

# Projet S4

## Le bilan

# TABLE DES MATIÈRES

Table des matières.....	2
Travail rendu.....	3
1. Présentation	3
2. Objectifs	3
3. Contenu	3
a) « Noyau » de calcul	3
b) Application	3
c) Joueurs non-humains (IA)	3
d) Compilation séparée	4
e) Tests automatisés	4
f) Documentation	4
Algorithmie et complexité.....	5
1. Complexité des types de génériques	5
2. Complexité des types spécifiques	5
3. Arbre Minimax	5
a) Algorithmie	5
b) Observations	6
c) Complexité de la structure Minimax	6
4. Détection de ponts et prise de décision	8
a) Algorithmie	8
b) Observations	10
c) Complexité de l'algorithme de décision	10
Complexité sans les ponts	11
Complexité avec les ponts	12
d) Conclusion	12
Organisation et déroulement.....	14
1. L'équipe	14
2. Organisation	14
3. Répartition du travail	15
Bilan.....	16
1. Respect de la consigne	16
2. Difficultés rencontrées et solutions trouvées	16
a) Travailler en équipe	16
b) Manipuler des données variées	17
c) Gérer la complexité	17
d) Gérer la mémoire	17
3. Ce que nous retenons de ce travail	18

# TRAVAIL RENDU

Pour guider le correcteur :-)

## 1. Présentation

Le programme que nous avons développé, et nommé « *cretinhex* », est un logiciel permettant de jouer au fameux jeu *hex*, inventé par (et pour) les mathématiciens.

## 2. Objectifs

Ce logiciel souhaite satisfaire ces objectifs :

- Offrir tous les services demandés dans la consigne disponible sur Moodle,
- Satisfaire les attentes de notre tuteur et correcteur,
- Respecter les « engagements » pris dans le dossier de spécification
- Démontrer notre capacité à programmer sainement dans différents langages

... et il tient ses promesses !

## 3. Contenu

### a) « Noyau » de calcul

Développé en C, il permet la manipulation des parties (placement des pions, détermination du vainqueur, etc.)

- `c/cretinlib/` contient les types **génériques** de base (LDC, Arbre rouge/noir, Graphe)
- `c/hex/` contient les types **spécifiques** à Hex (Joueur, Damier, GrapheHex, Partie, Arbre Minimax)

### b) Application

Développée en Java, elle permet de jouer à *hex* en utilisant le noyau de calcul :

- Deux classes Java (Partie et PartieJNI) interagissent, grâce à JNI, avec le noyau
- Une interface en ligne de commande, dite IHMConsole, permet de jouer en ssh sur azteca, par exemple
- Une interface graphique, dite IHMRasta, offre toutes les options de jeu, à la souris

### c) Joueurs non-humains (IA)

Développés en C, et utilisés par l'application grâce à JNI, ces joueurs non humains sont :

- RandomBot, qui joue au hasard
- GotooneBot, qui utilise un arbre Minimax
- BridgeBot, qui détecte les ponts pour prendre une décision

#### d) Compilation séparée

Des makefiles, dans chaque dossier, s'appellent récursivement pour compiler les .o, .so, .java, pour créer l'exécutable, mais aussi pour générer la documentation doxygen, les headers JNI, lancer les séries de tests, etc.

#### e) Tests automatisés

Le dossier `c/tests/` contient, pour chaque module testé, un script de tests (`script_tests_XXX.sh`), et un mini-programme en C (`main_XXX.c`).

On lance les tests le plus simplement avec le makefile de ce dossier : `make tests` ou `make memory_tests`

#### f) Documentation

Le code est commenté en utilisant *doxygen*, la documentation complète (html) est générée avec la commande `make doc`. Elle se trouve dans le dossier `doc/`.

