Rapport de Projet IA: DualSudoku

Raphael LEONARDI et Baptiste PRAS

13 avril 2025

Table des matières

1 Implémentation des Automates

1.1 Random Move No Validation

Pour implémenter l'automate *Random Move No Validation*, nous avons d'abord implémenté la classe *Couple* afin de stocker des positions dans des paires, où l'attribut $first$ représente la coordonnée x, et l'attribut *second* représente la coordonnée y.

Nous avons ensuite implémenté la méthode $selectMove(CSudokuBoard\ board, Player\ player)$ dans la classe $randomMoveNoValidation$, qui renvoie le coup à jouer quand elle est appelée. Si la grille est pleine, le coup à renvoyer est *null*. Sinon, nous créons une liste de toutes les cases vides et les stockons sous la forme de Couples dans un Array. Si l'Array est vide, le coup à renvoyer est également *null*.

Enfin, nous utilisons un *Random Generator* pour générer aléatoirement un nombre compris entre 0 et la taille de l'Array-1 pour sélectionner aléatoirement une case vide dans l'Array, puis un second *Random Generator* pour générer une valeur aléatoire entre 1 et *gridSize* (inclus). Ainsi nous obtenons un coup à jouer avec sa case à jouer et la valeur à mettre dedans et pouvons le renvoyer pour qu'il soit joué.

1.2 Random Move

Pour implémenter l'automate *Random Move*, nous avons d'abord implémenté dans la classe *Player* la méthode $isValidMove(CSudokuBoard\ board, Move\ move)$, qui vérifie qu'un coup proposé soit bien valide. Ainsi, la méthode vérifie d'abord que la case proposée soit bien vide, elle vérifie ensuite que la valeur proposée dans la case ne se trouve pas déjà ni dans la ligne, ni dans la colonne ni dans la sous-grille où se trouve le coup à vérifier. Enfin, la méthode vérifie que la case proposée ne possède aucune contrainte d'adjacence, et si elle en a une, vérifie que la différence absolue entre la valeur déjà présente (s'il y en a une) dans la case adjacente et celle proposée soit bien de 1. Si toutes ces conditions sont validées, alors la méthode renvoie *true*, elle renvoie $false$ si une seule de ces contraintes n'est pas respectée.

Nous avons ensuite implémenté la méthode $selectMove(CSudokuBoard\ board, Player\ player)$ dans la classe $randomMove$, qui renvoie le coup à jouer quand elle est appelée. Si la grille est pleine, le coup à renvoyer est *null*. Sinon, nous créons une liste de toutes les cases vides et les stockons sous la forme de Couples dans un Array. Si l'Array est vide, le coup à renvoyer est également *null*.

Nous parcourons ensuite toutes les cases vides et ajoutons dans un Array tous les coups possibles. Pour cela, pour chaque case, nous testons chaque valeur de 1 à *gridSize* (inclus) et appelons sur cette case avec cette valeur la méthode $isValidMove(CSudokuBoard\ board, Move\ move)$, si la méthode renvoie True, alors nous ajoutons le coup à l'Array, puis continuons les vérifications.

Enfin, si aucun coup valide n'est trouvé, nous renvoyons *null*, sinon nous utilisons à nouveau un *Random Generator* pour générer aléatoirement un nombre compris entre 0 et la taille de l'Array-1 pour sélectionner aléatoirement un coup à jouer dans l'Array, et renvoyons ce coup pour qu'il soit joué.

1.3 First Valid Move

Pour implémenter l'automate *First Valid Move*, nous avons d'abord implémenté dans la classe *Player* la méthode $coupsPossibles(CSudokuBoard\ board, int\ i, int\ j)$, qui renvoie un Array contenant tous les coups possibles pour une case donnée, en prenant i comme coordonnée x de la case et j comme coordonnée y, la méthode itère sur toutes les valeurs de 1 à *gridSize* (inclus) et appelle $isValidMove(CSudokuBoard\ board, Move\ move)$, si la méthode renvoie True, alors le coup actuel est ajoutée à l'Array et le reste des vérifications s'effectue.

