Rapport de TIPE : Étude du morpion généralisé et de divers changements de règles autour du morpion

Candidat 18371 : BLANCAL Aurélien

Session 2024

Abstract

Tic-tac-toe is a **turn based** game on a 3x3 grid that follows simple rules. However, this game is known for being unfair for the second player and often ending in draws. But, we can **change these rules** and, with the help of game theory, we can analyze their impact and see if it makes the game more fair. One goal of this study is to make Tic-tac-toe fairer while decreasing the number of draws. The other goal is to use Tic-tac-toe simulations to approximate a well-known **NP-complete** problem : MAX-2SAT.

Keywords	Tic-Tac-Toe	Game theory	Changing the rules	Logic	Stratégie gagnante
Mots-clé	Morpion	Théorie des jeux	Changements de règles	Logique	Winning strategy

Table des matières

1 Mise en contexte

Je vais présenter les changements de règles que j'introduis ainsi que poser les problématiques majeures.

1.1 Enjeux et Problématiques

Comme mentionné dans l'abstract, le morpion est un jeu avantageant le premier joueur (représenté par X). Cependant, un jeu est plus apréciable pour les 2 joueurs s'il est équitable et aboutit moins souvent à des égalités. Ainsi sont soulevées deux problématiques et enjeux de ce TIPE :

→ Existe-t-il un changement de règles rendant le morpion **équitable** ?

→ Si oui, en existe-t-il un qui **réduit le nombre d'égalités** ?

Mais, au cours de mon étude, je suis tombé sur une utilisation d'un jeu pour calculer un produit matriciel[3], me donnant ainsi l'idée de la problématique suivante :

→ Le morpion, avec un changement de règles, permet-il d'avoir une approximation de **MAX-2SAT** ?

Je m'interresse aussi au morpion généralisé et son étude.

1.2 Introduction au morpion généralisé

Le morpion généralisé est un changement de règles imaginé pour des études théoriques [1], modifiant le morpion ainsi :

→ Le morpion généralisé se joue sur une grille **infinie**

→ La condition de victoire du joueur 1 est d'**aligner k points**, k choisi arbitrairement

→ La condition de victoire du joueur 2 est de pouvoir **bloquer infiniment** le joueur 1.

1.3 Introduction aux changements de règles imaginés

Voici les autres changements de règles que j'ai introduit :

— la variante **misère** [2], applicable à toutes les règles suivantes, consiste à **inverser** gagnant et perdant.

Exemple 1. Sur une grille 3x3 :

X	0	
0	X	
		X

ici joueur 2 gagne

— Le morpion **sans diagonale** qui, comme son nom l'indique, correspond au morpion dont les combinaisons gagnantes sont **privées des diagonales**.

Exemple 2. Sur une grille 3x3 :

X	0	
0	X	
		X

Dans ce cas, joueur 1 ne gagne pas

X	0	
X		
X		0

ici, joueur 1 gagne

— Le morpion **modulaire**, qui à pour idée de **"prolonger"** les coups en dehors de la grille. Cela revient à jouer dans un **tore**.

Exemple 3. Sur une grille 3x3 :

0		X
X		
	X	0

0	X	
X	0	
		X

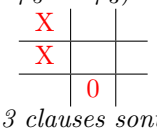
	X	0
	0	X
X		

	X	
	0	X
X		0

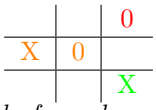
Voici les 4 configurations gagnantes ajoutées par le morpion modulaire.

— Le morpion **logique**. Le morpion classique peut être modélisé logiquement par une formule sous forme normale **disjonctive** (**DNF**). Ce changement de règles consiste à jouer avec une formule sous forme normale **conjonctive** (**CNF**) à la place, pouvant contenir des littéraux négatifs. (plus de détails dans la section suivante). Chaque littéral est associé à une case.

Exemple 4. Considérons la formule logique suivante sur une grille 3x3 :
$(\varphi_1 \vee \neg\varphi_4 \vee \neg\varphi_5) \wedge (\neg\varphi_8 \vee \varphi_5 \vee \neg\varphi_3) \wedge (\varphi_4 \vee \neg\varphi_6 \vee \neg\varphi_9)$



Ici joueur 1 gagne car les 3 clauses sont satisfaites pour lui mais pas pour joueur 2



Ici la formule est satisfaite par les 2 joueurs en même temps, ce qui donne une égalité

2 Implémentation et modélisation

Description de la modélisation du morpion et des implémentations nécessaires. Le code est en **Ocaml** (à part les diagrammes qui ont été faits en **Python**)

2.1 Modélisation

Chaque case du morpion va être associée à une variable logique, indexée selon la position (x, y) de la case ainsi : φ_{x*n+y}, où n est la longueur de la grille. Une règle va être modélisée par la formule logique représentant ses conditions de victoire, qui est sous forme **CNF** ou **DNF** en fontion de la règle. On va aussi introduire le type des **booléens partiels** en ajoutant la variable I (indeterminée) et **associer des valuations partielles aux joueurs** pour modéliser l'état actuel du jeu. Les **stratégies** des joueurs vont être modélisées par des fonctions *morpion* $\rightarrow$ *unit* (elles jouent le coup donné par la stratégie).