# ALGORITHMIE ET COMPLEXITÉ

## 1. Complexité des types de génériques

Action	Création	Insertion	Recherche	Modif	Fusion	Fusion sans doublon	Filtre
<b>LDC</b>	$\Theta(1)$	$\Theta(n)$ $\Theta(1)$ en fin	$\Theta(n)$	$\Theta(n)$	$\Theta(m)$	$\Theta(m \times n)$	$\Theta(m \times n)$
<b>ARN</b>	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(m \times \log n)$	$\Theta(m \times \log n)$	$\Theta(m \times \log n)$
<b>Graphe</b>	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$			

$n$  désigne le nombre d'élément contenu dans le type

$m$  désigne, pour la fusion, le nombre d'élément à fusionner

## 2. Complexité des types spécifiques

Action	Création (damier vide)	Création ( $m$ pions placés)	Modification d'une case	Gagnant ?
<b>Damier</b>	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	
<b>Historique</b>	$\Theta(1)$	$\Theta(m)$	$\Theta(1)$	
<b>GrapheHex</b>	$\Theta(n)$	$\Theta(n)$ **	$\Theta(1)$ **	$\Theta(1)$
<b>Partie</b>	$\Theta(n)$	$\Theta(n)$ **	$\Theta(1)$ **	$\Theta(1)$

$n$  désigne le nombre de case du damier. On a donc  $n = largeur^2$

$m \leq n$ , donc la création de damiers / graphes non vides est :  $\Theta(n + m) = \Theta(n)$

Le type *Partie* est une combinaison des trois autres, et hérite donc de la complexité maximale.

\*\* dans le cas moyen, car dans le graphe d'un damier de hex, le nombre d'arêtes et de l'ordre du nombre de sommets

## 3. Arbre Minimax

### a) Algorithmie

L'arbre minimax est défini récursivement :

- La racine est un nœud représentant la configuration actuelle (damier vide au début)
- Un nœud a autant de fils que de cases vides dans le damier, ou aucun s'il est gagnant.
- Un fils correspond à la configuration du père à laquelle on ajoute un placement de pion.

Ainsi chaque branche complète (de la racine à une feuille) représente un déroulement possible du jeu

L'arbre est ensuite noté : on attribue à chaque nœud une note booléenne (0 ou 1) : 1 signifie que le joueur qui posera le pion menant à ce nœud sera gagnant, et 0 qu'il sera perdant.

La notation récursive de l'arbre est donc très simple :

- Les feuilles sont gagnantes, et donc notées 1
- Les nœuds possédant au moins un fils noté 1 sont notés 0
- Et les nœuds ne possédant que des fils notés 0 sont notés 1

Ce qui se conçoit mieux en français : un coup est gagnant si en suivant l'adversaire ne peut que perdre.

Nous nous intéressons ensuite aux fils de la racine : leurs notes indiquent les coups gagnants de la situation actuelle.

## b) Observations

Nous avons développé un arbre Minimax exhaustif, tel que demandé dans la consigne. Une IA (GotooneBot) l'utilise pour prendre une décision : gagnante.

Nous avons pu le faire tourner sur un damier 2x2 (même 3x3 après une minute de calcul), et sur certaines grilles plus grandes, mais déjà bien remplies.

Nous avons compris par l'expérience que cet algorithme ne peut pas être utilisé de manière réaliste sur des grilles de tailles ordinaires (11x11 et supérieures). Cela est expliqué dans le paragraphe suivant.

## c) Complexité de la structure Minimax

L'arbre contient à la racine le damier actuel, puis récursivement il contient toutes les combinaisons possibles de placement de pions (chaque fils correspond au damier de son père dans lequel on a placé un pion.

Nous nous intéressons ici au cas le pire : la racine représente une partie vide, et tous les nœuds fils sont construits récursivement :

### ***Nombre de noeuds de l'arbre Minimax***

Taille du damier	1x1	2x2	3x3	4x4
Nombre de cases (n)	1	4	9	16
Noeuds : Racine (H=0)	1	1	1	1
H=1	1	4	9	16
H=2		12	72	240
H=3		24	504	3 360
H=4		24	3 024	43 680
H=5			15 120	524 160
H=6			60 480	5 765 760
H=7			181 440	57 657 600
H=8			362 880	518 918 400
H=9			362 880	4 151 347 200
H=10				29 059 430 400
H=11				203 416 012 800
H=12				1 017 080 064 000
H=13				4 068 320 256 000
H=14				12 204 960 768 000
H=15				24 409 921 536 000
H=16				24 409 921 536 000
<b>Nombre total de nœuds</b>	<b>2</b>	<b>65</b>	<b>986 410</b>	<b>66 347 413 863 617</b>

Le nombre de nœud croît à une vitesse ahurissante :

