Project Logisch Programmeren

Implementeren van het spel Con-Tac-Tix

Victor Matthijs

Juni 2020

Inhoudsopgave

1 Introductie **1**

2 Hoe zit het spel in elkaar? **2**

2.1 Parser Module 3

2.2 Game Module 3

2.3 Board Module 4

2.4 SVG Module 6

3 Het Con-Tac-Tix Algoritme **6**

4 Wat kan beter? **7**

5 Conclusie **7**

Appendices **8**

Referenties **11**

1 Introductie

Dit jaar is het project voor Logisch programmeren het implementeren van het project Con-Tac-Tix. Dit spel wordt gespeeld op een ruitvormig bord, waar elke tegel wordt voorgesteld door een hexagonaal. De borden waarmee we spelen kunnen verschillen van dimensies, maar het meest voorkomende bord is 11 x 11. De zijdes van het bord worden verdeeld tussen 2 spelers, deze 2 spelers kunnen elk een kleur kiezen en krijgen dan ofwel boven- en onderzijde of rechter- en linkerzijde. Elke speler mag om de beurt één tegel op het board plaatsen, dit mag om het even waar zijn op het board, maar niet op plaatsen waar er al een kleur ligt. Het spreekt voor zich dat een speler enkel zijn eigen kleur mag leggen op het bord.
Het doel van het spel is om als eerste een pad te vormen die van de ene zijde naar de andere zijde van het bord loopt. Men moet dus de 2 zijdes met hun kleur verbinden. Men moet er hier dus voor zorgen dat elke tegel op hun pad een andere tegel aan een van de zijdes van de hexagoon raakt, anders heeft men geen pad en kan men dus ook niet winnen.

Het spel wordt geïmplementeerd als een input-output programma, we aanvaarden dus een invoerFile met een bepaalde structuur. Deze file zal geparsed worden naar een interne bord voorstelling, waarna het algoritme de beste zet zal terug geven.

Men kan het programma oproepen aan de hand van het volgende commando:

```
$ cat invoerFile | swipl -f none -t halt -g main -q main.pl
```

Het programma aanvaardt verschillende argumenten, deze worden hieronder toegelicht:

- (No Arguments) : Wanneer het commando wordt uitgevoerd zonder argumenten mee te geven, dan zal het algoritme de beste zet teruggeven. Wanneer men in één stap kan winnen, dan zal het programma ook de winnende zet teruggeven.
- TEST: Het test argument zorgt ervoor dat het programma alle mogelijke zetten zal teruggeven vanuit de beginpositie van het bord. Al deze borden worden naar de standaar output geschreven.
- SVG: Het svg argument zorgt ervoor dat de oplossing van het algoritme terug wordt gegeven via SVG's, dit kunnen één of meerdere borden zijn.

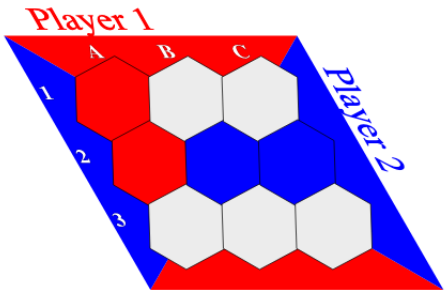
Bovenstaande argumenten kunnen met elkaar worden gebruikt.

2 Hoe zit het spel in elkaar?

Het spel begint zoals eerder is aangegeven, wanneer er een juist commando wordt ingegeven in de terminal. De eerste stap die moet gebeuren is het parsen van de $invoerFile$, deze file heeft een bepaalde structuur zoals weergegeven in de opgave op Ufora. De parser zelf zal gebruik maken van Prolog DCG (Prolog Definite Clause Grammer).
De manier waarop het spel intern voorgesteld wordt is als het volgt: De parser zal de invoerFile overlopen en zal alle verschillende velden afhandelen. Er wordt binnen het programma met twee voorstellingen gewerkt, één daarvan zal het spel zelf voorstellen, dit ziet eruit als volgt:

