Slim Tetris Algoritme

Luc Schouten
s2660148

Dylan Macquiné
s2592991

22 februari 2021

1 Inleiding

Dit project is de eerste opdracht van het vak Kunstmatige Intelligentie [2] en omvat drie agenten die het spel Tetris [4] spelen; een random agent, een Monte-Carlo agent en een slimme agent. Deze agenten maken gebruik van verschillende technieken uit de Kunstmatige Intelligentie [1]. Voor deze opdracht was de random agent gegeven en moesten de Monte-Carlo agent en de slimme agent nog geprogrammeerd worden. De agenten spelen het spel Tetris op verschillende manieren op verschillende bord groottes, de uitkomsten hiervan zijn met elkaar vergeleken.

2 Uitleg probleem

In het spel Tetris wordt op een bord van een statische grootte gespeeld. Het spel Tetris kent zeven verschillende stukken welke in stappen van 90 graden mogen worden gedraaid. Deze stukken kunnen niet door de speler worden gekozen maar worden door het spel random aangewezen. Het doel van het spel is om zo lang mogelijk door te spelen, dit kan door verschillende tetromino stukken op een bepaalde manier te plaatsen zodat de bovenkant van het bord nooit bereikt wordt. Zodra een rij op het bord wordt volgespeeld vervalt deze rij, hierdoor kan er voorkomen worden dat de bovenkant van het bord wordt bereikt.[4]

In de versie van Tetris die gebruikt is voor deze opdracht mogen de tetromino stukken boven het bord horizontaal worden verplaatst. Deze kunnen daarna worden losgelaten waardoor ze verticaal naar beneden vallen. De snelheid van het vallen blijft constant gedurende het spel. Daarnaast is het ook mogelijk om 'punten' te scoren in het spel, in deze versie bestaat de score van een spel uit twee componenten: het aantal weggespeelde rijen en het aantal weggespeelde stukken.[2]

Veel versies van het spel werken anders, daar is het namelijk vaak mogelijk om de stukken ook tijdens het vallen nog horizontaal te verplaatsen. Daarnaast vallen de stukken na verloop van tijd steeds sneller. Ook is er een aparte score welke uit verschillende componenten wordt samengesteld.[4]

In deze opdracht zijn er een aantal restricties, zo mag er door een agent maar één zet vooruit worden gekeken. Daarnaast is het ook niet toegestaan om voor een agent een zelflerend programma te schrijven.[2]

3 Relevant werk

Voor deze opdracht is er gebruik gemaakt van een al bestaande Tetris class.[2] Deze class bevat verschillende methods die het Tetris spel voor de opdracht simuleren.

Monte-Carlo agent
Voor de Monte-Carlo agent is gebruik gemaakt van een 'Pure Monte Carlo search'. Dit betekent dat de waarde van een zet wordt bepaald door de gemiddelde score van een aantal random simulaties. Aan de hand van de gemiddelde scores van de zetten kan worden afgeleid welk zet tot de hoogste score leidt, en

dus de beste optie is. De zet met de hoogste score wordt dan uiteindelijk gedaan.[1]

Slimme agent
Voor de slimme agent is gebruikt gemaakt van verschillende werken. Een tactiek die gebruikt wordt om Tetris te spelen is de zogenaamde "flat stacking"techniek[3]. Het idee van flat stacking is om de stukken op een zo horizontaal mogelijk manier te plaatsen waardoor het gevulde bord zo laag mogelijk blijft en er geen hoogteverschillen tussen de geplaatste stukken op het bord ontstaan. Hoogteverschillen zouden het moeilijk kunnen maken om met een stuk een vrije plaats op het bord nog goed op te vullen, waardoor rijen niet meer volledig kunnen worden gevuld. Verder is het volgens flat stacking de bedoeling om lege vakjes niet in te sluiten. Na het insluiten is het niet meer mogelijk om de rij van dit ingesloten vakje weg te spelen. Bij flat stacking wordt een blokje alleen verticaal geplaatst wanneer dit met de eerder opgestelde regels tot de beste uitkomst leidt. [3]

Een ander component van de slimme agent is een score functie. Met een score functie krijgt elke zet een score op basis van de regels die zijn opgesteld in deze functie. Zo krijgt een zet die aan meer regels voldoet een hogere score. Door gebruik te maken van een score functie kunnen verschillende zetten met elkaar worden vergeleken en de beste zet worden bepaald.

4 Aanpak

Monte-Carlo agent
De aanpak voor de Monte-Carlo agent is als volgt. Eerst wordt voor het gegeven tetromino stuk alle mogelijke manieren van plaatsen op het spel bord bepaald. Daarna wordt per mogelijke zet het spel een aantal keer, afhankelijk van het aantal ingevoerde playouts, op een random wijze in een simulatie uitgespeeld. Per optie wordt de gemiddelde score van de playouts berekend, deze worden vervolgens met elkaar vergeleken om de zet met de hoogste score te bepalen. De zet met de hoogste score wordt gezien als de 'beste' zet en wordt in het echte spel gespeeld. Vervolgens herhalen deze simulaties zich op het bord waar de vorige beste zet is gespeeld tot het echte spel is afgelopen.

