectif d'explorer l'ensemble des affectations possibles des pélicans, puis de lister l'ensembles de ces affectations qui répondent à la liste des contraintes.

Dans un premier temps, nous allons présenter le solveur implémentée initialement puis nous verrons comment nous avons pu l'améliorer par la suite.

3.2 Le solveur

3.2.1 Algorithme

Algorithm 1: Algorithme du solveur bruteforce

```
1 Générer l'ensemble des arrangements possibles d'affectations de pelicans
2 l ← liste_vide()
3 while On a pas parcouru l'ensemble des arrangements do
4 |   Évaluer le score de l'affectation courante
5 |   if Affectation respecte toutes les contraintes then
6 |   |   Ajouter l'affectation à la liste l
7 |   end
8 end
9 Retourner la liste l des affectations satisfaisant les contraintes
```

On notera que l'algorithme est ici simplifié. Celui qui est implémenté dans notre programme donne un résultat (la meilleure affectation possible) même s'il n'existe aucune affectation vérifiant toutes les contraintes.

3.2.2 Complexité

Même si elle est correcte, il est immédiat de constater que cette solution n'est pas efficace. En effet l'ensemble des affectations correspond à l'ensemble des arrangements des entiers de 0 à n - 1 (n étant la taille du plateau de jeu), ce qui correspond à un ensemble à n ! éléments.

On a donc une complexité algorithmique en $\theta(n!)$ qui devient très vite limité. En effet pour une board de taille 10 on atteint déjà un nombre d'opérations de 3628800 et pour une taille de 15 on arrive à 1.3076744e+12.

3.2.3 Amélioration

Lors de l'achèvement 4, le solveur a pu être amélioré en utilisant un système de sur-approximation couplé à apply_constraint. C'est-à-dire que certaines contraintes limitent de base les emplacements disponibles pour le pélican associé. Par exemple si un pélican a la contrainte "être au SUD", il est inutile de tester les arrangements où ce pélican est sur une case ne possédant pas le tag TAG_SOUTH. Cela réduit drastiquement le nombre de permutations à explorer dans le solveur initial et améliore donc sa complexité.

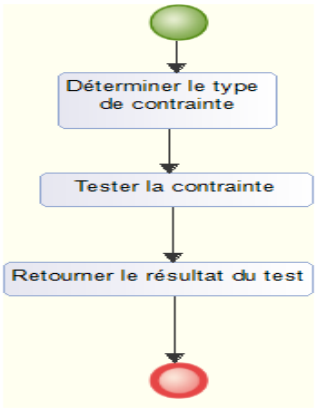
L'algorithme du solveur n'est pas modifié à proprement parler, c'est seulement l'ensemble des affectations parcourues qui est réduit. Le changement se situe donc lors de la génération de ces affectations où on prend seulement en considération les arrangements entre positions possibles (calculées préalablement).

3.3 Fonctionnement de apply constraint

Une fois tout les éléments générés, le fonctionnement de apply constraint est relativement simple.

On détermine d'abord le type de la contrainte. La première idée était qu'une fois le type déterminé d'appeler une fonction qui retournait, pour des positions données si la contrainte était respectée. Ainsi, on avait une fonction différente par type de contrainte. La première constraint_position détermine si une position contient un des tags entrés en paramètre (ceux de la contrainte du pélican). Si tel est le cas, elle retourne vrai. La fonction constraint_face détermine si un des pélicans de la contrainte bi-pélican 'FACE' est au nord et l'autre au sud (ou est et ouest). La fonction destinée à vérifier si deux pélicans sont du même

FIGURE 3.1 – Schéma du fonctionnement – Apply_constraint



côté (constraint_side) teste si les deux
possèdent les mêmes tags (TAG_NORTH, TAG_SOUTH, TAG_EAST, TAG_WEST). Enfin, la fonction constraint_corner teste si les positions des deux pélicans possèdent le tag TAG_CORNER et s'ils sont aussi voisin l'un de l'autre. Chacune de ces fonctions est appelée par apply_constraint pour vérifier si une contrainte est respectée. Néanmoins, avec l'achievement quatre, ces fonctions ne sont plus utilisées au sein de apply_constraint dans la mesure où les positions acceptées sont pré-calculées et rattachées à la contrainte. On a juste alors à lire la valeur d'un bit pour vérifier si la contrainte est applicable ou non.