Si  $n$  est le nombre de case du damier, ( $n = largeur^2$ ), alors :

$$nombre\ de\ nœuds = \sum_{i=0}^n \frac{n!}{i!} = \Omega(n!)$$

La création d'un nœud nécessitant la copie d'un damier, et la détermination du vainqueur (deux opérations en  $\Theta(n)$ ), on a alors la complexité de création de l'arbre =  $\Omega(n \times n!)$

L'utilisation d'un arbre initialisé est en  $O(1)$ , puisque il s'agit de consulter les fils de la racine pour trouver une configuration gagnante (si elle existe). Malheureusement, initialiser un tel arbre est irréaliste pour des damiers de taille ordinaire (11x11 et plus)

Notons que  $n$  représente le nombre de cases. Si  $g$  est la largeur de la grille, alors  $n = g^2$ , et la complexité est en  $\Omega((g^2)!)$ , ce qui est plus parlant.

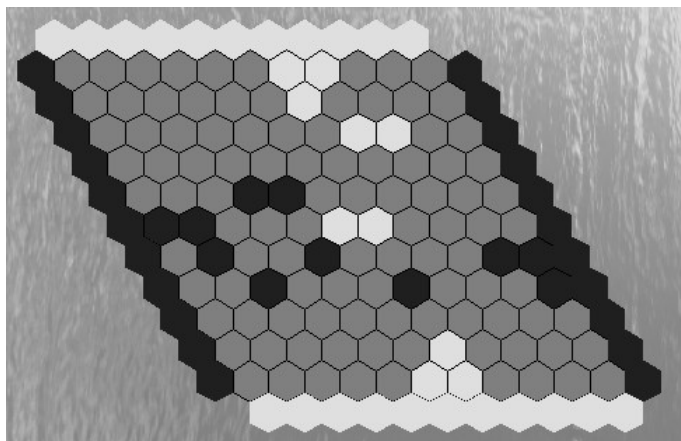
## 4. Détection de ponts et prise de décision

### a) Algorithmie

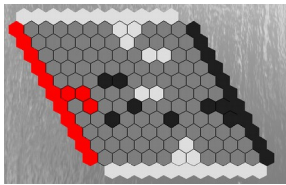
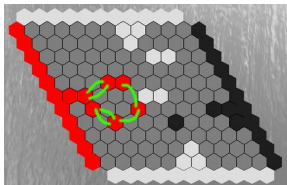
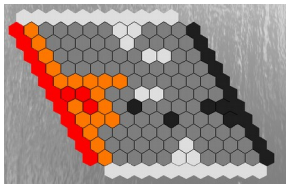
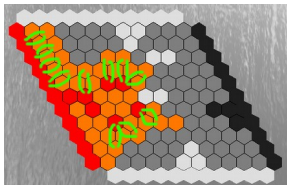
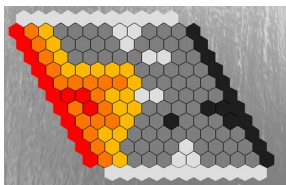
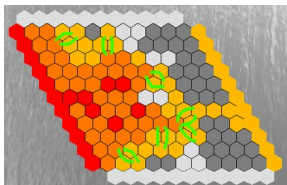
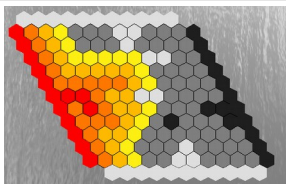
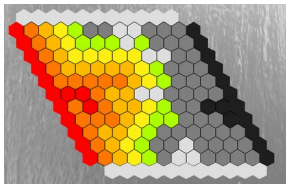
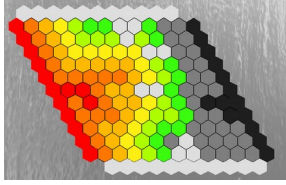
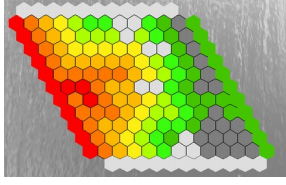
Nous avons développé plusieurs fonctions permettant de détecter les ponts (permettant de relier une case en un coup de deux façons différentes, et ne pouvant donc pas être bloqué en un coup) et de prendre une décision.