2.2 Implémentation de la logique

On implémente le type **booléen partiel** par une union disjointe ainsi :

```
type bool_partiel = V | F | I
(*Cette implémentation des booléens permet de modéliser une valeur logique indeterminée*)
```

Pour implémenter la structure de formule, j'introduit les types suivants :

→ Le type des littéraux est implémenté par un entier associé à la valeur du littéral :
```
type lit = V_ of int | N of int (*Désigne les littéraux, N veut dire Négation*)
```

→ Le type clause correspond à une liste de littéraux :
```
type clause = lit list (*Une clause va être une liste de littéraux *)
```

→ Le type formule est implémenté par l'enregistrement nommé suivant :
```
type formule = {f :clause list; nb_V : int; nb_cl: int;  cnf: bool}
(*le booléen cnf indique si la formule est une dnf ou une cnf, nb_V correspond
au nombre de variables, nb_cl au nombre de clauses, f est une liste de clauses*)
```

Pour évaluer une valuation sur une formule, on dispose de la fontion `eval` suivante qui renvoie le nombre de clauses satisfaites et un booléen indiquant la satisfiabilité de la formule pour la valuation partielle prise en argument (les parcours sont explicités en annexe) :

```
let eval (phi : formule) (valu: bool_partiel array) : int*bool = match phi.cnf with
  |true -> parcours_phi_cnf phi valu
  |false -> parcours_phi_dnf phi valu
```

2.3 Implémentation du morpion et des règles

→ Le type joueur est implémenté par :
```
type joueur = Vide | J1| J2
(*Type permettant de remplir la grille *)
```

→ Les **règles** sont implémentées par des **fonctions** $int \to formule$ qui vont générer la formule logique associée (exemples en annexe). La règle associée au morpion va prendre le nombre et la taille des clauses en arguments, qui vont être curryfiés pour correspondre au typage d'une règle. On peut aussi modifier la répartition des négations.

→ Le type **morpion** est implémenté par l'enregistrement nommé suivant :

```
type morpion = {
  grille : joueur array array; (*La grille remplie du type joueur *)
  misere: bool; (*Implémentation de la variante misère *)
  mutable joueur : joueur; (*Garde une trace du joueur dont c'est le tour de jouer *)
  valu_j1: bool_partiel array; (*Valuation associée au joueur 1 *)
  valu_j2: bool_partiel array; (*Valuation associée au joueur 2 *)
  phi: formule}
  (*Formule logique correspondant aux conditions de
  victoire de la règle du morpion actuelle*)
```

Même si inutile actuellement, le champ grille est toujours là car utilisé dans des fonctions codées avant le passage en logique. On dispose aussi d'une fonction `morpion_init` implémentée ainsi :

```
let morpion_init (n: int) (regle : int->formule) (mis: bool) : morpion=
(*n longueur de la grille*)
  let form = regle n in
  let g = Array.make_matrix n n Vide in
  let j1 = Array.make (n*n) I in
  let j2 = Array.make (n*n) I in
  {grille = g; misere = mis; joueur = J1; valu_j1 = j1; valu_j2 = j2 ;phi = form}
```

Cette fonction est utile pour écrire des tests.
Remarque : Le morpion modulaire va être utile pour simuler le morpion généralisé sur une grille finie car il n'est pas restreint à seulement 2 diagonales gagnantes.

2.4 Implémentation d'algorithmes de théorie des jeux

Initialement, j'aurais voulu faire un calcul des attracteurs pour déterminer les stratégies gagnantes mais le nombre de sommets à génerer est beaucoup trop gros même pour une grille 3x3 (exactement **986 410**), donné par la formule suivante (preuve en annexe) :

$$\sum_{i=0}^{n^2} \frac{(n^2)!}{(n^2-i)!} \tag{1}$$

Ainsi je me contenterais d'un minmax avec une profondeur assez grande pour mes tests (dans la mesure de ce que mon ordinateur peut supporter).
Donc, je vais implémenter trois stratégies différentes pour mes tests :

→ Une stratégie renvoyant un coup **aléatoire** parmi les coups possibles restants, implémentée par :

```
let strat_alea (m : morpion): unit  = (*Joue un coup aléatoire *)
  let coups = coups_possibles_tab m in
  (*calcul d'un tableau des coups possibles restants de la grille*)
  let p = Array.length coups in
  let nb = Random.int p in
  let (x, y) = coups.(nb) in
  jouer_coup x y m
```

→ Une stratégie basée sur un algorithme **minmax**, détaillé en annexe.