4 Achievement 2

4.1 Problématiques

Dans cette partie, apply_constraint devait être modifiée afin de lui faire traiter l'ensemble des contraintes tout en prenant en compte qu'à présent il puisse y-avoir des dépendances entre les contraintes. Ainsi, le pélican A peut vouloir la même contrainte du pélican B ou son opposé. Cela posait la question de ce qu'est l'opposé d'une contrainte. L'opposé d'être au Nord est-il être au Sud ou ne pas être au Sud ? De même, que faire lorsque le hasard a choisi que le pélican A veuille la même contrainte que le pélican B et inversement ? Cela introduirait forcément des cycles de tailles variables.

4.2 Choix

Pour se rapprocher de la logique propositionnelle, le choix a été fait de définir l'inverse de "vouloir une contrainte A" était de "vouloir tout sauf A". Ainsi, un pélican qui veut l'opposer de "être au Nord" accepte d'être placé n'importe où sauf au Nord. Pour faire face à ce nouveau type de contrainte, la fonction de génération des contraintes ("generation_constraint_array") a été modifiée. Cependant, bien qu'à la sortie de cette fonction, un pélican puisse vouloir l'opposé d'un autre, il ne lui est pas possible de vouloir l'opposé d'une contrainte. Ainsi, le pélican A peut vouloir l'opposé du pélican B qui possède une contraint C. Mais, il ne peut vouloir l'opposé de la contrainte C de base. Ce n'est qu'au moment de traiter les dépendances que lui sera assigné l'opposé de C. Un autre choix s'est posé lors d'un cycle. On a fait le choix de considérer l'affectation invalide lorsqu'un cycle se présentait. Il aurait aussi été possible de retirer toutes contraintes aux pélicans à l'origine du cycle (cela est d'ailleurs fait pour le solveur z3). Aussi, deux éléments ont été rajoutés à l'énumération constraint_type à savoir SAME_CONSTRAINT et OPPOSITE_CONSTRAINT. Il s'agit là de deux contraintes bi-pingouin. Si le pélican A a la contrainte SAME_CONSTRAINT avec en paramètre le pélican B alors il lui sera associé après traitement la contrainte du pélican B. S'il avait eu la contrainte OPPOSITE_CONSTRAINT, alors il aurait reçu la même contrainte sauf que le booléen 'opposite' aurait été mis à true pour signifier que le pélican souhaite l'opposé. La première version de apply_constraint a été conservée dans le code, la nouvelle étant nommée apply_constraint_rec.

4.3 apply_constraint_rec

4.3.1 Fonctionnement

La fonction utilise la récursivité pour pouvoir traiter l'ensemble de l'affectation. Son fonctionnement est simple. Comme pour la première version, on teste le type de contrainte. Si la contrainte est de type SAME_CONSTRAINT ou OPPOSITE_CONSTRAINT, on fait appel à la fonction treat_dependence pour retirer les dépendances. Si, on ne peut retirer les dépendances, alors l'affectation sera considérée comme invalide et la fonction retournera false. Si, les dépendances ont été traitées avec succès alors apply_constraint_rec est rappelée pour retraiter cette contrainte. En l'absence de dépendances, les mêmes actions que la première version sont effectuées. Elle stoppe et retourne false si une contrainte non applicable est trouvée. En l'absence de contrainte invalide, true est retourné.

4.3.2 Complexité

La complexité de cette fonction est linéaire puisqu'on traite chaque contrainte. Bien qu'on fasse pour chaque dépendance appelle à treat_dependence de complexité (dans le pire des cas) $\theta(n)$, les dépendances de chacune des contraintes ne sont traitées au plus qu'une fois.