Nous avons ensuite implémenté la méthode $selectMove(CSudokuBoard\ board, Player\ player)$ dans la classe $FirstValidMove$, qui renvoie le coup à jouer quand elle est appelée. Si la grille est pleine, le coup à renvoyer est *null*. Sinon, nous créons une liste de toutes les cases vides et les stockons sous la forme de Couples dans un Array. Si l'Array est vide, le coup à renvoyer est également *null*.

Nous parcourons ensuite toutes les cases vides dans l'ordre d'apparition (en commençant donc avec la première ligne et première colonne, puis en avançant de colonne en colonne puis ligne en ligne), et générons pour chaque case la liste des coups possibles. Si la liste des coups possibles est vide, nous passons à la case suivante, sinon nous choisissons dans cette liste le premier coup en partant de la fin, c'est-à-dire le coup dans cette case qui a le plus grand chiffre. Ainsi, nous avons un coup à jouer avec une case et son chiffre, que l'on renvoie pour qu'il puisse être joué.

2 Implémentation des Intelligences Artificielles

2.1 Evaluated Simulated Board

Afin de pouvoir implémenter différentes intelligences artificielles, il a d'abord fallu implémenter la classe *EvaluatedSimulatedBoard* afin de simuler une grille et de pouvoir jouer des coups fictifs dessus et les évaluer.

Pour cela, nous avons d'abord créé 3 constructeurs qui permettent de créer une *EvaluatedSimulatedBoard* soit à partir d'un *int*, qui est la taille de la grille, soit à partir d'une *EvaluatedSimulatedBoard*, soit à

partir d'une *CSudokuBoard* et d'un *Player*. Ces 3 constructeurs permettent d'initialiser tous les champs de la nouvelle grille et de copier les champs des grilles passées en paramètre dans le cas des 2 derniers constructeurs. La *EvaluatedSimulatedBoard* a comme attributs *eval* qui est l'évaluation de la grille et qui correspond au nombre de points qu'une telle grille rapporterait au joueur s'il arrivait à cet état, $zerosInRow$ et $zerosInColumn$, deux Array contenant le nombre de zéros dans chaque ligne et chaque colonne, *player* le prochain joueur à jouer et $lastMove$ le dernier coup joué.

De nombreuses méthodes (getters et setters) permettent d'accéder aux attributs privés de cette classe. Nous avons notamment $getZerosInRow(int\ row)$ qui renvoie le nombre de zéros dans la ligne row, $getZerosInColumn(int\ col)$ qui renvoie le nombre de zéros dans la colonne col, $decreaseZerosInRows(int\ row)$ qui réduit le nombre de zéros dans la ligne row, très utile lorsque l'on joue un nouveau coup, $decreaseZerosInColumn(int\ col)$ qui réduit le nombre de zéros dans la colonne col, également très utile pour garder la grille à jour lorsque l'on joue un coup. Nous avons également les méthodes $getEval()$, $setEval(int\ eval)$, $getLastMove()$ et $setLastMove(Move\ move)$ qui sont les getters et setters des attributs *eval* et $lastMove$.

D'autres méthodes servent-elles à vérifier rapidement certaines caractéristiques de la grille. La méthode $isFull()$ renvoie $true$ si la grille est complète, $false$ sinon. La méthode $isCellEmpty(int\ row, int\ col)$ renvoie $true$ si la case de coordonnées (row, col) est vide.

La méthode $isRowFilled(int\ row)$ renvoie $true$ si la ligne row est complète, la méthode $isColumnFilled(int\ col)$ renvoie $true$ si la colonne col est complète et la méthode $isSubgridFilled(int\ row, int\ col)$ renvoie $true$ si la sous-grille dans laquelle se trouve la case de coordonnées (row, col) est complète et les méthodes $isColumnAlmostFilled(int\ col)$, $isRowAlmostFilled(int\ row)$ et $isSubgridAlmostFilled(int\ row, int\ col)$ calculent si une ligne, colonne ou sous-grille pourront être complétées au tour suivant, et renvoient la meilleure valeur à mettre dans la case pour les compléter. Ces six méthodes utilisent les attributs $zerosInRow$ et $zerosInColumn$ pour vérifier rapidement ces informations et sont très utiles pour évaluer les bonus de complétion après un coup joué.