Voor het bepalen van de beste zet wordt er eerst gekeken naar het gemiddelde aantal rijen dat in de simulaties is weggespeeld. In het geval dat meerdere simulaties evenveel rijen wegspelen wordt er gekeken naar het aantal geplaatste stukken. Dit betekent dat de beste zet wordt bepaald door de simulatie die de meeste rijen wegspeelt en tevens de meeste stukken weet te plaatsen.

Slimme agent
De slimme agent werkt op de volgende manier. Voor het random toegewezen tetromino stuk worden alle mogelijke manier van plaatsen bepaald. Elke mogelijke zet krijgt een score op basis van een aantal regels die passen bij de flat stacking tactiek.

Allereerst wordt er gekeken of een zet het spel beëindigt, er wordt hier gebruik gemaakt van de 'endofgame' method die al deel uitmaakt van de Tetris class. Indien een zet het spel eindigt dan blijft de zet die op dat moment de hoogste score heeft gelden. In het geval dat elke zet het spel beëindigt wordt de eerste zet uitgevoerd. Als een zet het spel niet eindigt, wordt de score van een zet op de volgende manier samengesteld:

Als eerste wordt er gekeken of een zet een of meerdere rijen wegspeelt. Dit wordt gedaan door gebruik te maken van de 'rowscleared' variable die de rows telt in de 'clearrows' method die ook al vooraf in de Tetris class aanwezig was. Het aantal weggespeelde rijen wordt opgeteld bij de score van de zet.

Daarna wordt gekeken naar de hoogste rij die door een zet bereikt wordt. Om dit punt te bepalen wordt gebruik gemaakt van de 'letitfall' method. Wanneer het blokje gevallen is wordt de hoogst bereikte rij opgeslagen in een variabele. De hoogte van het bord min dit hoogste punt wordt dan opgeteld bij de score van de zet. Dit betekent dat een zet die op een heel laag punt op het bord speelt een hogere score krijgt dan een zet die heel hoog op het bord speelt.

Vervolgens wordt er gekeken naar het aantal vakjes dat wordt ingesloten wanneer de zet wordt gedaan. Ingesloten vakjes hebben een grote negatieve impact op het spel omdat de ingesloten vakjes niet gemakkelijk meer opgevuld kunnen worden. Dit zorgt ervoor dat rijen minder makkelijk kunnen worden weggespeeld waardoor de kans groter is dat het spel eerder zal eindigen. Op basis van onze bevindingen scoort de agent het beste wanneer het aantal ingesloten vakjes niet één keer wordt meegeteld in de score maar vier keer. Daarna wordt gekeken naar het aantal 'schacht' vakjes die ontstaan na het plaatsen van een zet. Een schacht is één of een aantal vakjes die van bovenaf niet is ingesloten en precies één blokje breed is. Afhankelijk van de diepte van alle schachten die op het bord ontstaan wordt er een score toegekend. Echter wordt er wel gekeken naar het verschil tussen het aantal schacht vakjes dat aanwezig was voor de zet en na de zet. Als er meer schacht vakjes ontstaan dan dat er voor de zet aanwezig waren, wordt er een negatieve score toegekend aan de zet. Deze waarde is dan gelijk aan het extra aantal schacht vakjes dat is ontstaan. Minder schacht vakjes zorgen aan de hand van dezelfde waardering voor een hogere score. Dit betekent dat een zet die een schacht vult als beter gezien wordt dan een zet die dat niet doet, en dus meer schacht vakjes maakt.

Als laatste worden de scores bij elkaar opgeteld. De zet met de hoogste score wordt gezien als de 'best' zet. Als de score hoger is dan de score van de beste zet die op dat moment bekend was, dan worden de score en de zet vervangen door de zo net bekeken zet die beter was. Wanneer alle mogelijke zetten voor het stuk bekeken zijn wordt de eerste (of enige) van de hoogst scorende zetten uitgevoerd. Na het uitvoeren van de zet wordt het hiervoor besproken proces herhaald voor het volgende toegewezen stuk, totdat het bord vol wordt gespeeld en het spel eindigt.

5 Implementatie

Voor de opdracht moest gebruik worden gemaakt van de programmeertaal C++. Tevens is er een voorbeeldprogramma verschaft waarin een class met een aantal methods al gespecificeerd zijn. De random agent maakte al onderdeel uit van de Tetris class.