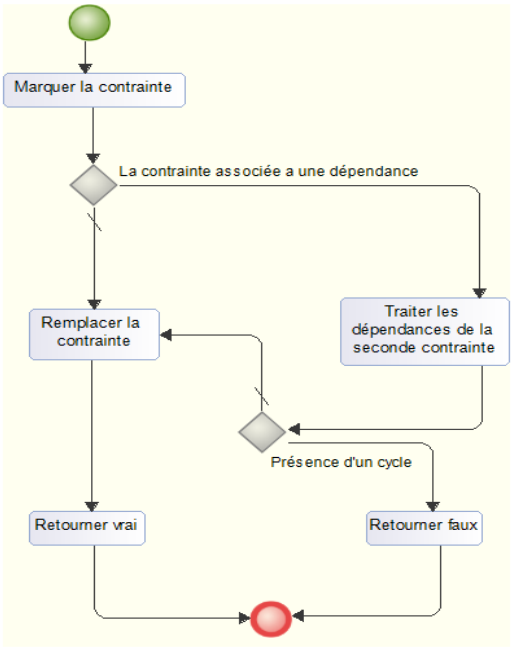
4.4 Traitement des dépendances

4.4.1 Fonctionnement

La fonction treat_dependance a pour but de régler le problème des dépendances et des cycles. Pour cela, elle prend en paramètre la contrainte qu'on souhaite traiter mais aussi l'ensemble des contraintes. En effet, pour résoudre une dépendance, on peut être amené à devoir résoudre d'autres dépendances. Ainsi, pour cela cette fonction est récursive. En fait, on commence par marquer la contrainte en cours c1 puis on vérifie si la contrainte dont elle dépend c2 possède elle aussi une dépendance. Si ce n'est pas le cas, alors les valeurs de c2 sont affectées à c1. Le type de la contrainte c1 est remplacé par le type de la contrainte c2. Le booléen opposite est mis à true ou false selon si le pélican voulait l'opposé ou la même contrainte. Ceci est le cas de figure le plus simple. Cependant, si la contrainte c2 est une contrainte bi-pélican (autre que SAME_CONSTRAINT et OPPOSITE_CONSTRAINT) avec le pélican ayant la contrainte c1, alors on se retrouve dans une boucle.

Nous résolvons ce type de boucle en générant aléatoirement un nouveau pélican pour c1. c1 prend tout de même les mêmes attributs que c2 mais sera lié avec un autre pingouin. Le second problème concerne les dépendances plus importantes. Par exemple, le pélican 1 veut l'opposé du pélican 2 qui veut l'opposé du pélican 3. Dans ce cas de figure, si le pélican 3 a une contrainte c alors le pélican 2 aura la négation de c et comme le pélican 1 veut l'opposé du pélican 2, alors il aura la négation de la négation de c soit c. Ainsi, pour ce type de cas de figure pour traiter une contrainte c1, on traite la contrainte c2 dont elle dépend avec un appel récursif. On fait de même pour c2. L'autre problème est qu'il est ainsi possible d'avoir des cycles plus importants du type le pélican p1 veut la même chose que p2 qui veut la même chose que p3 et ainsi de suite jusqu'à pn voulant la même chose que p1 . Dans ce cas là, on vérifie à chaque traitement si la contrainte associée n'a pas déjà été traitée. Si c'est le cas, cela signifie qu'on est dans une boucle. Cette boucle est différente de la première. Dans ce cas là, si une telle boucle est repérée, on considère l'affectation comme invalide. La fonction retourne donc false. En revanche, si les dépendances ont pu être traitées alors true est retourné.

FIGURE 4.1 – treat_dependence



4.4.2 Complexité

La complexité dans le pire des cas est linéaire. Cela correspond au cas où il existerait une chaîne de la taille de l'affectation. Cependant, apply_constraint_rec reste linéaire car dans ce cas là tout les problèmes de dépendance seraient traités lors du premier appel de treat_dependence ce qui sous-entend que la fonction ne serait appelée qu'une seule fois. Au final, si n est la taille de l'affectation, il y-aurait n passages dans treat_dependence et n+1 passages dans apply_constraint_rec donc une complexité de $\theta(n)$

4.5 Difficultés

La véritable difficulté a été de trouver un algorithme pour régler le problème des cycles. Pour le reste, le code devenant de plus en plus touffu, il a fallu le réorganiser. Cela sous-entend de rajouter des règles de nommage et un meilleur découpage en fonctions. La convention de nommage concernait surtout les structures. Il a donc été imposé que chaque nom de structure se termine par _s et que les typedef sur pointeur de structure se termine par _t. Chacun de ces typedef devait aussi avoir un nom reprenant celui de la structure associée. Étant donné qu'il s'agit de typedef sur des pointeurs, on manipulait donc à présent uniquement des pointeurs à l'instar du Java. On a aussi légèrement modifié la première version de apply_constraint pour que les dépendances puissent aussi être traitées.

5 Achievement 3

5.1 Problématiques

Ici, nous devions utiliser le logiciel z3 afin de déterminer les affectations valides. Cela nécessitait de prendre connaissance de ce logiciel, de son fonctionnement. Z3 nécessite d'entrer une formule définie selon un format précis. L'entrée représente en définitive une formule logique. Le problème ici était donc de transformer apply_constraint afin qu'il génère une formule logique. L'autre problème était de lancer Z3 et récupérer sa réponse. Z3 est un logiciel s'exécutant séparément et donc dans un espace mémoire séparé. Ainsi, difficile d'avoir accès à sa réponse. Ceci a été la source de nombreuses recherches.