→ Une stratégie de **blocage** pour le joueur 2, utile pour le morpion généralisé, implémentée ainsi :

```
let strat_blocage (h: morpion -> int* (int*int))(p: int) (m:morpion) : unit =
(*vole le coup optimal du joueur 1 pour l'état actuel du jeu*)
  m.joueur<- joueur_suivant m.joueur; (*joueur_suivant renvoie J2 pour J1 et J1 pour J2*)
  let vale, (x,y) = minmax_blocage h p m in (*minmax mais qui ne joue pas le coup et le renvoie*)
  m.joueur<- joueur_suivant m.joueur;
  jouer_coup x y m
```

Pour pouvoir utiliser l'algorithme minmax pour des analyses de règles on dispose des heuristiques suivantes :

```
let heuris_cnf (m: morpion): int* (int*int)

let heuris_cnf_misere (m: morpion): int* (int*int)

let heuris_dnf (m: morpion): int* (int*int)

let heuris_dnf_misere (m: morpion): int* (int*int)
```

→ Pour les **DNF**, les heuristiques se basent sur le calcul d'un score associé à un coup qui dépend de combien de clauses il bloque pour l'adversaire et de l'avancement dans la complétion d'une clause qu'il permet. Bien sur, si un pion ennemi est déjà sur cette clause ou si le joueur à déjà bloqué la clause, elle va compter pour 0 dans le score. C'est un parcours sur toutes les clauses.

→ Pour les **CNF**, elle se base sur l'évaluation de la formule qui renvoie le nombre de clauses satisfaites. On évalue avec les valuations des 2 joueurs et, pour estimer la rentabilité d'un coup, on soustrait l'evaluation de celle du joueur actuel par celle de l'autre.

Pour l'approximation de MAX-2SAT, on va modifier un peu minmax et utiliser une heuristique de coopération spécifiée ainsi :

```
let heuris_cooperative (m:morpion): int* (int*int)
```

Son code entier est en annexe comme exemple d'implémentation d'heuristique.
Pour simuler des parties de morpions, on dispose des 2 fonctions suivantes :

```
let morpion (m: morpion) (strat_j1: morpion -> unit) (strat_j2: morpion ->unit): joueur  =
  reset m ; (*remet la grille à Vide, les valuations à I, et joueur à J1 *)
  let n = Array.length m.grille in
  let rec aux (p: int) (n: int) sat_j1 sat_j2= match p with
    |p when p = (n*n ) -> if m.phi.cnf then (if sat_j1>sat_j2 then J1
                                        else (if sat_j1 = sat_j2 then Vide
                                              else J2))
        else Vide
    |p -> if (m.joueur=J1) then strat_j1 m
        else strat_j2 m;
        let (valeur1, b1)= eval m.phi m.valu_j1  in
        let (valeur2,b2) = eval m.phi m.valu_j2 in
        if (b1 && b2) then Vide
        else if (b1 && m.misere) then  J2
        else if (b1 && not(m.misere)) then J1
        else if (b2 && m.misere) then J1
        else if (b2 && not(m.misere)) then J2
        else  aux (p+1) n valeur1 sat_j2
  in aux 0 n 0 0;;

let morpion_max_sat (m:morpion) (strat: morpion-> unit) : int
(*va renvoyer le nombre de clauses satisfaites par le joueur 1 à la fin de la partie*)
```

3 Résultats théoriques et tests en pratique

3.1 Conjectures sur les changements de règles et résultats connus

Morpion généralisé : D'après [1], le joueur 1 gagne pour $k \leqslant 5$ (le cas $k = 5$ par brute force, les autres ont une stratégie en moins de 5 coups), tandis que le joueur 2 gagne pour les cas $k \geqslant 8$ par un système de pavage. Les cas $k = 6$ et $k = 7$ restent indeterminés, mais les chercheurs conjecturent que c'est le joueur 2 qui gagne[1].

Morpion sans diagonale : Je conjecture que pour des grilles 3x3 et plus, il va aboutir à des égalités.

Morpion modulaire : Vu qu'il est utilisé pour simuler le morpion généralisé, je conjecture que les résultats vont être assez similaires à ceux du morpion généralisé.

Morpion logique : Dû à sa flexibilité de paramètres, on devrait pouvoir obtenir un jeu équitable tout en réduisant les égalités.

3.2 Le problème MAX-2SAT

Le problème **MAX-2SAT** est un problème d'optimisation formulé ainsi :

MAX-2SAT
Entrée : une formule ψ en forme cnf à m clauses de taille $k \leqslant 2$
Sortie : une valuation $\nu : \mathbb{V} \to \mathbb{B}$ sur les variables de ψ
Optimisation : ν doit satisfaire le maximum de clauses possibles

Avec un seuil, on peut le transformer en problème de décision. Mais, au vu de mon implémentation, j'approxime plutot le problème **STRICT-MAX-2SAT**, qui se concentre sur les clauses de taille exactement 2. **STRICT-MAX-2SAT**, avec un seuil, est **NP-Complet** (preuve en annexe). Ainsi, vu que les deux sont très liés (la preuve se fait par une réduction de MAX-2SAT à STRICT-MAX-2SAT), approximer **STRICT-MAX-2SAT** revient à approximer **MAX-2SAT**.
Voici le code de calcul par force brute de MAX-2SAT que j'utiliserais comme comparaison :

```
let max_sat_valeur (phi: formule) :int =
  let valu = Array.make phi.nb_V I in
  let rec aux valu acc=
    if acc = phi.nb_V then (let (vale, b)= eval phi valu in vale)
    else (
      let bis = Array.copy valu in
      valu.(acc)<-V; bis.(acc)<-F;
      max (aux valu (acc+1)) (aux bis (acc+1))
    )
  in aux valu 0
```

Cette algorithme est de compléxité temporelle exponentielle en $|\psi|$
(en $\mathcal{O}(2^{|V|})$ avec V l'ensemble des variables de ψ).