Monte-Carlo agent
De Monte-Carlo agent moest nog (gedeeltelijk) gemaakt worden. De method 'playMCgame', die een enkele Monte-Carlo game zou spelen was al geïmplementeerd. De 'doMCmove' method had al een gedeeltelijke opzet. Voor de 'doMCmove' is gebruik gemaakt van de eerdergenoemde 'playrandomgame' method die al compleet en werkend was. Tevens is er gebruik gemaakt van een kopie van het Tetris object om steeds de volgende move te bepalen. Hierdoor oefende de bekeken zet geen invloed uit op het echte bord.

Slimme agent
De slimme agent moest nog volledig geïmplementeerd worden. De slimme speler bestaat uit vier methods. De 'playsmartgame' method is toegevoegd en speelt één spel. Voor het gebruik van de 'playsmartgame' method is gebruik gemaakt van de 'playrandomgame' method met als enige aanpassing de aanroep van de 'dosmartmove' method in plaats van de 'dorandommove' method. De 'dosmartmove' method doet een enkele zet en roept zelf weer verschillende methods aan. Om de door een zet weggespeelde rijen te kunnen bepalen gebruikt de 'dosmartmove' method de 'rowscleared' method. Om vervolgens te kunnen bepalen welke rij een zet maximaal bereikt, wordt de 'letitfall' method gebruikt. Hiervoor is een kleine aanpassing aan de 'letitfall' method gedaan. Om te kunnen bepalen hoeveel blokjes zijn ingesloten en daardoor onbereikbaar zijn is de 'numberofclosedspaces' method toegevoegd. Als laatste wordt de 'countshafts' method gebruikt om de hoeveelheid blokjes die exact één opening breed zijn en nog bereikbaar zijn te achterhalen. Ook voor de 'dosmartmove' geldt dat deze, voor het doen van elke mogelijke zet, gebruik maakt van een kopie van het huidige object om de move op uit te voeren.
Om gebruik te maken van het programma moeten het programma worden gecompileerd en zullen er argumenten aan het programma moeten worden meegegeven. Het compileren kan worden gedaan met een compiler via de terminal, bijvoorbeeld g++. De parameters die moeten worden meegegeven worden aangegeven wanneer het programma wordt gerunned.

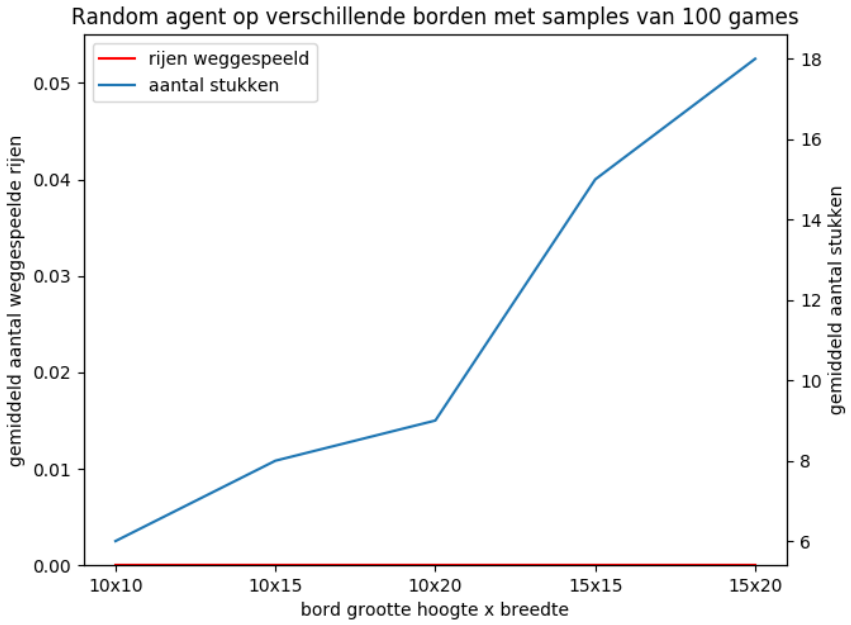
6 Experimenten

Om werking en kwaliteit van de agenten te testen zijn er voor de random agent en de Monte-Carlo agent 100 spellen gespeeld op verschillende bord groottes. Voor de slimme agent is er voor de borden 15×15 en 15×20 gekozen voor twintig spellen omdat de runtime voor die borden erg hoog is. Zo duurde het ongeveer vijf uur om voor een bord van 15x20 twintig spellen te spelen. De groottes van de gebruikte borden zijn respectievelijk: 10x10 10x15 15x15 10x20 15x20 vakjes. Het eerste getal is steeds de hoogte en het tweede getal is steeds de breedte.