Enfin, la méthode $setValue(Move\ move, boolean\ isMaximisingPlayer, boolean\ ia, int\ depth)$ permet de simuler un coup en changeant la valeur d'évaluation de la grille. Elle joue le coup donné en paramètre et mets à jour la grille en changeant notamment le nombre de zéros dans la ligne et la colonne concernées et en plaçant la valeur dans la case. Elle attribue ensuite des points, positifs si c'est le joueur maximisant qui joue, négatif si c'est le joueur minimisant qui joue. Les points attribués sont $+m$ où m est la valeur du chiffre placé, et $n * n$ où n est la taille de la grille pour toute ligne, colonne ou sous-grille complétée par ce mouvement. La fonction d'évaluation comprend également deux heuristiques qui servent au $joueurIA$ et seront détaillées dans la section adéquate ci-dessous. À noter que le paramètre *ia* sert juste à appliquer une heuristique au joueur IA en particulier pour en vérifier l'efficacité.

2.2 Minimax

Le premier joueur IA implémenté est $Minimax$. Nous lui imposons une profondeur maximale de 3. Par convention, le cas d'arrêt est $depth == 1$, c'est-à-dire qu'avec $MAX\_DEPTH = 3$, nous jouerons le coup du joueur maximisant puis le coup du joueur minimisant, et c'est tout. L'explosion de possibilités est bien trop importante pour faire plus. Par exemple, jouer le premier coup se décline en 729 possibilités, et jouer les deux premiers coups en $729 * 701 = 511069$ possibilités.

Tout d'abord, nous avons implémenté la méthode $getValidMoves(EvaluatedSimulatedBoard\ board, Player\ player)$ qui, en utilisant la méthode $coupsPossibles(CSudokuBoard\ board, int\ i, int\ j)$ de la classe *Player*, renvoie tous les coups qui peuvent être joués.

L'algorithme est assez simple, nous le lançons avec une *EvaluatedSimulatedBoard*, si la depth==1, alors il renvoie l'évaluation courante de la grille. Sinon, il simule tous les coups possibles récursivement jusqu'à avoir

atteint sa profondeur maximale. Le joueur maximisant comme le joueur minimisant font appel à la méthode $setValue(Move\ move, boolean\ isMaximisingPlayer, boolean\ ia, int\ depth)$ pour prendre en compte leur coup, et à la fin le coup bénéficiant le plus au joueur maximisant est renvoyé.

2.3 AlphaBeta

Le second joueur implémenté est *AlphaBeta*, ce dernier est en tout point similaire à $Minimax$, à la seule différence près qu'à chaque appel récursif, il effectue, si cela est possible, un *pruning* pour éviter d'explorer des branches de l'arbre qui seraient inutiles. Ainsi, l'algorithme renvoie le même coup à jouer que $Minimax$ mais beaucoup plus rapidement (voir la section 3 sur l'analyse des joueurs).

2.4 Joueur IA

Pour la compétition, nous avons implémenté un joueur IA qui utilise l'algorithme de *AlphaBeta* pour déterminer le meilleur coup. Certaines heuristiques entrent ici en jeu dans la fonction d'évaluation et la profondeur n'est pas fixe mais adaptative à l'état de la grille pour maximiser la profondeur de recherche.

Tout d'abord, la méthode $countEmptyCells(CSudokuBoard\ board)$ renvoie le nombre de cases vides dans la grille en la parcourant. Ensuite, la méthode $doubleRho(CSudokuBoard\ board, Player\ player)$ parcourt la grille et compte le nombre de cases vides ainsi que le nombre de coups possibles. Un coup possible est une valeur unique pouvant être placé dans une case. Enfin, cette méthode renvoie $totalValidMoves/(emptyCount * n)$, cela permet de calculer le degré de liberté de la grille, c'est-à-dire le rapport entre le nombre de coups supposés possibles en prenant simplement en compte le nombre de cases vides, et le nombre de coups réellement possibles. Ces deux méthodes sont utiles au calcul des paramètres nécessaires pour déterminer la profondeur optimale.