- *noeudsRelies* renvoie la liste des nœuds considérés comme reliés à un nœud donné (en considérant que ce nœud appartient à un joueur donné) : ce sont donc tous les voisins de ce nœud appartenant à ce joueur, ainsi que toutes les cases « pontées » appartenant au joueur. Les cases pontées sont celles qui ont au moins deux voisins libres en commun avec le nœud donné.  
La fonction boucle sur son propre résultat afin de trouver un maximum de ponts (en effet, un nœud récemment trouvé peut permettre, lui aussi, de ponter une case)
- *noeudsSuivants* est une fonction qui travaille par « couches ». Elle cherche les nœuds qui seraient reliés en posant 1, 2, 3, etc. pions.  
On lui passe une liste de contenant les nœuds accessibles en n-1, n-2, etc. coups, et elle y ajoute les nœuds accessibles en plaçant un jeton de plus.
- *noeudsAccessiblesEnNCoups* appelle récursivement la fonction *noeudsSuivants* pour établir la liste couches 0 à n (les nœuds accessibles en 0, 1, 2, ..., n coups).
- *distanceMini* est une fonction qui utilise la précédente pour chercher le meilleur moyen de relier les deux bords du joueurs
- *BridgeBot* utilise ces fonctions et trouve ainsi les chemins les plus courts permettant de relier les bords opposés. Ils sont représentés par une liste contenant dessous-listes : les nœuds à 1, 2, etc. coups de distance d'un bord. *BridgeBot* choisit de jouer un coup parmi la sous-liste la moins grande, ce sont donc les coups les plus « urgents ».

Exemple : pour un damier tel que celui ci-dessous, le joueur noir calcule :





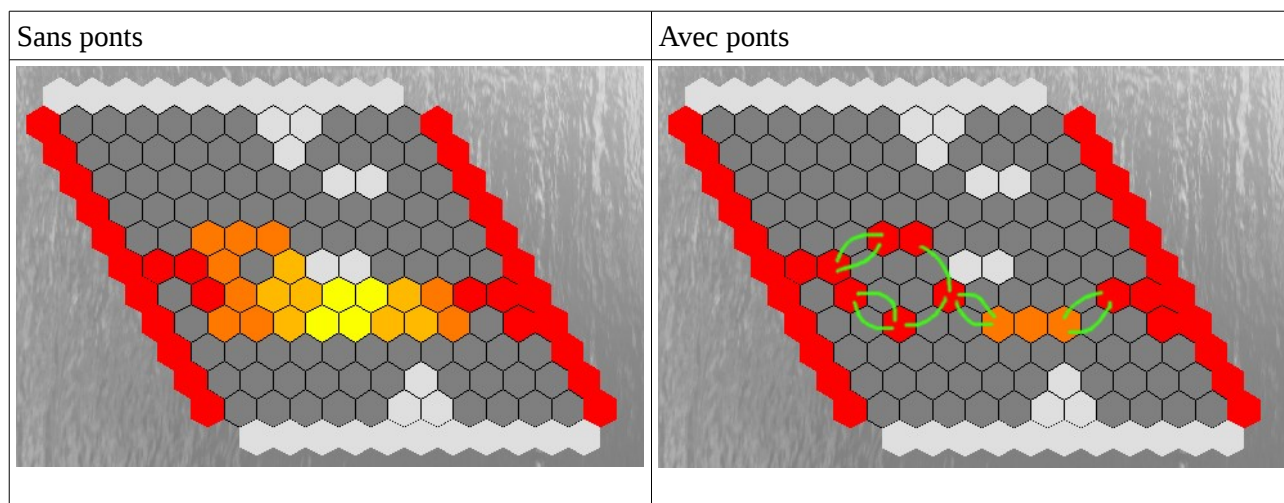
Distance	Sans ponts	Avec ponts
0		
1		
2		
3		
4		
5		
6		

On a trouvé la distance  $d$  séparant les côtés est et ouest (6 sans ponts, 2 avec ponts).

On a aussi trouvé toutes les couches utiles depuis le côté ouest, et il faut maintenant le faire depuis l'est.

On en déduit les nœuds utiles : l'intersection des couches  $i$  (en partant de l'est) et  $d-i+1$  (en partant de l'ouest)

Les nœuds utiles sont donc :



Sans ponts, le coup *le plus urgent* est l'un des deux oranges de droite. Avec ponts, les deux coups à jouer sont équivalents.