De resultaten van de experimenten zijn te vinden in onderstaande tabel:

Tabel 1: Random agent op verschillende bord groottes met 100 games

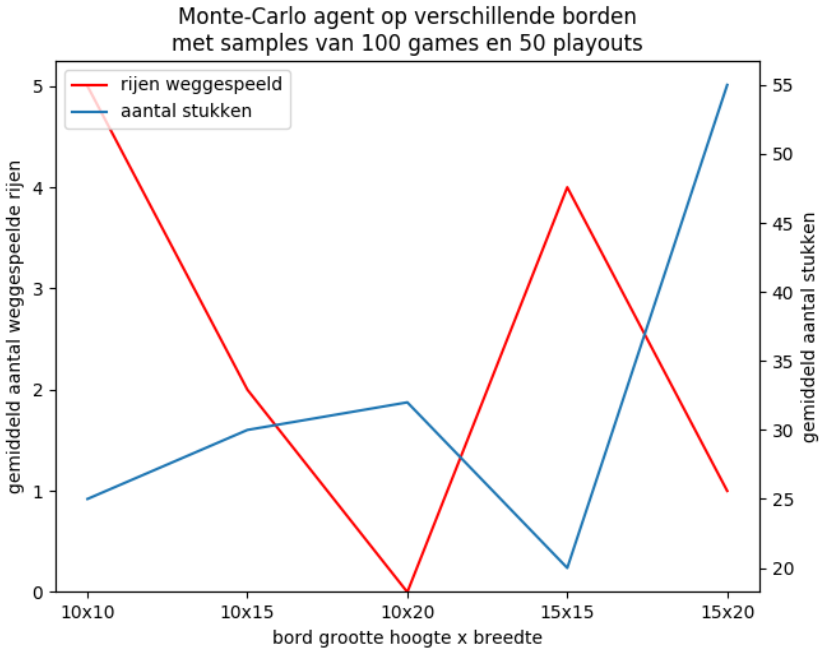
grootte bord hoogte×breedte	gemiddeld aantal rijen weggespeeld	gemiddeld aantal geplaatste stukken
10×10	0	6
10×15	0	8
10×20	0	9
15×15	0	15
15×20	0	18



Random agent
De random agent speel voor alle bord groottes nul rijen weg, het aantal geplaatste stukken neemt samen met het grotere wordende bord toe. De random speler maakt echter geen afgewogen keuzes waardoor het bord snel wordt volgespeeld.

Tabel 2: Monte-Carlo agent op verschillende bord groottes met 100 games en 50 playouts

grootte bord hoogte×breedte	gemiddeld aantal rijen weggespeeld	gemiddeld aantal geplaatste stukken
10×10	5	25
10×15	2	30
10×20	0	32
15×15	4	20
15×20	1	55

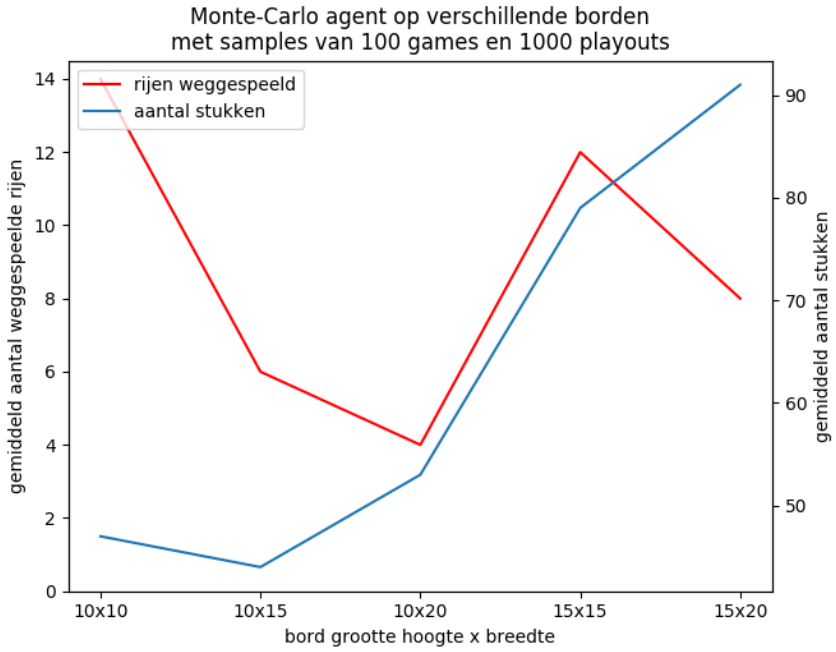


Monte-Carlo agent

Voor de Monte-Carlo agent met 50 playouts kan er geen duidelijk verband worden gelegd tussen het aantal rijen dat wordt weggespeeld, het geplaatste aantal stukken en de toename van de grootte van het bord.