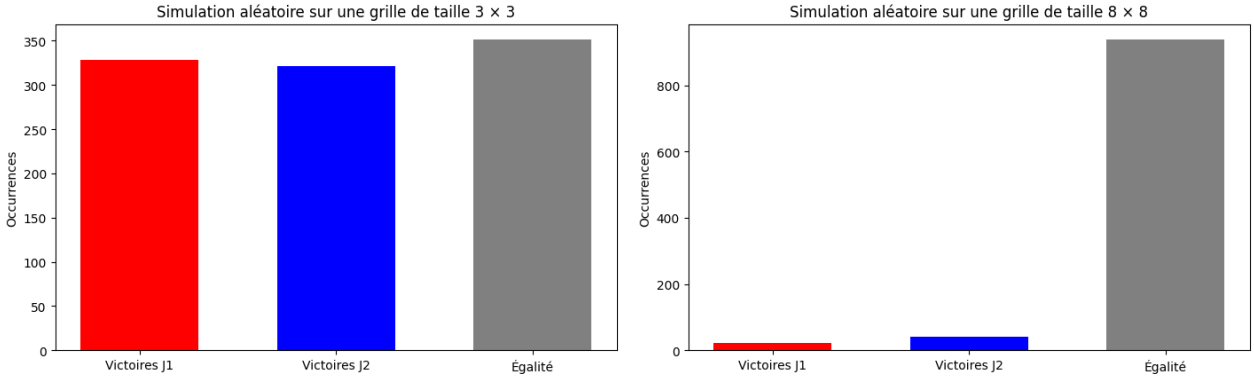
3.3 Tests concrets et analyse

Morpion généralisé :
Voici les résultats obtenus pour le morpion généralisé (simulé avec le morpion modulaire) pour l'algorithme minmax avec une stratégie de blocage pour le joueur 2 : (**E= égalité** *i.e.* une victoire du Joueur 2 pour le morpion généralisé)

	profondeur	1	2	3	4	5
k						
3		J1	J1	J1	J1	J1
6		E	E	E	E	E
7		E	E	E	E	E

Les résultats obtenus sont cohérents avec les résultats déjà prouvés ($k = 3$) et les conjectures actuelles des chercheurs.

Morpion sans diagonales :
→ Pour une stratégie aléatoire, on obtient :



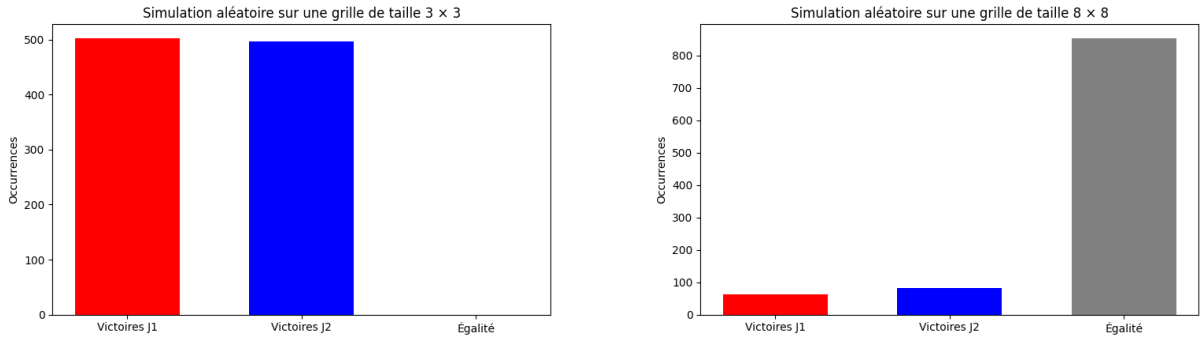
On voit déjà la tendance à donner plus d'égalités.

→Pour minmax, voici les résultats :

	profondeur	1	2	3	4	5	6	7
taille grille								
3		E	E	E	E	E	E	E
5		E	E	E	E	E	E	E
7		E	E	E	E	E	E	E

c'est cohérent avec les conjectures.