$$game(Size, Turn, State, Ori, Tiles)$$

De file die instaat voor het parsen van de input zal deze Game voorstelling ook terug geven aan de Main regel. Deze Game voorstelling zal dan daarna gebruikt worden om het effectieve bord voor te stellen. Ik heb ervoor gekozen om het bord als een matrix voor te stellen (List of Lists). Hier zullen de lege plaatsen van het bord worden voorgesteld door een 0, en de andere plaatsen worden ingenomen door de naam van de kleur. Aan de hand van deze 2 voorstelling is de rest van het programma ook opgebouwd, alles start vanuit de main module. Hier wordt er gekeken naar welke argumenten meegegeven werden met het commando.
Een voorbeeld van deze voorstelling kan men hieronder zien:



$$M = \begin{bmatrix} red & 0 & 0 \\ red & blue & blue \\ 0 & 0 & 0 \end{bmatrix}$$

De rest van het programma wordt verder opgedeeld in verschillende modules, naast de main module is er ook nog een game, board, minimax_ai, parser en SVG module. Deze opsplitsing zorgt ervoor dat het programma duidelijk en

gestructureerd blijft. Al deze modules worden ook ondersteund doormiddel van commentaar bij elke (logische) regel. Wat volgt zijn hele kleine beschrijvingen van wat elke module precies doet.

2.1 Parser Module

Dit is de eerste module die wordt opgeroepen vanuit de main. Deze module zal ook maar één regel/predicaat exporteren via het PublicList argument. Alles start vanuit de parse regel op lijn 45. Deze regel zal deel van de invoerFile als een apart *game_part* behandelen. Ik heb de code zo geschreven dat de verschillende *game_parts* in verschillende volgordes kunnen voorkomen. Hier wordt ook telkens rekening gehouden met de mogelijke whitespaces en newlines die kunnen voorkomen in de invoerFile.

```
42 % This is the main rule that will be called to parse a file
43 % The variable Game will unify with a rule representing a game
44 % this will also try to satisfy the subgoal: game_parts
45 parse(game(Size, Turn, State, Ori, Tiles)) -->
46     game_parts(game(Size, Turn, State, Ori, Tiles)), !.
47
48 % This following rule game_parts will try to satisfy all
49 % it's subgoals, this rule is recursive. It is made recursive
50 % so we make sure we get all the different structures from
51 % the input file to construct our game
52 game_parts(game(Size, Turn, State, Ori, Tiles)) -->
53     game_part(game(Size, Turn, State, Ori, Tiles)),
54     game_parts(game(Size, Turn, State, Ori, Tiles)).
55 game_parts(game(Size, Turn, State, Ori, Tiles)) -->
56     game_part(game(Size, Turn, State, Ori, Tiles)).
```

Zoals eerder aangegeven maakt deze parser module gebruik van Prolog Definite Clause Grammer. Ik heb hier ook zoveel mogelijk gebruik proberen maken van de basic DCG regels die beschikbaar zijn door Prolog. De bedoeling van deze parser module is dus om de invoerFile om te zetten naar een Game representatie zoals eerder aangehaald is. Deze representatie zal dan gebruikt worden bij het verdere verloop van het spel.

2.2 Game Module

De Game module is één van de twee modules die de representatie van het spel afhandelen, zoals de naam al prijs geeft zal deze module dus alle regels en

predicaten bevatten voor een game. Wat volgt zijn een paar belangrijke aspecten die in deze module aanwezig zijn.

- Het eerste belangrijke wat deze module bevat zijn regels die ervoor zorgen dat de argumenten van onze game toegankelijk zijn tijdens het minimax algoritme. Deze verschillende regels bevinden zich op lijn 16-72.
- Lijn 73-125 bevat de logica voor het afprinten van de games naar de gebruiker. Al deze regels zullen worden opgeroepen wanneer er geen SVG argument is meegegeven. Hier zal dus de regel *print_all_games* elke game representatie die het krijgt afprinten naar standaard output/terminal.

```
94  % print_all_games(+Games)
95  % print all Games from the list to the output
96  print_all_games([]).
97  print_all_games([Game]) :-
98      print_output_one_game(Game).
99  print_all_games([Game|Rest]) :-
100     print_output_one_game(Game),
101     print(~), nl,
102     print_all_games(Rest).
```