Tabel 3: Monte-Carlo agent op verschillende bord groottes met 100 games en 1000 playouts

grootte bord hoogte×breedte	gemiddeld aantal rijen weggespeeld	gemiddeld aantal geplaatste stukken
10×10	14	47
10×15	6	44
10×20	4	53
15×15	12	79
15×20	8	91

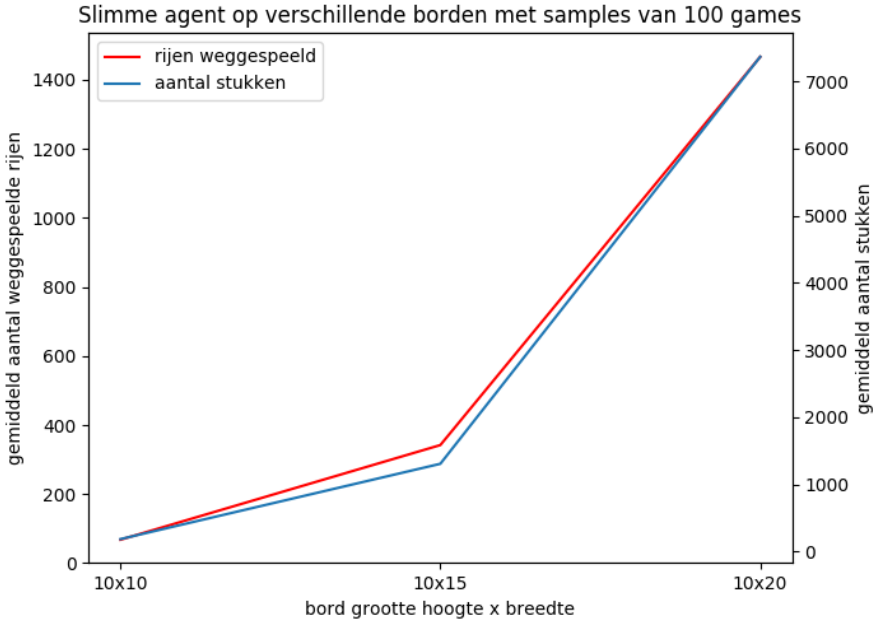


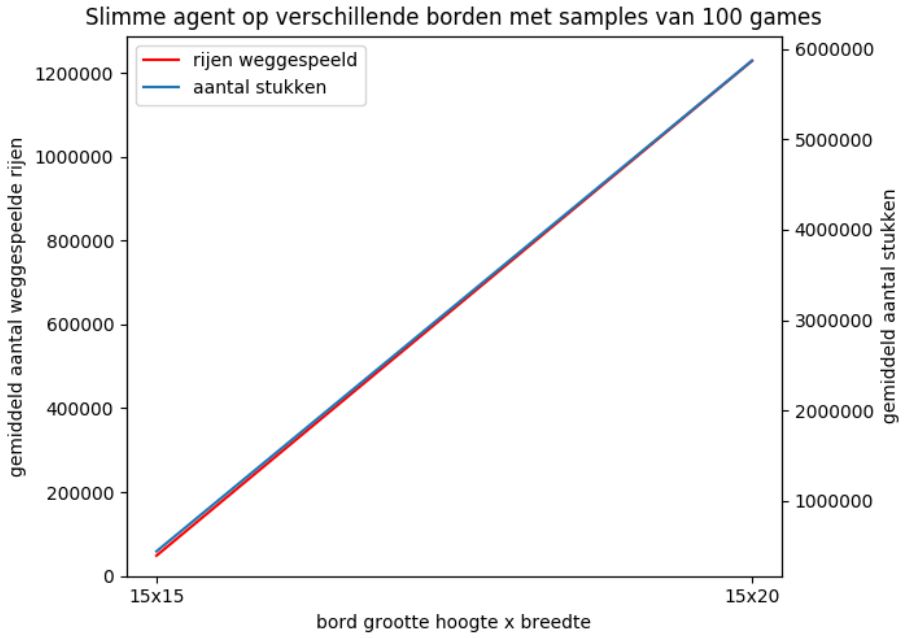
Voor de Monte-Carlo agent met 1000 playouts kan er net zoals bij de agent met 50 playouts geen duidelijk verband worden gelegd tussen het aantal rijen dat wordt weggespeeld, het aantal geplaatste stukken en de toename van het bord. Het aantal weggespeelde rijen en geplaatste stukken is wel beduidend hoger ten opzichte van de Monte-Carlo agent met 50 playouts. Hoe meer playouts de Monte-Carlo agent kan doen hoe beter de beste zet is die hij kan vinden. Na een variabel aantal playouts zal echter de Monte-Carlo agent geen betere beste zet vinden. Alle mogelijke spellen zullen dan gespeeld zijn waardoor een extra playout geen nieuwe inzichten geeft.

Tabel 4: Slimme agent op verschillende bord groottes met 100 games

grootte bord hoogte×breedte	gemiddeld aantal rijen weggespeeld	gemiddeld aantal geplaatste stukken
10×10	68	186
10×15	342	1.307
10×20	1.466	7.366
15×15	48.257	442.115
15×20	1.228.724	5.868.939

Opmerking: voor het 15×15 en het 15×20 bord zijn maar 20 games gerund aangezien deze games per stuk lang duren.