5.2 Choix

Générer le code s'est fait au travers de diverses fonctions qui seront détaillées dans les sections suivantes. Elles génèrent la formule et l'écrivent dans un fichier. Pour cela, il a suffit de reprendre le code qui avait servi précédemment dans le "mini projet" de la matière "Logique et preuve" et de l'adapter au C. L'écriture et la lecture de fichiers en C étaient déjà connues, en revanche exécuter z3 a posé beaucoup plus de problèmes. Plusieurs solutions ont été envisagées. Certaines sont un peu alambiquées, d'autres plus simples mais non fonctionnelles. Au final c'est une solution à la fois simple et fonctionnelle qui a été retenue. Ainsi, une première idée avait été tentée en utilisant la primitive "syteme". Cette primitive est utilisable aussi bien sur Unix que sur Windows et permet de lancer une ligne de commande depuis un programme. Cela permettait bien d'exécuter z3 mais c'est oublier qu'il ne s'exécutait pas dans la même zone mémoire. Ainsi, il était impossible de récupérer sa réponse. L'idée suivante visait à lancer z3 avec une redirection vers un fichier pour ensuite le lire. Cela était non seulement alambiqué mais en plus posait le problème de la durée. Il n'est pas possible de savoir à l'avance le temps que mettra z3 à résoudre le problème. Ainsi, on ne peut pas non plus savoir quand lire la réponse. Ensuite vint une autre idée. Pourquoi ne pas récupérer le code source de z3 pour l'intégrer au projet. Cela aurait été possible mais nécessitait du temps pour l'étudier. On s'est alors porté sur les bibliothèques disponibles en C. Ainsi, stdlib propose un ensemble de fonctions "exec." pour exécuter un programme. De même, il est aussi possible avec certaines fonctions de modifier et d'accéder à des variables d'environnement. Ainsi, l'idée était de créer une variable d'environnement puis de lancer z3 avec une redirection vers la variable créée et la lire. Non seulement la redirection ne fonctionnait pas et cela ne résolvait pas le problème de la durée d'exécution. On devait l'exécuter au sein de la

même zone mémoire que le logiciel. Cela devait être un sous-processus. C'est ainsi que nos recherches se sont portées sur les processus. Cela a débouché sur la découverte de la fonction popen pour créer un processus. Ce processus est un sous processus s'exécutant dans un sous-ensemble de la mémoire du programme et donc accessible par ce dernier. Cette fonction retourne un FILE*, c'est à dire l'emplacement mémoire de la sortie. Ceci se lit comme un fichier. On avait donc enfin la solution pour lancer z3 et récupérer sa réponse. Ceci était la véritable difficulté technique de cette partie, une fois cette solution en main on pouvait se lancer dans l'écriture des fonctions.

5.3 Génération du script

5.3.1 Fonctionnement

La génération du script se passe en trois principales étapes. Une première fonction init_z3_formula initialise la formule avec la déclaration des booléens utilisés et les règles de bases (une position par pélican, un pélican par position). Il s'agit de la transcription de la formule suivante :

Chaque pion est sur au moins une position :
$\boxed{\wedge_{i=1}^{n}(\vee_{j=0}^{n-1}P_{i,j})}$ où n est la taille du plateau et donc de l'affectation et $P_{i,j}$ signifiant le pélican i (i commençant à 1) est placé à la position j.

Chaque position est occupée par au plus un pion :
$\boxed{\wedge_{j=0}^{n-1}\wedge_{i=1}^{n}(P_{i,j} \Longrightarrow \wedge_{k=1,k\neq i}^{n}\neg P_{k,j})}$

Une seconde fonction transcrit la contrainte en cours de traitement. En fait, il existe deux fonctions une pour les contraintes mono-pingouin, une autre pour les contraintes bi-pingouin. La première "generate_z3_position_constraints" calcule les positions possibles pour l'ensemble de tags lié à la contrainte qu'on veut formuler. Ainsi, elle transcrit un ou logique sur toutes les positions comportant ce tag. Par exemple : le pélican i veut être au nord et les positions avec le tag TAG_NORTH sont les 3 et 4 alors on aura la transcription de la formule suivante :

$\boxed{P_{1,3} \vee P_{1,4}}$

S'il s'agit d'une contrainte bi-pingouin, alors on utilise le fait que lors de la génération des contraintes, on calcule l'ensemble des couples de positions possibles pour chaque

contrainte pour transcrire dans la fonction "generate_z3_fcs_constraints" la formule suivante :

$\boxed{\wedge_{k=0}^{n-1}(P_{j,k} \Longrightarrow \vee_{l=0,(k,l)\in R}^{n-1} P_{i,l})}$ avec i et j les deux pélicans concernés qui sont passé en paramètre de la fonction au travers de la contrainte.