## b) Observations

La fonction de détection de ponts permet de trouver des chemins plus court, et demande donc de manipuler des listes moins grandes.. Une fois le chemin établi (l'est et l'ouest sont reliés, en considérant les ponts, et la distance les séparant est de 0), *BridgeBot* utilise l'algorithme sans ponts pour « boucher les trous ».

On a constaté que l'algorithme est plus rapide lorsqu'il considère les ponts : il semble que le gain amené par un chemin plus court compense largement le surcoût lié à la détection des ponts.

Les nœuds, pour être ajoutés à la couche  $n$  par la fonction *noeudsSuivants*, doivent être absents des couches inférieures. Comme on cherche les voisins des nœuds de la couche  $n-1$ , on peut se contenter de vérifier qu'ils sont absents de cette couche  $n-1$  et de la couche  $n-2$ .

Après avoir développé et validé un algorithme fonctionnel reposant sur les listes chaînées, nous avons décider d'implémenter et d'utiliser des arbres rouge-noir pour stocker les nœuds d'une couche. Ainsi les recherches sont bien plus rapides (en  $O(\log n)$  au lieu de  $O(n)$  ). L'effet a été constaté à l'utilisation.

## c) Complexité de l'algorithme de décision

Si  $n$  est le nombre de cases dans la grille, on peut considérer que

- le nombre de nœuds est  $O(n)$  : on a au plus  $n$  noeuds
- le nombre de voisins d'un nœud non fusionné (arêtes), est  $O(1)$  :
  - Les nœuds au centre de la grille ont 6 voisins
  - Les nœuds sur le tour de la grille ont au plus 5 voisins
  - Les nœuds fictifs symbolisant les zones nord, sud, est et ouest (les zones à rejoindre) ont  $\sqrt{n}$  voisins (soit la largeur du damier)

En complexité amortie, on peut « répartir » les  $\sqrt{n}$  voisins de ces nœuds fictifs sur les nœuds du tour de la grille. Ainsi le nombre de voisins est toujours inférieur ou égal à 6 =  $O(1)$

- La complexité du traitement sur le graphe simplifié (où des sommets ont été fusionnés) est au plus égale à celle sur le graphe non simplifié : simplifier le graphe équivaut à supprimer des sommets et des arêtes, jamais à en ajouter : traiter une fusion de nœuds est donc plus rapide que traiter ces nœuds séparément.

Nous considérons donc que le nombre de voisins d'un nœud est  $O(1)$ , sans perte de précision.

## Complexité sans les ponts

- *noeudsRelies* itère sur les voisins d'un nœud et produit donc un résultat  $O(1)$  en un temps  $O(1)$
- *noeudsSuivants* itère sur les voisins des  $i$  nœuds de la couche précédente, et utilise *noeudsRelies*.
  - Elle trouve tous les voisins de la couche en  $\Theta(i)$
  - Elle vérifie qu'ils ne sont pas dans les couches précédentes en  $\Theta(\log a + \log b)$ , où  $a$  et  $b$  sont le nombre d'éléments des couches précédentes (stockés dans des arbres rouge-noir).

Elle produit donc un résultat  $\Theta(i)$  en un temps  $\Theta(i + \log a + \log b)$

- *noeudsAccessiblesEnNCases* est une fonction récursive utilisant *noeudsSuivants*.

On note  $Nb(\lambda)$  le nombre de nœuds de la couche  $\lambda$

- Cas de base 1 : nœuds accessibles en 0 coup = *noeudsRelies*(noeud de départ) :

$$Nb(0) = \Theta(1) \text{ et } T(0) = \Theta(Nb(0)) = \Theta(1)$$

- Cas de base 2 : nœuds accessibles en 1 coup (sur le résultat tu cas de base 1)

$$Nb(1) = \Theta(1) \times Nb(0) = \Theta(1),$$

$$a = \Theta(1), \text{ et}$$

$$T(1) = \Theta(Nb(1) + \log Nb(0)) + T(0) = \Theta(1) + \Theta(1) = \Theta(1)$$

- Cas général :

$$Nb(\lambda) = \Theta(1) \times Nb(\lambda-1) = \Theta(Nb(\lambda-1))$$

$$\rightarrow Nb(\lambda) = c \times Nb(\lambda-1) = \Theta(\lambda)$$

$$T(\lambda) = \Theta(Nb(\lambda) + \log Nb(\lambda-1) + \log Nb(\lambda-2)) + T(\lambda-1) + T(\lambda-2)$$