Slimme agent
Bij de slimme agent neemt met de bordgrootte zowel het aantal weggespeelde rijen als het aantal geplaatste stukken toe.

Resultaten
Gekeken naar de resultaten van de experimenten kan worden geconcludeerd dat de slimme agent op alle geteste borden beter speelt dan de Monte-Carlo agent, die op zijn beurt weer beter speelt dan de random agent.

De Monte-Carlo agent probeert alle mogelijke posities van een stuk en speelt dan random het spel uit om de beste zet te bepalen. Het kan zo zijn dat een zet de beste zet leek, maar dat het random afspelen toevallig snel eindigde omdat de random aangewezen stukken niet gunstig waren voor het huidige bord. Deze zet wordt dan niet als beste zet gezien waardoor de Monte-Carlo speler dus niet optimaal speelt. Bij het toenemen van het aantal playouts zullen uitzonderlijke slechte situaties worden geminimaliseerd. Tevens is het met een groter aantal playouts duidelijker wat een goede zet is omdat de gemiddelde scores over meer playouts bepaald worden. Een betere gemiddelde score zou betekenen dat de zet vaker beter eindigt en dus beter is. Echter de Monte-Carlo agent zal voor het bepalen van een zet altijd alle mogelijke zetten doen en daarna random moet afspelen. Hierdoor zullen random stukken vaak op een slechte plek worden geplaatst waardoor de Monte-Carlo agent geen goede afweging kan maken over de invloed van een zet op het toekomstige bord.

De slimme agent neemt voor de overweging van het plaatsen van een stuk de eerder geplaatste stukken mee. De slimme agent maakt gebruik van een tactiek en speelt daardoor veel rijen weg. De slimme agent prioriseert een zet niet alleen op basis van het aantal weggespeelde rijen maar neemt ook andere aspecten van het spel Tetris mee in zijn beslissing. Hierdoor bereid de slimme agent het bord relatief goed voor om goede toekomstige zetten mogelijk te maken.

7 Conclusie

In dit verslag hebben we drie verschillende agenten gemaakt die het spel Tetris spelen. Dit omvat een random agent, een Monte-Carlo agent en een slimme agent die werkt via een score functie.
Gekeken naar de resultaten van de experimenten kan worden geconcludeerd dat de slimme agent op alle geteste borden beter speelt dan de Monte-Carlo agent, die op zijn beurt weer beter speelt dan de random agent.

In een vervolgonderzoek zou gekeken kunnen worden naar het optimaliseren van de parameters voor de slimme agent. In dit project is voornamelijk gebruikt gemaakt van zowel gezond verstand als enkele kleine tests met verschillende samenstellingen van parameters. Om de slimme agent nog beter te laten spelen zouden de waarden van de parameters mogelijk nog geoptimaliseerd kunnen worden. Daarnaast zou gekeken kunnen worden de optimale waardes van de parameters per bord-grootte, uit de tests bleek namelijk dat verschillende waardes van de parameters beter presteerde bij verschillende borden.

Referenties

[1] S.J. Russel, P. Norvig, (2021), Artificial Intelligence A Modern Approach (Fourth edition, Vol. 1), Pearson Education.

[2] W. Kosters,(z.d.), Kunstmatige intelligentie - Programmeeropgave 1 - Tetris, Geraadpleegd op 19 februari 2021, van https://liacs.leidenuniv.nl/~kosterswa/AI/teet2021.html.

[3] Fandom, (z.d.), Flat Stacking - Tetris Wiki - Fandom, Tetris Wiki, Geraadpleegd op 19 februari 2021, van https://tetris.fandom.com/wiki/Flat_Stacking.

[4] Wikipedia-bijdragers, (2020, 1 mei), Tetris - Wikipedia, Geraadpleegd op 19 februari 2021, https://nl.wikipedia.org/wiki/Tetris.