- Het laatste deel van deze module bevat regels voor het maken van nieuwe game representatie uitgaande van een bord of oude game. De regel *create_game_from_board* wordt opgeroepen vanuit de main nadat het minimax algoritme klaar is. Deze regels zijn te vinden op lijn 126-200.

```
147 % create_next_games(+Game, +OldBoard, +Columns, +Boards,
    ↪  -Games)
148 % create a list of games from a list of boards.
149 create_next_games(_, _, _, [], []):- !.
150 create_next_games(Game, OldBoard, Columns,
    ↪  [Board|RestBoards], [NewGame|RestGames]) :-
151     create_game_from_board(Game, OldBoard, Board, Columns,
        ↪  NewGame),
152     create_next_games(Game, OldBoard, Columns, RestBoards,
        ↪  RestGames).
```

2.3 Board Module

Een andere module die wordt opgeroepen vanuit de main is de Board module. Deze module beschikt over alle regels die te maken hebben met het Con-Tac-Tix spelbord en de spelregels er rond. Na het parsen van de invoerFile zal de *construct_board_from_tiles* worden opgeroepen vanuit de main. Deze zal een list van tiles meegeven als argument, en het zal een matrix vormen met alle tiles

op de juiste plaats. Deze matrix is dus de voorstelling van het spelbord.
Bij het schrijven van deze regels heb ik mij deels gebaseerd op de volgende code die ter beschikking stond op GitHub.[1, 2]

```
15 % construct_board_from_tiles(+Rows, +Columns, +Tiles, -Board)
16 % This first rule will be exported in the module board
17 % it construct a matrix with the tile at the correct position,
18 % these positions where specified in the parsing fase.
19 construct_board_from_tiles(Rows, Columns, Tiles, Board) :-
20     make_empty_matrix(Rows, Columns, Board1),
21     fill_matrix_with_tiles(Rows, Columns, Tiles, Board1, Board).
```

Daarnaast bevat de board module ook nog heel wat andere belangrijke regels en predicaten die helpen bij het spelen van het spel. Een daarvan is de regel die kijkt of een huidige speler gewonnen heeft op het bord. Dit deel van de code is gebaseerd op het volgende algoritme. Om te checken of een nieuw bord een gewonnen bord is, worden de volgende stappen ondernomen:

- Het algoritme start vanaf een paalde nieuwe tile op het bord. We kijken voor een bepaalde kleur of deze een pad heeft kunnen vormen van de bovenkant van het bord naar de onderkant. Wanneer de speler aan zet is die van links naar rechts speelt, dan draaien we het bord tijdelijk om een pad van boven naar onder te vinden.

```
98  % did_current_player_win(+NewTile, +MaxColumn, +MaxRow,
    ↪  +Board, +CurrentColor)
99  % This is the start of the algorithm to check if a player
    ↪  has won
100 did_current_player_win(NewTile, MCol, MRow, Board, Color) :-
101     find_top_hulp([NewTile], [], MCol, MRow, Color, Board,
        ↪  _, path),
102     find_bottom_hulp([NewTile], [], MCol, MRow, Color,
        ↪  Board, _, path).
```

- Om een pad te zoeken van boven naar onder dat door de nieuwe tile passeert, zoeken we eigenlijk zowel een pad vanaf de nieuwe tile naar boven als een pad van de nieuwe tile naar beneden. Dit doen we door gebruik te maken van de functies *find_top_hulp* en *find_bottom_hulp*. (dit stuk code is te vinden in Appendix A).

- De twee regels die zojuist vermeld zijn geweest, werken dan weer op hun eigen manier om een pad te vinden van een tile naar een rand van het bord. Hier moet men echter rekening houden dat het werkelijke spel gespeeld wordt op een hexagonaal bord, terwijl de representatie van het bord in de code wordt gedaan door een matrix. Hierdoor moeten een aantal conversies gebeuren, maar dit stelt geen al te grote problemen voor.

De conversies tussen een hexagonaal bord en een vierkant bord/matrix worden getoond in Appendix B. De algemene werking is dat er uit een bepaalde positie 4 buurposities worden gecreëerd, en deze 4 buurposities kunnen een mogelijke extensie zijn van het pad dat het algoritme aan het zoeken is.