$$T(\lambda) = \Theta(\Theta(\lambda) + \log \Theta(\lambda-1) + \log \Theta(\lambda-2)) + T(\lambda-1) + T(\lambda-2)$$

$$T(\lambda) = \Theta(\lambda) + T(\lambda-1) + T(\lambda-2)$$

$$T(\lambda) = T(\lambda-1) + T(\lambda-2) + c \times \lambda$$

On vérifie alors facilement que  $T(\lambda) = O(\lambda)$  par récurrence :

- Les cas de base sont en  $\Theta(1) = O(\lambda)$
- Si  $T(\lambda-1) = O(\lambda)$  et  $T(\lambda-2) = O(\lambda)$ , alors  
 $T(\lambda) = O(\lambda) + O(\lambda) + \Theta(\lambda) = \Theta(\lambda) = O(\lambda)$

La complexité est donc de l'ordre de  $\lambda$ , la distance voulue (ce qui est appelé N dans le code)

- *distanceMini* utilise le même algorithme tant que le bord opposé n'est pas atteint (s'il est atteignable), et renvoie cette distance.
  - Si le bord est atteignable, la complexité est de l'ordre du résultat, qui est lui de l'ordre de la largeur du damier.
  - Sinon, la complexité est liée à la « rapidité » dont les couches recouvrent l'espace disponible. Nous affirmons, sans preuve, que c'est une surface qui croît donc « au carré » dans le cas moyen. La complexité moyenne est donc aussi de l'ordre de la largeur du damier (mais celle dans le pire cas est de l'ordre du nombre de cases)

Si  $n$  désigne le nombre de cases du damier, alors la complexité est  $\Theta(\sqrt{n})$

- La prise de décision se fait en cherchant l'intersection de deux listes calculées en cherchant les nœuds accessibles à 0, 1, ...,  $distanceMini(est-ouest)$  cases de distance. La création de ces listes est donc en  $\Theta(\sqrt{n})$ , et elles contiennent chacune  $\Theta(\sqrt{n})$  arbres. Chacun de ces arbres contient au plus  $n$  éléments, mais en moyenne  $\Theta(\sqrt{n})$  (car le nombre d'éléments  $\times$  le nombre d'arbres ne peut dépasser le nombre total  $n$  de nœuds du graphe, puisqu'on n'a aucun doublon). La recherche de l'intersection se fait en *filtrant* les arbres rouge-noir de la première liste avec ceux de la seconde. La complexité moyenne d'un filtre est donc en  $\Theta(\sqrt{n} \log \sqrt{n})$

Enfin, la complexité moyenne du traitement total est donc :

$$\Theta(\sqrt{n}) \times \Theta(\sqrt{n} \log \sqrt{n}) = \Theta(n \log \sqrt{n}) = \Theta(n \log n)$$

## Complexité avec les ponts

L'algorithme général étant identique, nous allons pouvoir réutiliser l'étude de complexité ci-dessus. Les deux seules différences sont la prise en compte des ponts dans les fonction *noeudsRelies* et *noeudsSuivants*.

La complexité de la première est assez difficile à exprimer car il nous faudrait connaître la probabilité qu'un pont existe pour un groupe de  $i$  nœuds donné. Cette probabilité n'est pas constante car elle est liée à  $i$ . Néanmoins on se rend compte que, le graphe étant fini, elle ne peut pas être constamment croissante.

Pour approximer cette complexité, on utilisera le fait que *BridgeBot* cherche à avancer « en ligne droite » par le plus court chemin. Ainsi on peut estimer que :

- Chaque pont provoque un calcul supplémentaire pour la fonction *noeudsRelies*, en  $\Theta(i)$
- Chaque pont réduit la distance entre l'est et l'ouest d'une case (on avance de deux au lieu d'un), et évite un calcul en  $\Theta(\sqrt{n} \log \sqrt{n})$

Or dans le cas moyen nous avons vu que  $i$  est  $\Theta(1)$ , et le surcoût est donc faible par rapport au gain

On peut donc déduire que, lorsqu'il est utilisé par *BridgeBot*, l'algorithme utilisant les ponts est  $O(n \log n)$ .