Morpion modulaire :
→Pour une stratégie aléatoire, on obtient :



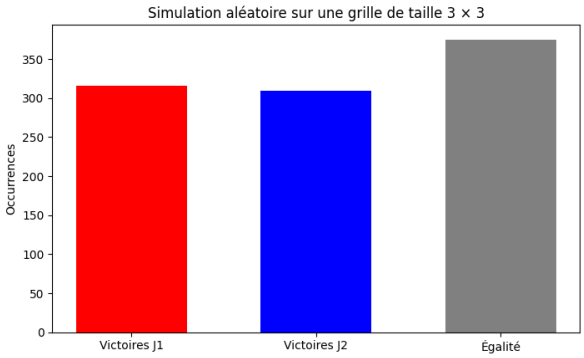
On voit déjà un peu la similitude avec le morpion généralisé pour la taille 8×8.

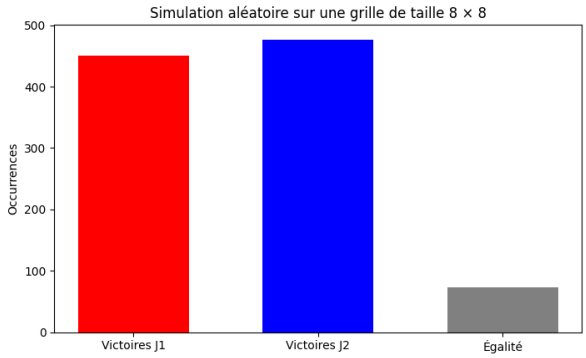
→Pour minmax, voici les résultats :

	profondeur	1	2	3	4	5
taille grille						
3		J1	J1	J1	J1	J1
5		J1	E	E	E	E
7		J1	E	E	E	E

On constaste une similitude avec le morpion généralisé (la profondeur 1 doit venir de la construction de mon heuristique et le cas $k = 5$ est prouvé par force brute [1], donc c'est normal que mon algorithme n'arrive pas à faire gagner le joueur 1).

Morpion logique :
→Pour une stratégie aléatoire, on obtient (on note m le nombre de clauses, k leurs tailles) :



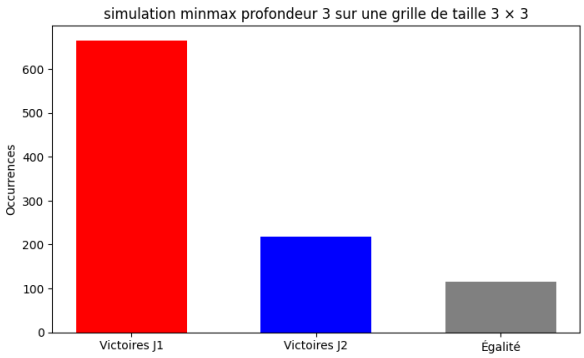


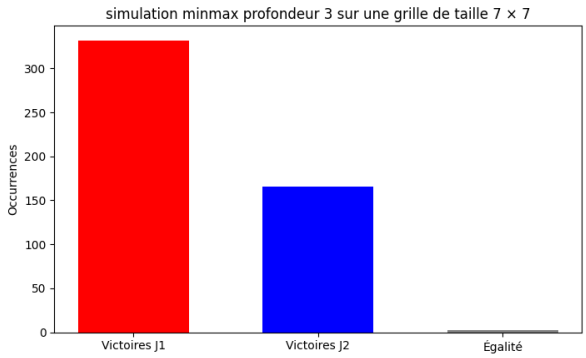
Ici, $m = 50, k = 6, \mathbb{P}(negation) = 0.82$

Ici, $m = 75, k = 10, \mathbb{P}(negation) = 0.55$

On voit déjà l'efficacité des paramètres pour génerer une bonne répartition, même pour une stratégie aléatoire.

→ Pour minmax, voici les résultats (les formules sont générées aléatoirement) :



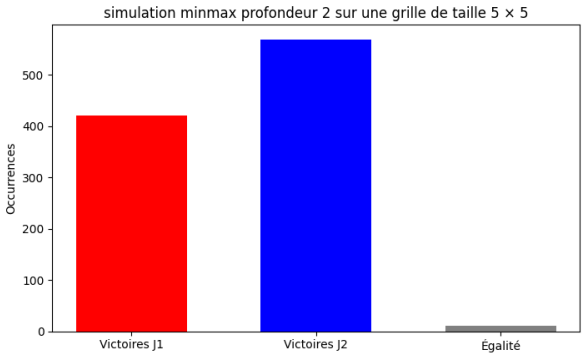


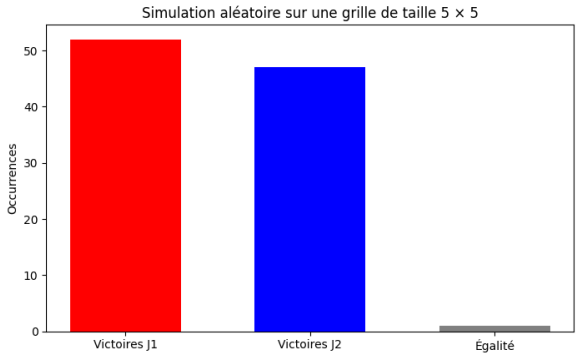
Ici, $m = 8, k = 5, \mathbb{P}(negation) = 0.45$

Ici, $m = 30, k = 8, \mathbb{P}(negation) = 0.50$

Pour des grilles de taille 3 et 7, c'est assez peu équitable, même si c'est les meilleurs paramètres que j'ai trouvés.