Une fois les contraintes générées il ne reste qu'à définir l'affectation des pélicans sur le plateau. Pour cela, on transcrit avec un et logique pour chaque placement. Il est possible de choisir de ne pas représenter le placement des pélicans selon si l'on cherche à tester la validité d'une affectation ou s'il existe une affectation possible.

5.3.2 Complexités

Leurs complexités sont polynomiales. L'initialisation est dans tout les cas en $\theta(n^3)$, la transcription des contraintes bi-pélicans en $\theta(n^2)$ et les autres sont linéaires avec n pour la taille du plateau. Ces fonctions étant appelées de façon séquentielle, la complexité de la génération du fichier destiné à z3 est de $\theta(n^3)$.

5.4 apply_constraint_z3

5.4.1 Fonctionnement

Cette nouvelle version de apply_constraint fait appel à quatre fonctions pour quatres principales étapes. La première est de générer le script. Cela est fait par la fonction generate_z3_script qui une fois la formule initialisée parcours l'ensemble des contraintes et appelle les fonctions présentées ci-dessus. Une fois le fichier écrit, la fonction "get_z3_output" utilise popen pour exécuter z3 et récupère le résultat retourné sous la forme d'une chaîne de caractères. Il suffit ensuite de vérifier s'il s'agit d'un "sat" ou d'un "unsat".

5.4.2 Complexités

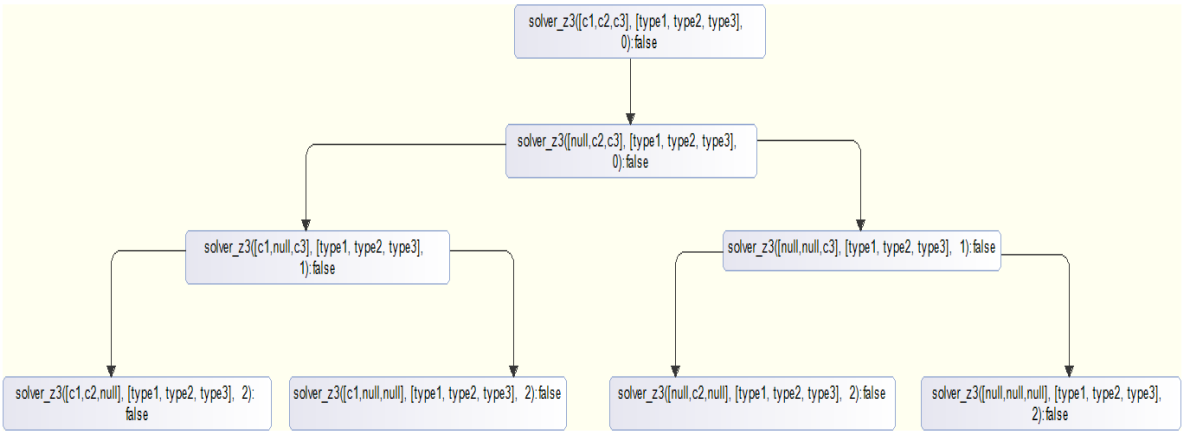
La complexité dépend de celle de la génération du script. La complexité est donc $\theta(n^3)$.

5.5 Le solveur Z3

5.5.1 Fonctionnement

Le but du solveur est de trouver au moins une affectation valide pour un ensemble de contrainte quitte à retirer quelques contraintes. Ainsi, il va tout d'abord tester avec toutes les contraintes puis retirer des contraintes, retester et ainsi de suite jusqu'à trouver une solution. Si on considère C l'ensemble de contraintes c1,c2,c3, ... cn alors le solveur va chercher une solution en retirant à chaque test une combinaison de contrainte jusqu'à retirer l'ensemble des contraintes. Pour retirer une contrainte, on lui affecte le type NO_CONSTRAINT. Si après avoir tout testé, il n'y-a toujours aucune solution alors il retourne NULL sinon il retourne l'affectation valide. Ce qui suite correspond à son arbre d'appel. Pour faire cela, nous utilisons la récursivité qui va générer au final une sorte d'arborescence. A chacun des noeuds, il est testé une combinaison de contrainte. Ce qui suit est une illustration simplifiée de l'arbre d'appel de la fonction. On n'y représente que trois contraintes. Dans cette représentation, les contraintes sont remplacées par null lorsque leur type est mis à NO_CONSTRAINT. L'ensemble des contraintes correspond au premier paramètre. Le second est un tableau dont la seule utilité est de stocker les anciens types afin de pouvoir les réassigner. Enfin, le dernier paramètre correspond à un index.