## d) Conclusion

L'utilisation d'arbres rouge-noir à la place des LDC (utilisées dans la première version) à permis de faire diminuer la complexité de  $\Theta(n^2)$  à  $\Theta(n \log n)$ .

Ce résultat est satisfaisant, mais il faut le nuancer :

- Si on a bien une complexité s'écrivant  $c \times n \times \log n + o(n \log n) = \Theta(n \log n)$ . Il faut pour être honnête préciser que  $c$  est une constante importante
- $n$  désigne le nombre de cases de la grille, et non sa largeur. En exprimant la complexité selon la largeur  $g$  de la grille, on obtient :  $n = g^2$  et

$$c' \times g^2 \times \log g + o(g^2 \log g) = \Theta(g^2 \log g).$$

Hors l'esprit humain à plutôt considérer qu'une grille 5x5 est « une case plus grande » qu'une grille 4x4, c'est pourquoi la complexité en  $\Theta(g^2 \log g)$  est plus parlante.

# ORGANISATION ET DÉROULEMENT

*Comment c'est qui qui à fait quoi ?*

## 1. L'équipe

François et Pierre.

Nous avons décidé de travailler ensemble car nous nous connaissions et nous nous apprécions déjà avant (nous sommes tous deux en reprise d'étude, cela nous a rapproché).

## 2. Organisation

Nous n'avons pas les mêmes méthodes de travail, mais nous avons réussi à bien nous coordonner. Nous avons commencé très tôt (dès que possible), et :

1. Nous avons découpé le travail en modules distincts.
2. Nous avons décidé les interactions entre ces modules
3. Nous avons défini les opérateurs de ces modules
4. Nous avons établi l'ordre dans lequel développer et intégrer ces modules
5. Nous avons planifié l'implémentation de ces modules

Cela a été formalisé dans le document de conception rendu (diagrammes UML et définitions des TAD)

Un fois cela fait, nous nous sommes réparti les tâches au fur et à mesure, selon notre avancement respectif.

Pour cela, nous avons fait des « réunions » régulières (toutes les semaines), en plus de nos discussions informelles (nous sommes dans la même classe).

Nous avons mis en place un dépôt git (sur *bitbucket*) afin de mieux nous coordonner.

### **Le mettre sur github et mettre un lien**

Nous avons finalement réussi à tout faire, dans les délais que nous nous étions fixés (et donc ceux fixés par la consigne)

### 3. Répartition du travail

Nous avons travaillé en commun pour toutes les tâches de réflexion et de décision, c'est-à-dire celles précisées dans le chapitre précédent.

Puis nous nous sommes réparti les modules afin de les implémentés seuls, comme l'illustre le calendrier ci-dessous

#### **Périodes consacrées à la spécification et à l'implémentation des différents modules**

Semaine :	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Contrôles :					Maths	Système			Réseaux POO	Archi BDD		Maths SD		Rendu
Vacances :								X				X	X	X
Module	Sous-module													
cretinplay	Application	Spec.	F	F		Spec.		F						
	IHMConsole	Spec.	F	F	F	Spec.	P	F						
	IHMGraphique					Spec.	Spec.		Spec.	P	P		F	
	Joueurs			F		Spec.		P						
	Partie	Spec.		F			P							
	Intégration							P					F	
cretinlib	LDC		P											
	Graphe				Spec.	P	P	P						
	ABR et ARN										P			
cretinhex	Partie	Spec.	P	P			P							
	Historique			P										
	Graphe simplifié				Spec.	Spec.	P	P						
IA 0	Décision (aléa)							P						
IA 1	Arbre minimax					Spec.	Spec.					F	F	
	Fonction éval					Spec.	Spec.					Spec.	F	
	Décision					Spec.	Spec.					Spec.	F	
IA 2	Détection ponts					Spec.	Spec.	P	P	P	P			
	Fonction éval					Spec.	Spec.		P	P	P			
	Décision					Spec.	Spec.				P			
Papiers	CDC													
	Doc conception													
	Avancement													
	Bilan + code													

En jaune : réflexion et spécification (en commun) - En vert : implémentation (François / Pierre séparément)

Les tâches ont été réalisées a-peu-près à l'échéance prévue (cf Document de conception)

La charge de travail n'est pas parfaitement homogène (L'un a pris un peu de retard pendant les examens, l'autre a assuré quelques tâches supplémentaires pour compenser), mais cela n'a pas eu d'incidence.