→Cependant, pour une grille 5x5 :



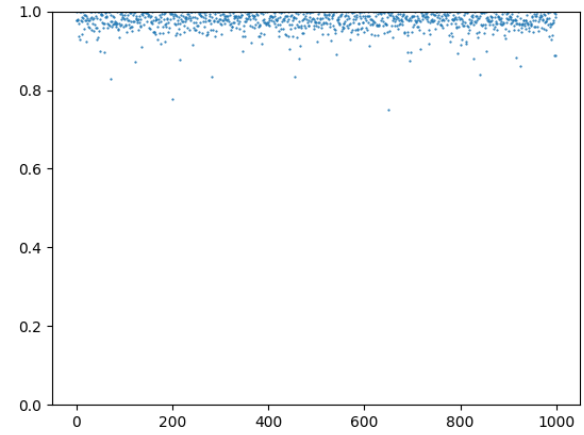


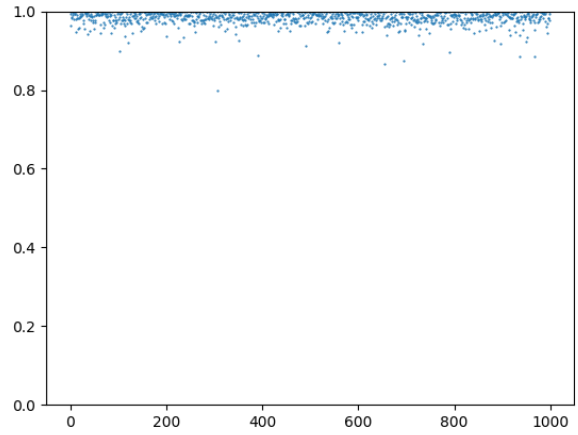
on a des résultats plus convaincants et cohérents avec les conjectures.

Variante misère : J'ai effectué des tests avec la variante misère mais je n'ai rien obtenu de notable, ce qui est cohérent car cette variante ne change pas grand chose du point de vue d'un ordinateur. C'est une variante efficace du point de vue humain pour modifier la façon de penser.

Approximation de MAX-2SAT :

Voici les résultats de tests sur 2 profondeurs (quotient $\frac{morpion}{forcebrute}$) :





Profondeur 1 *Profondeur 2*

En fait, on a une $\frac{1}{2} - approximation$ de MAX-2SAT (preuve en annexe), qui a une moyenne autour de 0.9 sur des formules générées aléatoirement avec $\mathbb{P}(negation) = 0.5$. La profondeur 2 à la meilleure rentabilité en terme de résultat par rapport à la durée d'éxécution, et l'algorithme s'éxécute bien en temps polynomial en $|\psi|$, car la profondeur est indépendante de $|\psi|$.
La profondeur 2 est environ en $\mathcal{O}(n^6 * m * k)$, où n^2 =nombre de variables de ψ, m est le nombre de clauses et k la taille des clauses.

4 Conclusions

Morpion généralisé : On a bien eu les résultats souhaités.

Conclusions sur les règles :

→ La variante misère n'apporte rien théoriquement, c'est une variante faite pour perturber les joueurs humains.

→ Le morpion modulaire est assez similaire au morpion généralisé, donc deséquilibré.

→ Il existe des variantes du morpion le rendant équitable. Nottament le morpion sans diagonale qui aboutit tout le temps à des égalités, mais qui est inintéressant à jouer.

→ Le morpion logique permet d'aboutir à un jeu plus équitable tout en réduisant les défaites, nottament sur une grille 5x5. Cependant, il semble difficile pour un humain de jouer au morpion logique, étant donné le nombre de clauses et leurs tailles à analyser à chaque coup. De plus, cela change beaucoup du morpion de base, à tel point que ça peut être méconnaissable. Ainsi, même si il est équitable théoriquement, le morpion logique n'est pas jouable en pratique et peut ne plus être considéré comme un morpion.

Conclusion sur l'approximation : Le morpion logique donne une $\frac{1}{2} - approximation$ de MAX-2SAT, qui est un problème NP-Complet avec un seuil, en un temps polynomial.