FIGURE 5.1 – Arbre d'appel du solveur z3



On teste ainsi chacune des combinaisons. Une fois une combinaison trouvée, une

fonction "get_z3_affect" utilisée dans apply_constraint_z3 s'occupe de traiter la sortie de z3 pour en retirer une affectation valide.

5.5.2 Complexité

Le gros problème avec ce solveur est sa complexité dans le pire des cas. Sa complexité est au moins exponentielle. En effet, même si on ne prend pas en compte la génération du script à chaque passage, le simple fait de tester toutes les combinaisons est exponentiel. En fait, cela reviendrait à prendre les parties de C or, il y-aurait 2^n éléments. Ainsi, il serait impossible logiquement d'avoir une complexité inférieure à celle-ci. Nous n'avons donc pas trouvé de meilleure solution.

5.5.3 Terminaison

La condition d'arrêt réside dans une condition qui vérifie si l'indice en paramètre est supérieur à la dernière case du plateau soit égal à la taille du plateau. Chaque appel récursif voit l'indice incrémenté de un, ainsi on pourrait considérer cet algorithme comme un ensemble de boucles imbriquées ayant chacune pour terminaison cette condition. Pour démontrer la terminaison de l'algorithme, il faut alors démontrer que l'invariant de boucle board_size-indice est décroissant. Ceci est une fonction de type $f(x) = constante - x$ avec $x \in \{0, 1, 2, .., constante\}$. La fonction est bornée. Sa dérivée est négative et par conséquent la fonction est décroissante. Or, toute fonction décroissante et bornée converge donc l'algorithme termine.

5.5.4 Tests

On test ce solveur ainsi que le précédent grâce à un jeu de test. Ce jeu de test est constitué de six combinaisons de contraintes dont trois valides et trois valides à condition de retirer quelques contraintes. On test consécutivement le solveur Z3 qui va chercher une solution en retirant éventuellement des contraintes. Puis, sur l'ensemble de contraintes modifiées par le solveur Z3, on exécute le premier solveur. Ce dernier retourne un certain nombre de solutions, il suffit ensuite de vérifier si celle de Z3 y est inclus. Si tel est le cas, le test réussis sinon il s'agit d'un échec.

5.6 Difficultés

Cette partie était la plus difficile de toutes. Il a fallu beaucoup chercher pour trouver un moyen d'utiliser z3 depuis l'application. Cependant, c'est surtout au moment de

comparer les résultats entre les trois différentes versions de apply_constraint que cela a posé des difficultés. Les résultats n'étant pas les mêmes, cela a mis en lumière des erreurs qu'il a fallu corriger. Souvent peu graves mais nombreuses et parfois longues à trouver puisqu'elles n'engendraient pas forcément d'erreurs ou de problèmes de compilation.

6 Achievement 4

6.1 Résumé

L'achèvement quatre consiste en une sur-approximation d'une solution. Cela consistait à ne considérer uniquement les affectations valides pour chaque contrainte. Les versions présentées dans ce rapport sont les versions finales, une fois tout les achèvements terminés. Dans les versions initiales de apply_constraint et apply_constraint_rec, on recherchait systématiquement la présence de tag pour déterminer si une position était valide ou non. Le nombre de tags étant fixe, cela était constant. Seulement, que se passerait-il si l'on souhaitait implémenter des plateaux plus grand ? Le nombre de tags nécessaires augmenterait et il serait alors nécessaire d'enlever cette limite. Cela augmenterait significativement les complexités étant donné que la recherche de tag se ferait en temps linéaire. Ainsi, apply_constraint serait linéaire, apply_constraint_rec quadratique. Nous avons pour cela rajouté la fonction compute_available_positions qui génère les positions possibles pour chaque contrainte affectée. Ceci est fait à partir de deux tableaux : pos_a et pos_relation_a. Le premier contient l'ensemble des positions possibles pour chaque tag. Le second lui contient l'ensemble des couples de positions pour chaque contrainte bi-pingouin. Ceci sont générés par les fonctions compute_position_a pour le premier et compute_relation_a pour le second. Une fois les positions possibles calculées, elles sont affectées aux contraintes dans le champ "positions". Par la suite, il suffit donc de vérifier si le pélican est bien positionné. Ceci se limite alors à une lecture de bit et se fait donc en temps constant.