La méthode $Depth(int\ f, int\ n, double\ rho, double\ threshold)$ permet de calculer la profondeur de recherche que l'on peut se permettre d'atteindre dans un temps imparti. Ainsi, la méthode estime le nombre de noeuds à visiter en fonction de l'état de la grille en prenant f le nombre de cases vides, n la taille de la grille, rho le coefficient de liberté calculé précédemment et $threshold$ le nombre de noeuds maximums qu'on s'autorise à visiter pour calculer notre coup. À chaque calcul, un facteur /1.6 est appliquée pour prendre en compte qu'au moins 40% des noeuds ne seront pas visités grâce au *pruning* de *AlphaBeta*. Cela permet ainsi d'avoir une meilleure estimation du nombre de noeuds visités à chaque profondeur et donc de la profondeur à choisir. Un $threshold$ à 2 000 000 a été estimé être le meilleur. Assez paradoxalement, l'augmenter réduit les performances de l'IA alors que cela lui fait explorer plus loin, et le réduire est également contre productif. Nous fixons par ailleurs une profondeur minimale à 3 quoi qu'il arrive.

D'autres heuristiques apparaissent dans la fonction d'évaluation, avec les méthodes $isColumnAlmostFilled(int\ col)$, $isRowAlmostFilled(int\ row)$ et $isSubgridAlmostFilled(int\ row, int\ col)$ qui ajoutent un bonus ou malus si le coup joué donne la possibilité, au prochain tour, d'obtenir un bonus de complétion, cela permet d'orienter le jeu vers les bonus de complétion, qui sont ce qui fait la grosse différence de points. Il y a également un bonus de mobilité, calculé avec la méthode $evaluateMobility()$, cela ajoute à l'évaluation de la grille le nombre de coups possibles dans la grille divisé par 2, favorisant ainsi de garder une grille ouverte avec beaucoup de coups possibles. Cette heuristique ne s'applique que si c'est le joueur maximisant qui joue, car lui jouera avec cette logique mais il est peu probable que son adversaire (le joueur minimisant) joue aussi ainsi. Ces deux heuristiques ne s'appliquent que si la taille de la grille est supérieure à 4x4, car elles favorisent l'exploration, or en 4x4 nous favorisons les meilleurs coups immédiats plutôt que l'exploration.
Enfin, si le meilleur coup possible est moins bon que de prendre un malus de n avec un coup invalide, le $joueur\ IA$ choisit de prendre un malus.

3 Analyse des Joueurs

Pour analyser les statistiques des joueurs, nous avons implémenté dans la classe *IACompetitionStrategy* les structures *Stats* et *Return* qui permettent de collecter les statistiques de temps et de noeuds visités par chaque joueur lors d'une partie, et de retourner dans la fonction de jeu principale le coup à jouer et les statistiques à chaque appel à la fonction de stratégie d'un joueur. Cette structure a été étendue à toutes les autres classes pour tous les joueurs pour que la fonction *play*() dans *CSudokuGame* collecte puis affiche ces données en fin de partie.

3.1 Random Move No Validation

Cet automate ne vérifiant pas les coups qu'il joue, il prend très souvent des points négatifs pour avoir joué un coup invalide et est donc très facile à battre pour un humain. Cet automate finit très souvent la partie en négatif et une partie entre deux *Random Move No Validation* peut prendre très longtemps, car bien que le choix du coup soit très rapide, ils peuvent jouer énormément de coups avant d'avoir complété la grille.

Par exemple, une partie sur une grille 9x9 avec cette automate comme joueur 1 et 2 a duré plusieurs minutes (à cause de l'affichage), le temps de calcul total était d'environ 0.009 secondes par joueur pour environ 280 coups joués chacun (alors qu'il faut 81 coups au total pour remplir la grille). Le score final était :

Random Move No Validation -1824 – -1733 *Random Move No Validation*

3.2 Random Move

Cet automate s'améliore beaucoup par rapport au dernier puisqu'en vérifiant ses coups, il n'engrange pas de malus et finit par compléter la grille dans un temps raisonnable. Puisqu'il ne prend pas de malus, il réussit à battre *Random Move No Validation* mais ne fait toujours pas le poids face à un humain.