En effet, le découpage en modules très peu dépendants et bien spécifiés a permis une bonne flexibilité.

# BILAN

## 1. Respect de la consigne

Tâche à réaliser	Réalisée ?
Documents de conception / spécification, etc.	OK
Tests unitaires et d'intégration	OK
V1 : Afficher une grille (ascii + graphique)	OK
V1 : Afficher les information (tour, joueur, etc.)	OK
V1 : Gérer les actions de l'utilisateur	OK
V1 : Lancer les fonctionnalités associées	OK
V1 : Mettre à jour l'interface	OK
V1 : Initialiser un plateau de taille variable	OK
V1 : Gérer le jeu humain contre humain	OK
V1 : Choisir le joueur qui commence	OK
V1 : Vérifier la légalité des coups (ihm graphique)	OK
V1 : Historique, sauvegarde, restauration	OK
V1 : Annulation de coup	OK
V1 : SD dynamique représentant le plateau (type Damier)	OK
V1 : Repérer les groupes	OK
V1 : Graphe simplifié	OK
V2 : Arbre minimax	OK
V2 : Implémenter le minimax exhaustif	OK
V2 : L'utiliser pour détecter une strategie gagnante <i>cf chapitre maths</i>	OK
V3 : Détecter les ponts	OK
V3 : Gérer les groupes de groupes séparés par des ponts	OK
V3 : L'utiliser pour une IA	OK

Nous avons donc honoré notre contrat :-)

## 2. Difficultés rencontrées et solutions trouvées

### a) Travailler en équipe

Après quelques indécidatesses (l'un modifiant le travail de l'autre dans la nuit, le rendant méconnaissable), nous avons rapidement compris qu'il fallait nous tenir à une certaine rigueur.

- Nous avons appris de nos erreurs
- Nous avons mis en place un dépôt *git*



## b) Manipuler des données variées

Les structures de données en C sont toutes dynamiques. Nous avons donc eu à implémenter des types types simples comme la liste chaînée ou l'arbre RN.

Nous avons cherché à rendre ces types réutilisables (par exemple, l'historique est une liste, les voisins d'un nœud sont une liste, etc. )

- L'élément de ces types est un alias de (*void\**)
- L'élément possède une méthode pour être effacé
- L'élément peut posséder une méthode pour être évalué (arbres binaires)

Cela permet une manipulation aisée de ces types, comme par exemple dans la recherche de ponts, qui utilise une liste chaînée d'arbres RN, qui contiennent les nœuds d'un graphe rangés par distance **cf chapitre maths**

## c) Gérer la complexité

Comme expliqué dans le chapitre un peu de maths, la complexité de l'arbre Minimax rend son utilisation irréaliste.

Nous avons néanmoins cherché à le rendre le plus performant possible.

Une attention particulière a été portée au graphe, ainsi qu'à l'algorithme de détection des ponts, comme précisé dans **le chapitre maths**

## d) Gérer la mémoire

La programmation en C est appréciée pour sa performance, c'est aussi parfois presque un défi d'éviter les fuites mémoires. Hors, cette bonne utilisation de la mémoire est particulièrement importante dans notre cas, vu les complexités en mémoire de certains processus.

- Tout ce qui est écrit en C est passé par le crible de *valgrind*, (**cf tests**), et en ressort avec un *in use at exit : 0 bytes in 0 blocs*
- Les tests automatisés permettent de vérifier la mémoire facilement, en cas de modifications ultérieure.

Nous pouvons donc garantir que les bibliothèques natives en C ne produisent aucune fuite mémoire.

### 3. Ce que nous retenons de ce travail

Le travail demandé était très intéressant, et nous avons beaucoup appris.

1. Nous avons appris à (mieux) travailler en équipe
2. Nous avons réalisé l'importance de l'optimisation (complexité)
3. Nous avons découvert JNI (interface Java / C)
4. Nous avons découvert *git*
5. Nous avons pu (re-)mobiliser beaucoup de connaissances déjà acquises, cela fait un bon exercice

Et nous en sommes très contents.

Merci.

François MAHÉ et Pierre POMERET-COQUOT