2.4 SVG Module

Deze laatste module kan heel kort worden uitgelegd en heeft niet zoveel speciale functionaliteiten. Het enige doel dat deze module heeft is het uitschrijven ven een SVG representatie van het bord na het uitvoeren van de juiste algoritmes erop. De vorm van de gebruikte SVG's is vooral gebaseerd op de SVG's die zijn teruggevonden in de projectopgave op Ufora.

3 Het Con-Tac-Tix Algoritme

Wat volgt is een kleine toelichting over hoe het Con-Tac-Tic algoritme in elkaar zit. Het AI gedeelte van dit programma wordt enkel uitgevoerd wanneer er geen TEST argument is meegegeven bij het oproepen van het programma. Het algoritme is een versie die gebaseerd is op het minimax algoritme, maar dan geïmplementeerd met een maximum zoekdiepte van 4 zoals aangegeven in de opgave.
De module min_max_ai zelf bevat twee implementatie voor het minimax algoritme. De eerste implementatie is gebaseerd en opgesteld met ondersteuning van volgende bron [3]. Deze eerste versie bevat geen alpha-beta pruning waardoor sommige borden niet konden worden afgehandeld door het algoritme. De zoekruimte bij die bepaalde borden was gewoon te groot. Dus dit was zeker een reden om aan een andere implementatie te werken die de zoekruimte in de minimax zoekboom kan verkleinen. (Deze eerste implementatie is in block comments toegevoegd helemaal onderaan deze module.)
Deze tweede implementatie is dan gebaseerd op voorbeelden uit de les en verschillende online bronnen. [4, 5] Hier wordt Alpha-Beta pruning toegepast waardoor de zoekruimte met een groot stuk kan verkleind worden. Ook hier werk ik met een maximum zoekdiepte van 4, maar dit zou men nog zelf kunnen aanpassen indien men dieper wil zoeken en tot een betere zet wil komen. Om ervoor te zorgen dat alpha-beta pruning wel degelijk werkt, heb ik ook nog een heuristiek moeten maken. Deze heuristiek is uiteindelijk gebaseerd op het aantal paden dat een bepaalde speler op dat moment al heeft en hoe lang deze zijn. Dus een speler met een langer pad van boven naar beneden zal een betere heuristiek hebben dan een speler met kortere paden naar beneden. Deze heuristiek wordt echter wel aangepast wanneer de tegenspeler een winnende zet kan doen. Op

dat moment zal de huidige speler deze winnende zet moeten zien te voorkomen.

4 Wat kan beter?

Als we het programma van begin tot eind overlopen dan kunnen sommige delen van de code toch nog in aanmerking komen voor verbetering:

- Het eerste deel is de parser, hier was het in het begin wat wennen om DCG's onder de knie te krijgen. Al met al is dit deel van de code toch redelijk goed gelukt, het enige wat mss wat beter kan is het meer gebruiken van de module DCG/basics.
- Zoals eerder aangehaald werkt de module board met een matrix. Hier ben ik niet 100% tevreden over hoe ik de regels voor het opbouwen van een bord heb geschreven. Het probleem dat men nu stelt, is dat elk bord/matrix eerst wordt omgezet naar één lange lijst om dan alle tiles daarin toe te voegen. Daarna wordt die lijst weer omgevormd tot een matrix. Dit kan waarschijnlijk op een efficiëntere manier.
- Bij de module game wordt er een regel gebruikt die nagaat of een bepaalde speler heeft gewonnen met zijn nieuwste zet of niet. Dit kleine algoritme had misschien wat algemener kunnen gecodeerd worden. Zodat het makkelijker is om dit te hergebruiken in een latere fase van het programma.
- Algemeen was het misschien gemakkelijker en duidelijker geweest had ik overal gewoon de game representatie doorgegeven als argument. Dit is nu anders, met het verschil dat vaak het bord en dan nog enkele andere argumenten worden doorgegeven.

5 Conclusie

Het implementeren van het spel Con-Tac-Tix in prolog was vooral veel opzoekingswerk in het begin. De lessen geven een goede basis, maar om aan het project te beginnen is het zeker wel nodig om zelf op zoek te gaan op het internet naar andere voorbeelden en tutorials. Het deel waar er het meeste tijd is ingekropen was het implementeren en testen van de AI voor het spel. Het minimax algoritme is al iets dat we gezien hebben in het vak artificiële intelligentie, maar om het om te zetten naar prolog code is toch een ander verhaal. Het vergt wat tijd om in de minset te komen van prolog, wat op zich weer een heel ander denken is dan andere programmeer talen.