Appendix: Code

Er is gebruik gemaakt van de skeletcode die te vinden is via de website van het college. In de letitfall method zijn er drie regels toegevoegd, dit omvat regel 7, 132 en 133. De code van de toegevoegde en aangepaste methods zien er als volgt uit:

```
1  // let piece fall in position and orientation given
2  // assume it still fits in top rows. 3 rows are added to this function.
3  void Tetris::letitfall (PieceName piece, int orientation, int position) { //
       positions is equal to the move
4    int x[4] = {0};
5    int y[4] = {0};
6    int i;
7    highestPositionMove = 0; //This line is added to the given letitfall method.
8    piececount++;
9    switch (piece) {
10     case Sq: x[0] = position; y[0] = h-2;
11              x[1] = position; y[1] = h-1;
12              x[2] = position+1; y[2] = h-2;
13              x[3] = position+1; y[3] = h-1;
14              break;
15     case LG: switch (orientation) {
16                case 0: x[0] = position+2; y[0] = h-2;
17                        x[1] = position+2; y[1] = h-1;
18                        x[2] = position+1; y[2] = h-1;
19                        x[3] = position; y[3] = h-1;
20                        break;
21                case 1: x[0] = position; y[0] = h-3;
22                        x[1] = position; y[1] = h-2;
```

```
23                          x[2] = position; y[2] = h-1;
24                          x[3] = position+1; y[3] = h-1;
25                          break;
26                    case 2: x[0] = position; y[0] = h-2;
27                          x[1] = position+1; y[1] = h-2;
28                          x[2] = position+2; y[2] = h-2;
29                          x[3] = position; y[3] = h-1;
30                          break;
31                    case 3: x[0] = position; y[0] = h-3;
32                          x[1] = position+1; y[1] = h-1;
33                          x[2] = position+1; y[2] = h-2;
34                          x[3] = position+1; y[3] = h-3;
35                          break;
36                 }//switch
37                 break;
38          case RG: switch (orientation) {
39                    case 0: x[0] = position; y[0] = h-2;
40                          x[1] = position+2; y[1] = h-1;
41                          x[2] = position+1; y[2] = h-1;
42                          x[3] = position; y[3] = h-1;
43                          break;
44                    case 1: x[0] = position; y[0] = h-3;
45                          x[1] = position; y[1] = h-2;
46                          x[2] = position; y[2] = h-1;
47                          x[3] = position+1; y[3] = h-3;
48                          break;
49                    case 2: x[0] = position; y[0] = h-2;
50                          x[1] = position+1; y[1] = h-2;
51                          x[2] = position+2; y[2] = h-2;
52                          x[3] = position+2; y[3] = h-1;
53                          break;
54                    case 3: x[0] = position+1; y[0] = h-3;
55                          x[1] = position+1; y[1] = h-1;
56                          x[2] = position+1; y[2] = h-2;
57                          x[3] = position; y[3] = h-1;
58                          break;
59                 }//switch
60                 break;
61          case LS: switch (orientation) {
62                    case 0: x[0] = position+1; y[0] = h-2;
63                          x[1] = position+1; y[1] = h-1;
64                          x[2] = position+2; y[2] = h-2;
65                          x[3] = position; y[3] = h-1;
66
67                          break;
68                    case 1: x[0] = position; y[0] = h-3;
69                          x[1] = position; y[1] = h-2;
70                          x[2] = position+1; y[2] = h-1;
71                          x[3] = position+1; y[3] = h-2;
72                          break;
73                 }//switch
74                 break;
75          case RS: switch (orientation) {
76                    case 0: x[0] = position+1; y[0] = h-2;
77                          x[1] = position+1; y[1] = h-1;
78                          x[2] = position+2; y[2] = h-1;
79                          x[3] = position; y[3] = h-2;
80                          break;
81                    case 1: x[0] = position+1; y[0] = h-3;
82                          x[1] = position; y[1] = h-2;
83                          x[2] = position+1; y[2] = h-2;
```

```
84                          x[3] = position; y[3] = h-1;
85                          break;
86                }//switch
87                break;
88        case I : switch (orientation) {
89                  case 0: x[0] = position; y[0] = h-1;
90                          x[1] = position+1; y[1] = h-1;
91                          x[2] = position+2; y[2] = h-1;
92                          x[3] = position+3; y[3] = h-1;
93                          break;
94                  case 1: x[0] = position; y[0] = h-4;
95                          x[1] = position; y[1] = h-3;
96                          x[2] = position; y[2] = h-2;
97                          x[3] = position; y[3] = h-1;
98                          break;
99                }//switch
100               break;
101       case T : switch (orientation) {
102                 case 0: x[0] = position+1; y[0] = h-2;
103                         x[1] = position; y[1] = h-1;
104                         x[2] = position+1; y[2] = h-1;
105                         x[3] = position+2; y[3] = h-1;
106                         break;
107                 case 1: x[0] = position; y[0] = h-3;
108                         x[1] = position; y[1] = h-2;
109                         x[2] = position; y[2] = h-1;