5 Bibliographie

→ [1] Ce que vous ne savez pas sur le morpion - Aline Parreau

→ [2] Tic-Tac-Toe Variants - Wikipédia

→ [3] BEN BRUBAKER : AI Reveals New Possibilities in Matrix Multiplication

6 Annexe

Annexe : Code des parcours de eval

```
let rec parcours_clause_conjonctive (clause : clause) (valu: bool_partiel array): bool =
    match clause with
        |[]->false
        |V_ x ::t when valu.(x)=V-> true
        |N x :: t when valu.(x)=F->true
        |_::t ->parcours_clause_conjonctive t valu

let parcours_phi_cnf (phi: formule) (valu: bool_partiel array): int*bool  =
  let rec aux f  acc= match f with
    |[]-> acc, (acc = phi.nb_cl)
    |cl::t->
        if parcours_clause_conjonctive cl valu then aux t (acc +1 )
        else aux t acc
  in
  aux phi.f 0

let rec parcours_clause_disjonctive (clause: clause) (valu: bool_partiel array) : bool =
  match clause with
  |[]->true
  |V_ x::t when valu.(x)=I -> false
  |V_ x::t when valu.(x)=F -> false
  |_::t-> parcours_clause_disjonctive t valu

let parcours_phi_dnf (phi : formule) (valu: bool_partiel array): int* bool =
  let rec aux f = match f with
    |[]->0, false
    |cl::t->
        if parcours_clause_disjonctive cl valu then 1, true
```

```
        else aux t
  in aux phi.f
```

Annexe : Exemples de règles, le morpion modulaire et le morpion logique

```
let modulo (n: int): formule = (* n: longueur de la grille *)
  let cnf = false in
  let nb_v = n*n in
  let nb_cl = (4*n ) in
  let phi = ref [] in
  for i=0 to n-1 do
    let col_i = ref [] in
    let lig_i = ref [] in
    let diag_i = ref [] in
    let anti_diag_i = ref [] in
    for j=0 to n-1 do
      lig_i:= V_ (i*n + j)::!lig_i;
      col_i:= V_ (i + j*n):: !col_i;
      diag_i:= V_ (j*n + (congru (i+j) n) ) ::!diag_i;
      anti_diag_i := V_ (j*n + (congru (i-j) n ))::!anti_diag_i
    done;
    phi := (!anti_diag_i)::(!diag_i)::(!col_i):: (!lig_i) :: !phi
  done;
  {f= !phi; nb_V = nb_v; nb_cl= nb_cl; cnf= cnf} (* renvoie la formule *)

  let genere_cnf (cl: int) (k: int) (n: int): formule =
  (* Pour l'utilser en règle, on va curryfier le nombre
  de clauses et d'éléments par clause *)
  let cnf = true in
  let nb_v = n*n in
  let phi = ref [] in
  let vu = Array.make nb_v false in
  for i=0 to cl-1 do
    let clau = ref [] in
    for j=0 to k-1 do
      let y = ref (Random.int nb_v) in
      while vu.(!y) do (* assure qu'il n'y a pas de clauses contenant le tiers exclus *)
        y := Random.int nb_v
      done;
      vu.(!y)<-true;
      let rand = Random.int 100 in (* permet de controler la repartition de négation *)
      if (rand<50) then clau := N (!y)::!clau
      else  clau := V_ (!y)::!clau
    done;
    for j=0 to nb_v -1 do
      vu.(j)<-false
    done;
    phi:= (!clau)::!phi
  done;
  {f= !phi; nb_V = nb_v; nb_cl= cl;cnf = cnf}
```

Annexe : Preuve par dénombrement de (1)

On a le sommet de départ : $\frac{(n^2)!}{(n^2-0)!} = 1$ sommet

$\hookrightarrow$Du sommet de départ partent n^2 arêtes vers de nouveaux sommets : $\frac{(n^2)!}{(n^2-1)!} = n^2$ nouveaux sommets

↪De ces n^2 sommets partent n^2-1 arêtes vers de nouveaux sommets : $\frac{(n^2)}{(n^2-2)} = n^2 * (n^2-1)$ nouveaux sommets à créer
De proche en proche, cela fait un total de :

$$\sum_{i=0}^{n^2} \frac{(n^2)!}{(n^2-i)!}$$

sommets à créer pour simuler une partie de morpion.

Annexe : MinMax

```
let minmax (h: morpion-> int* (int*int)) (p: int) (m: morpion): unit =
(*On va curryfier l'heuristique et la profondeur pour l'utiliser en statégie *)
  assert (p>=1);
  let rec aux prof coup_prec =
    let (nb1, b1) = eval m.phi m.valu_j1 in
    let (nb2, b2) = eval m.phi m.valu_j2 in
    if (b1 && b2) then (0, coup_prec)
    else if b2 then (min_int+1, coup_prec)
    else if b1 then  (max_int-1, coup_prec)
    else if est_plein m then (((nb1 - nb2), (coup_prec))  )
    else if prof = 1 then (h m)
    else(
      match m.joueur with
      |J1->
          let liste = coups_possibles_list m in
          let rec parcours_max l max coup= match l with
            |[] -> max , coup
            |(x2, y2):: t ->
                jouer_coup x2 y2 m;
                let (ma, c) = aux (prof -1) (x2, y2) in
                annul_coup x2 y2 m;
                if ma> max then parcours_max t ma (x2,y2)
                else parcours_max t max coup
          in
          parcours_max liste min_int (0, 0)
      |J2->
          let liste = coups_possibles_list m in
          let rec parcours_min l min coup= match l with
            |[] -> min , coup
            |(x2, y2):: t ->
                jouer_coup x2 y2 m;
                let (mi, c) = aux (prof -1) (x2, y2) in
                annul_coup x2 y2 m;
                if mi< min then parcours_min t mi (x2,y2)
                else parcours_min t min coup
          in
          parcours_min liste max_int (0, 0)
      |Vide->failwith "impossible")
  in
  let valeur, (x, y) = aux p (-1, -1) in
  match (x,y) with
  |(x',y') when (x',y') = (-1, -1)->failwith "impossible car la partie est déjà finie
  dans ce cas"
  |(x',y')-> jouer_coup x y m
```