Deux parties simulées sur une grille 9x9 entre *Random Move No Validation* et *Random Move* ont pris en moyenne un total de 0.01 secondes de calcul pour *Random Move* et les scores finaux étaient :

Random Move No Validation -186 – 322 *Random Move*

Random Move 345 – -308 *Random Move No Validation*

Deux parties simulées sur une grille 4x4 entre *Random Move No Validation* et *Random Move* ont quant à elles pris en moyenne un total de 0.001 secondes de calcul pour *Random Move* et les scores finaux étaient :

Random Move No Validation 15 – 54 *Random Move*

Random Move 69 – 51 *Random Move No Validation*

Il ne fait donc aucun doute que *Random Move* est bien meilleur que *Random Move No Validation*.

3.3 First Valid Move

First Valid Move sélectionne le premier coup disponible dans l'ordre lexicographique des coordonnées de la grille, en mettant le chiffre le plus haut possible dans la case choisie. Malgré tout, il reste assez naïf et peut facilement être battu par un humain. *Random Move No Validation* ayant été battu par *Random Move*, nous le comparerons donc uniquement à ce dernier.

Deux parties simulées sur une grille 9x9 entre *First Valid Move* et *Random Move* ont pris en moyenne un total de 0.001 secondes de calcul pour *First Valid Move* et les scores finaux étaient :

First Valid Move 656 – 195 *Random Move*

$$Random\ Move\ 176 - 260\ First\ Valid\ Move$$

Deux parties simulées sur une grille 4x4 entre *First Valid Move* et *Random Move* ont quant à elles pris en moyenne un total de 0.0002 secondes de calcul pour *First Valid Move* et les scores finaux étaient :

$$First\ Valid\ Move\ 80 - 53\ Random\ Move$$

$$Random\ Move\ 19 - 60\ First\ Valid\ Move$$

Nous pouvons donc en conclure que *First Valid Move* est le meilleur automate, et c'est donc lui qui sera utilisé face aux joueurs IA.

3.4 Minimax

Nous allons maintenant pouvoir comparer *Minimax* avec une profondeur 3, qui va donc calculer 2 coups à l'avance. Il ne fait aucun doute que cette IA sera meilleure que *First Valid Move* car avec 2 coups d'avance, elle prendra le meilleur coup pour elle-même (ce que ne fait pas l'automate), mais ne devrait également pas laisser à l'automate de bonus de complétion, alors que lui ira spécifiquement les chercher.

Deux parties simulées sur une grille 9x9 entre *First Valid Move* et *Minimax* ont pris en moyenne un total de 2.6 secondes de calcul pour *Minimax*, visitant un total de 3 427 169 noeuds et en moyenne 110 553 par coup, avec un maximum à 470 937 noeuds visités en un coup. Les scores finaux étaient :

$$First\ Valid\ Move\ 135 - 447\ Minimax$$

$$Minimax\ 465 - 127\ First\ Valid\ Move$$

Deux parties simulées sur une grille 4x4 entre *First Valid Move* et *Minimax* ont quant à elles pris en moyenne un total de 0.01 secondes de calcul pour *Minimax*, visitant un total de 4 383 noeuds et en moyenne 547 par coup, avec un maximum à 2 346 noeuds visités en un coup. Les scores finaux étaient :

$$First\ Valid\ Move\ 66 - 166\ Minimax$$

$$Minimax\ 120 - 112\ First\ Valid\ Move$$

Nous en concluons donc que *Minimax* est bien meilleur que les automates implantés jusqu'ici.

3.5 AlphaBeta

AlphaBeta testé avec une profondeur 3 devrait obtenir les mêmes scores que *Minimax* puisque l'algorithme est le même, mais le temps d'exécution et le nombre de noeuds visités devraient être nettement plus faibles.

Deux parties simulées sur une grille 9x9 entre *First Valid Move* et *AlphaBeta* ont pris en moyenne un total de 0.775 secondes de calcul pour *AlphaBeta*, visitant un total de 252 921 noeuds et en moyenne 7 664 par coup, avec un maximum à 25 068 noeuds visités en un coup. Les scores finaux étaient :

$$First\ Valid\ Move\ 135 - 447\ AlphaBeta$$

$$AlphaBeta\ 465 - 127\ First\ Valid\ Move$$

Deux parties simulées sur une grille 4x4 entre *First Valid Move* et *AlphaBeta* ont quant à elles pris en moyenne un total de 0.01 secondes de calcul pour *AlphaBeta*, visitant un total de 1 346 noeuds et en moyenne 168 par coup, avec un maximum à 486 noeuds visités en un coup. Les scores finaux étaient :