110                         x[3] = position+1; y[3] = h-2;
111                         break;
112                 case 2: x[0] = position; y[0] = h-2;
113                         x[1] = position+1; y[1] = h-2;
114                         x[2] = position+2; y[2] = h-2;
115                         x[3] = position+1; y[3] = h-1;
116                         break;
117                 case 3: x[0] = position+1; y[0] = h-3;
118                         x[1] = position+1; y[1] = h-2;
119                         x[2] = position+1; y[2] = h-1;
120                         x[3] = position; y[3] = h-2;
121                         break;
122               }//switch
123               break;
124   }//switch
125   while ( y[0] > 0 && !board[y[0]-1][x[0]]
126           && !board[y[1]-1][x[1]] && !board[y[2]-1][x[2]]
127           && !board[y[3]-1][x[3]] )
128     for ( i = 0; i < 4; i++ )
129       y[i]--;//Move it down
130   for ( i = 0; i < 4; i++ ){
131     board[y[i]][x[i]] = true;
132     if (y[i] > highestPositionMove) //This line is added to the given letitfall
          method.
133       highestPositionMove = y[i]; //Store the row where the highest part of the
            piece can be found. This line is added to the given letitfall method.
134   }
135   clearrows ( );
136 }//Tetris::letitfall
137
138 // do a MC move for piece.
139 void Tetris::doMCmove (PieceName piece) {
140   int bestmove = 0;
141   int mostrowscleared = 0;
142   int mostmoves = 0;
```

```
143     for (int move = 0; move < possibilities(piece); move++){
144     int totalrowscleared = 0;
145     int totalmoves = 0;
146     //Return the rows cleared and the number of moves.
147     for (int x = 0; x < theplayouts; x++){
148       Tetris copy = *this;
149       copy.dothemove(piece,move);
150       copy.playrandomgame(false);
151       totalrowscleared += copy.rowscleared;
152       totalmoves += copy.piececount;
153      }//for
154      if (totalrowscleared > mostrowscleared){
155         mostrowscleared = totalrowscleared;
156         bestmove = move;
157       }else if (totalrowscleared == mostrowscleared){
158         if (totalmoves > mostmoves){
159           mostmoves = totalmoves;
160           bestmove = move;
161         }
162       }
163   }//for
164   dothemove(piece, bestmove);
165 }//Tetris::doMCmove
166
167 //Do a smart move
168 void Tetris::dosmartmove (PieceName piece) {
169   int bestmove = 0;
170   int bestScore = INT32_MIN;
171   int currentclosedspaces = 0;
172   for (int move = 0; move < possibilities(piece); move++){
173     Tetris copy = *this;
174     int premove_openshaftspots = copy.countshaft();
175     copy.dothemove(piece,move);
176     int aftermove_openshaftspots = copy.countshaft();
177     currentclosedspaces = copy.numberofclosedspaces();
178     int currentscore = 0;
179     if (copy.endofgame() == false){
180       /*calculate the score of a move based on certain properties
181       that are changed by doing the move.*/
182       currentscore += copy.rowscleared;
183       currentscore += 1 * (h - copy.highestPositionMove);
184       currentscore -= 4 * currentclosedspaces;
185       currentscore += 1 * (premove_openshaftspots-aftermove_openshaftspots);
186       if (currentscore > bestScore){
187         bestScore = currentscore;
188         bestmove = move;
189       }
190     }
191   }
192     dothemove(piece, bestmove);
193 }
194
195 //Check the number of shafts there are in a board. A shaft is defined by an one
      space wide opening.
196 int Tetris::countshaft(){//
197 int openspaces = 0;
198 for (int column = 0; column < w; column++){
199   for (int row = h-1; row >=0 ; row--){
200     if (board[row][column]) break;
201     if (//Checks if a space is enclosed from the right and left.
202        (board[row][column+1] && board[row][column-1]) ||
```

```
203         (board[row][column+1] && column == 0) ||
204         (board[row][column-1] && column == w-1)){
205            openspaces+=1;
206            while (row < h && row > 0){//Counts the openspaces of a shaft when a
                   shaft is found.
207            if (board[row-1][column]){break;}
208            row--;
209            openspaces+=1;
210         }
211       }
212     }
213   }
214     return openspaces;
215   }
216
217   //Find the number of spaces which are not accesible by a piece at the moment.
218   int Tetris::numberofclosedspaces(){
219     int totalclosedpositions = 0;
220     for (int row = 0; row < h; row++){
221       for (int column = 0; column < w; column++){
222         if (!board[row][column]){//If there is no piece
223         for (int a = row; a < h-1; a++){//Looks up to see open space is closed off.
224           if (board[a+1][column]){
225             totalclosedpositions = totalclosedpositions + 1;//Counts all the spaces
                    in the column that are closed off.
226             break;
227           }
228         }
229       }
230      }
231     }
232     return totalclosedpositions;
233   }
234
235   //Plays a smart game.
236   void Tetris::playsmartgame (bool print) {
237     PieceName piece;
238     int nr, emp;
239     bool therow[wMAX];
240     if ( print )
241       displayboard ( );
242     while ( ! endofgame ( ) ) {
243       getrandompiece (piece);    // obtain random piece
244       dosmartmove(piece);          // and drop it using MC
245       if ( print ) {
246         displayboard ( );        // print the board
247         toprow (therow,nr,emp);  // how is top row?
248         if ( nr != -1 )
249           cout << "Top row " << nr << " has " << emp << " empties" << "." << endl;
250       }//if
251     }//while
252   }//Tetris::playMCgame
```