Annexe : L'heuristique de coopération

```
let heuris_cooperative (m:morpion): int* (int*int)=
  let tab= coups_possibles_tab m in
  let n = Array.length (m.grille) in
  let p = Array.length tab in
  let max = ref 0 in
  if m.joueur = J1 then max := min_int
  else max := max_int;
  let coup = ref (0,0) in
  for i = 0 to p-1 do
    let (x, y) = tab.(i) in
    match m.joueur with
    |J1->
        m.valu_j1.(x*n + y)<-V;
        m.valu_j2.(x*n+ y)<-F;
        let j, b = eval m.phi m.valu_j1 in
        if b then (max := max_int; coup:= (x, y))
        else(
          if !max < j then (max:=j; coup:= (x,y) )
        )
        ;
        m.valu_j1.(x*n + y)<-I;
        m.valu_j2.(x*n+ y)<-I
    |J2->
        m.valu_j2.(x*n + y)<-V;
        m.valu_j1.(x*n+ y)<-F;
        let j, b = eval m.phi m.valu_j2  in
        if b then (max := min_int; coup:= (x, y))
        else(
          let z, b = eval m.phi m.valu_j1  in
          if b then ()
          else (
            if !max > -z+j then (max:=-z+j; coup:= (x,y) )
          )
        );
        m.valu_j1.(x*n + y)<-I;
        m.valu_j2.(x*n+ y)<-I
    |_-> failwith "impossible"
  done;
  !max, !coup
```

Annexe : STRICT-MAX-2SAT$_p$ est NP-Complet (p est le seuil)

Preuve que STRICT-MAX-2SAT$_p \in NP$:

Vu que les formules de STRICT-MAX-2SAT$_p$ sont incluses dans celles de MAX-2SAT$_p$, on peut utiliser les certificats et la vérification de MAX-2SAT$_p$, donc STRICT-MAX-2SAT$_p \in NP$

Preuve que STRICT-MAX-2SAT$_p$ est NP-dur :
Montrons que MAX-2SAT$_p \leqslant_P$ STRICT-MAX-2SAT$_p$:
Pour transformer une instance de MAX-2SAT$_p$ en instance de STRICT-MAX-2SAT$_p$:
→ On copie toutes les clauses de taille 2 dans la nouvelle formule qu'on va renvoyer.
→ Pour chaque clause de la forme φ_i,φ_i littéral, on ajoute la clause $(\varphi_i \vee \varphi_i)$ a la nouvelle formule.
On a bien une instance de STRICT-MAX-2SAT$_p$, l'équivalence est vérifiée facilement (la même valuation

donne le même résultat pour les 2 instances). De plus, cette transformation s'effectue en temps polynomial en $|\psi|$. STRICT-MAX-2SAT$_p$ est NP-dur.

$\hookrightarrow$STRICT-MAX-2SAT$_p$ est NP-Complet

Annexe : preuve de la $\frac{1}{2} - approximation$ de STRICT-MAX-2SAT

Remarque :
Le morpion logique force la valuation créée à contenir environ une moitié de valeurs de vérité à Vrai, l'autre moitié à Faux. Ainsi, le pire cas serait une formule contenant que des négations ou que des variables, et aussi tous les $\varphi_i, i \in [\![0; n^2 - 1]\!]$ (φ littéral). Le pire cas contiendrait les clauses $\varphi_i \vee \varphi_i, \forall i \in [\![0; n^2 - 1]\!]$

Preuve :
On suppose que ce sont des clauses de littéraux positifs. Ainsi, la valeur optimale serait n^2 (toutes les variables à Vrai).

- $\rightarrow$ Si n est pair, alors l'approximation va donner $\frac{n^2}{2}$ clauses satisfaites (la moitié des cases occupées par J1) ($\frac{1}{2}$-approx)
- $\rightarrow$ Si n est impair, alors l'approximation va donner $\lfloor \frac{n^2}{2} \rfloor + 1$ ($\geqslant \frac{n^2}{2}$) clauses satisfaites (la moitié des cases sont occupées par J1)
- $\hookrightarrow$ On a bien une $\frac{1}{2}$-approx de MAX-2SAT

Si ce n'était que des clauses de littéraux négatifs, on aurait le même résultat en prenant la valuation du Joueur 2.