6.2 Custom_type

Le stockage des positions valides se fait par une structure particulière. En effet, bien qu'il était possible de stocker cela dans un tableau de booléen, une autre solution a été envisagée. Le problème était que le type booléen de la bibliothèque stdbool réserve un octet par booléen. Cela gaspille 7/8 de la mémoire. Ainsi, le "custom_type" peut être vu comme un tableau de bit. Concrètement, il s'agit d'une structure comportant un tableau sans type. Une fonction permet d'initialiser la structure avec une taille personnalisée (d'où le custom du nom) dont chaque bit est initialisé à zéro. Puis, plusieurs fonctions permettent d'accéder et manipuler les bits. Ainsi, il n'y-a plus de gaspillage de mémoire. Il est à noter que les fonctions utilisées sont toutes de complexité constante exceptée celle qui permet de copier deux éléments de ce type (linéaire).

6.3 Pré-calcule des positions par tag

6.3.1 Fonctionnement

La fonction compute_position_a permet de créer un tableau. Les indices de ce tableau correspondent à un tag. Ainsi, l'indice 0 correspond au tag TAG_NORTH, l'indice un à TAG_SOUTH, etc ... . Comme déjà précisé, tag est une énumération et donc à chaque tag correspond un entier. Le choix de ces entiers a été fait de manière à les faire correspondre aux indices du tableau. Il est donc possible d'écrire tab[TAG_NORTH] ce qui rend plus explicite la lecture du code. Notre première idée était d'en faire un tableau d'entiers. Chaque bit de cet entier correspondait aux positionnements possibles. Cela avait l'inconvénient de limiter la taille des plateaux qu'on pouvait utiliser, cependant, ça permettait en temps constant de se positionner sur le i-ème bit et vérifier si la position i était ou non possible pour ce tag. Ainsi, pour le tag TAG_NORTH sur une board de taille seize, on aurait 0000001100010011 par exemple. Si on se positionne sur le bit à l'indice quatre, on voit qu'il est à un et donc que la position quatre est possible. Si on veut vérifier qu'un pélican ayant cette contrainte peut bien être placé, on a juste à se positionner sur le bit représentant la position du pélican et on voit de suite en temps constant s'il peut ou non être positionné. Cette méthode nécessite de tester pour chaque tag les positions possibles. Ceci est coûteux en temps mais effectué qu'une seule fois. Le seul problème était la limitation inhérente au type choisi. C'est pourquoi, il a été créé un type particulier, celui présenté ci-dessus.

6.3.2 Complexité

La complexité est relativement importante. On teste chaque combinaison tag/position et on en déduit l'ensemble des positions possibles. Le test est fait avec la fonction constraint_position qui était initialement utilisée par la première version de apply_constraint. En considérant constantes les fonction utilisées, on a alors une complexité de $\theta(n)$. De plus, ceci ne concerne que les contraintes mono-pingouin. La fonction suivante s'occupe des contraintes bi-pinguin.

6.4 Pré-calcule des couples de positions par contrainte bi-pinguin

6.4.1 Fonctionnement

La fonction compute_relation_a a pour objectif de calculer pour chaque contrainte bi-pingouin, les couples de positions la respectant. Ainsi, elle retourne un tableau à deux dimensions. La première correspond au type de contrainte (FACE, SAME_SIDE, CORNER). La seconde est les positions possibles. Ceci consiste aussi en un tableau de "custom_type" où chaque bit est une position. Cela a permis aussi de ne plus avoir à le faire systématiquement lors de la génération des scripts z3 diminuant par la même leur complexité.

6.4.2 Complexité

Pour obtenir cet ensemble, on doit tester tout les types de contraintes puis pour chacun, l'ensemble des couples de positions. On teste chaque combinaison avec les fonctions présentées à l'achèvement un qui sont chacune en temps constant (si on considère une limite fixe aux tags). Ainsi, on obtient une complexité $\theta(n^2)$. Une complexité importante mais qui permet par la suite de tester les contraintes bi-pingouin en temps constant même sans limites sur le nombre de tags ouvrant la voie à des plateaux plus grands.

6.5 Difficultés

Le code permettait déjà ce type de modifications. Ainsi, il a simplement fallu ajouter ces trois fonctions. La génération des tableaux utilise des fonctions déjà définies. Ainsi, peu de modifications ont été apportées par cet achèvement. Ceci nous a aussi amené à l'idée suivante ...