$$First\ Valid\ Move\ 66 - 166\ AlphaBeta$$

$$AlphaBeta\ 120 - 112\ First\ Valid\ Move$$

Après les avoir fait s'affronter l'une contre l'autre en match aller-retour, *AlphaBeta* et *Minimax* se neutralisent avec chacune une victoire $173 - 165$. Nous trouvons donc bien que *AlphaBeta* et *Minimax* donnent les exacts mêmes coups à jouer, mais *AlphaBeta* est beaucoup plus rapide et visite en moyenne 3x moins de noeuds sur une grille 4x4, et 13x moins de noeuds sur une grille 9x9, réduisant le temps de calcul par presque 4.

3.6 Joueur IA

Nous allons enfin tester le *joueur IA* contre *AlphaBeta* et contre *First Valid Move* pour attester de son efficacité. Ce joueur se veut avoir une profondeur adaptative, qui augmente la profondeur de recherche quand cela est possible, sans aller trop loin. Le joueur n'exploite pas les 5 minutes de temps de calcul accordés par partie (ou 2 minutes sur une grille 4x4) car paradoxalement, trop explorer rend l'IA moins efficace car elle a tendance à jouer des coups qui sont immédiatements peu optimaux, mais le seront dans plusieurs coups avec une très faible probabilité d'arriver. Ainsi trop explorer est contre productif et peut faire perdre cette IA. De plus, sur une grille 4x4, nous gardons la profondeur maximale à 3 car cela est plus optimal que la profondeur adaptative, la grille étant petite et ne nécessitant pas d'explorer loin, il faut privilégier les coups qui rapportent beaucoup de points immédiatement.

Deux parties simulées sur une grille 9x9 entre *First Valid Move* et le *joueur IA* ont pris en moyenne un total de 3.01 secondes de calcul pour le *joueur IA*, visitant un total de 2 473 607 noeuds et en moyenne 74 957 par coup, avec un maximum à 797 972 noeuds visités en un coup. Les scores finaux étaient :

$$First\ Valid\ Move\ 152 - 548\ Joueur\ IA$$

$$Joueur\ IA\ 554 - 146\ First\ Valid\ Move$$

Deux parties simulées sur une grille 4x4 entre *First Valid Move* et le *joueur IA* ont quant à elles pris en moyenne un total de 0.01 secondes de calcul pour le *joueur IA*, visitant un total de 1 299 noeuds et en moyenne 144 par coup, avec un maximum à 486 noeuds visités en un coup. Les scores finaux étaient :

$$First\ Valid\ Move\ 67 - 157\ Joueur\ IA$$

$$Joueur\ IA\ 178 - 50\ First\ Valid\ Move$$

Le *joueur IA* semble donc bien meilleur que *AlphaBeta* contre les automates, mais nous allons maintenant faire s'affronter les deux pour être sûr que les améliorations faites au *joueur IA* sont bonnes. Ainsi, voici deux parties simulées sur une grille 9x9 et deux parties simulées sur une grille 4x4 :

$$AlphaBeta\ 271 - 426\ Joueur\ IA$$

$$Joueur\ IA\ 513 - 351\ AlphaBeta$$

$$AlphaBeta\ 54 - 178\ Joueur\ IA$$

$$Joueur\ IA\ 174 - 54\ AlphaBeta$$

Nous pouvons conclure que le *joueur IA* est donc meilleur que *AlphaBeta* et représente la meilleure stratégie, il maximise ses scores et victoires tout en restant extrêmement rapide (au plus 3 secondes pour une partie complète).

3.7 Conclusion

Sur une grille 4x4, il semble bon de privilégier le gain instantané plutôt que l'exploration et le gain potentiel. Sur une grille 9x9, il semble par contre préférable d'explorer un peu, mais trop explorer est mauvais et il faut trouver un équilibre entre exploration et exploitation des coups à jouer.

Le *joueur IA* semble mieux s'en sortir contre n'importe quelle autre joueur, que ce soit les automates où les autres IA, ce qui est logique puisqu'elle intègre une meilleure exploration et des heuristiques qui favorisent en plus cette exploration.