7 Achievement 5 : backtracking ?

7.1 Backtracking

Après avoir amélioré nos précédents solveurs lors de l'achievement 4, il nous est venu l'idée de penser le problème de manière différente. En effet au lieu de tester toutes les permutations possibles (ou un nombre sensiblement réduit), pourquoi ne pas procéder par tâtonnement ?

Concrètement, l'idée de l'algorithme est de conserver l'atout de l'achievement 4, c'est-à-dire de mémoriser pour chaque contrainte l'ensemble des positions disponibles (dans un custom type ou un tableau de booléens).

Ensuite, il faut partir d'une affectation où aucun pélican n'est positionné (valeur à -1), puis lancer un algorithme récursif. Celui-ci va placer un premier pélican p1 (celui qui a le moins de positions disponibles). On peut alors mettre à jour les positions possibles des autres pélicans (par exemple ceux qui veulent être en face de p1, ou ceux qui voulaient être à la place de p1). On continue à placer les pélicans comme cela (on rappelle la fonction récursivement) jusqu'à avoir toutes les contraintes validées (jackpot) ou sinon atteindre une condition d'arrêt.

Les deux conditions d'arrêt sont :

— On ne pourra pas dépasser le score maximum (ie. le nombre de contraintes satisfaites obtenues avec une affectation ultérieure). Cela se vérifie grâce à une fonction que nous appellerons score_maximum_potentiel() vérifiant le nombre de contraintes pouvant être encore validées (ie. où il reste au moins une position à 1 dans le tableau des positions possibles)

— On est dans une impasse, c'est-à-dire qu'on ne peut plus satisfaire de contrainte valide. Si on est arrivés jusqu'ici, cela veut dire que l'affectation courante devient la meilleure découverte jusqu'à maintenant (on la sauvegarde)

Dans les deux cas précédents, on effectue un "pas en arrière" (ie. on annule le positionnement qu'on vient de faire, cela revient à mettre la valeur -1 dans le tableau d'affectation sur le pélican qu'on vient de positionner), puis on place le pélican courant à la position suivante dans la liste des positions disponibles et ainsi de suite jusqu'à avoir parcouru l'ensemble des positions disponibles puis on effectue un "pas en arrière" etc. On peut finalement récupérer dans le tableau global associé l'affectation ayant réalisé la meilleure performance.

L'avantage de cette approche est que l'on ne gaspille pas de temps à explorer des chemins qui sont voués à l'échec. En effet le fait de calculer le score_maximum_potentiel()

permet d'éliminer d'office (ou du moins rapidement) des affectations non prometteuses et passer à une suivante pouvant égaler ou dépasser le score maximum actuel.

Note : cet algorithme n'a pas été implémenté par manque de temps, mais il nous a semblé pertinent de le présenter dans le rapport (en guise d'ouverture).

8 Conclusion

La solution a probablement ses imperfections. Ainsi, il existe peut être des solutions qui pourraient être simplifiées ou améliorées. Les idées de l'achèvement cinq n'ont malheureusement pas été implémentées. Quoiqu'il en soit, le projet a permis de se familiariser un peu plus avec la programmation mais aussi d'échanger. Cela a permis l'échange de connaissances et de bonnes pratiques. Cela a aussi permis de se familiariser avec les calculs de complexité et prendre en compte cet aspect lors de la conception du livrable. Il a été fait le maximum pour que le code soit modifiable facilement par une tierce personne. En revanche, en l'état tout type de board ne peut être implémenté. Une amélioration qui pourrait être apportée serait une généralisation ouvrant à n'importe quelle board et peut être même à la génération aléatoire d'une board. De même, ceci qui pourrait permettre de toucher d'autres thèmes. Le "pelican cove" est un jeu de placement mais la vie quotidienne regorge de problèmes de ce type. Par exemple, on pourrait extrapoler cela avec la gestion des salles de cours. Ici, les pélicans sont les enseignants et les élèves. Les contraintes seraient le planning, le besoin matériel (projecteur, ordinateurs), la taille de la salle. Le plateau serait alors les salles. On pourrait même envisager un plateau à deux dimensions (temps, espace). Selon l'heure (temps), certaines salles sont disponibles d'autres non (espace).
En conclusion, ce projet a été très utile pour expérimenter les connaissances mais aussi pour se familiariser avec le travail en équipe. Enfin, cela permettra aussi d'appréhender d'autres problèmes de ce type en se rappelant des "facetious pelican".