Uiteindelijk is het me toch goed gelukt om een degelijk project af te leveren, waar alle vereiste doelen werken zoals in de opgave staat beschreven.

Appendices

Appendix A

```
104 % find_top_hulp(+ListOfTiles, +History, +MaxColum, +MaxRow,
    ↪ +CurrentColor, +Board)
105 % This rule will help to construct a path to the top of the
    ↪ board, it is a
106 % recursive rule, we use a history so we don't get into a loop.
107 find_top_hulp([], His, _, _, _, _, His, nopath).
108 find_top_hulp([NewTile|Rest], His, MCol, MRow, Color, OldBoard,
    ↪ FinalHis, FindTop) :-
109     nth1(1, NewTile, SCol),
110     nth1(2, NewTile, SRow),
111     append([NewTile], His, NewHis),
112     (find_path_to_top(SCol, SRow, MCol, MRow, Color, OldBoard,
        ↪ NewHis, FinalHis, FindTop), ! ;
113     find_top_hulp(Rest, NewHis, MCol, MRow, Color, OldBoard,
        ↪ FinalHis, FindTop)).
114
115 % find_bottom_hulp(+ListOfTiles, +History, +MaxColum, +MaxRow,
    ↪ +CurrentColor, +Board)
116 % This rule is almost the same, but here we try to construct a
    ↪ path from the current tile
117 % to the bottom of the board
118 find_bottom_hulp([], His, _, _, _, _, His, nopath).
119 find_bottom_hulp([NewTile|Rest], His, MCol, MRow, Color,
    ↪ OldBoard, FinalHis, FindBottom) :-
120     nth1(1, NewTile, SCol),
121     nth1(2, NewTile, SRow),
122     append([NewTile], His, NewHis),
123     (find_path_to_bottom(SCol, SRow, MCol, MRow, Color, OldBoard,
        ↪ NewHis,FinalHis, FindBottom), ! ;
124     find_bottom_hulp(Rest, NewHis, MCol, MRow, Color, OldBoard,
        ↪ FinalHis, FindBottom)).
```

Appendix B

```
155 % generate_top_positions(+StartCol, +StartRow, +MaxColumn,
    ↪ -SolutionTiles)
156 % This will generate all surrounding tiles that lay above or next
    ↪ to the current tile
157 % It will generate all the N tiles below
158 %   |   | N | N |
159 %   | N | x | N |
160 %   |   |   |   |
161 generate_top_positions(SCol, SRow, MCol, Solution) :-
162     findall(Position, next_top_position(SCol, SRow, MCol,
        ↪ Position), Solution).
163
164 % next_top_position(+Column, +Row, +MaxColumn,
    ↪ -[NewColumn,NewRow])
165 next_top_position(Col, Row, _,  [Col, NewRow])    :- Row > 1,
    ↪ NewRow is Row - 1.
166 next_top_position(Col, Row, MC, [NewCol, NewRow]) :- Row > 1, Col
    ↪ < MC, NewCol is Col + 1, NewRow is Row - 1.
167 next_top_position(Col, Row, MC, [NewCol, Row])    :- Col < MC,
    ↪ NewCol is Col + 1.
168 next_top_position(Col, Row, _,  [NewCol, Row])    :- Col > 1,
    ↪ NewCol is Col - 1.
169
170 % generate_bottom_positions(+StartCol, +StartRow, +MaxColumn,
    ↪ +MaxRow, -SolutionTiles)
171 % This will generate all surrounding tiles that lay under or next
    ↪ to the current tile
172 % It will generate all the N tiles below
173 %   |   |   |   |
174 %   | N | x | N |
175 %   | N | N |   |
176 generate_bottom_positions(SCol, SRow, MCol, MRow, Solution) :-
177     findall(Position, next_bottom_position(SCol, SRow, MCol,
        ↪ MRow, Position), Solution).
178
179 % next_bottom_position(+Column, +Row, +MaxColumn, +MaxRow,
    ↪ -[NewColumn, NewRow])
180 next_bottom_position(Col, Row, _, MR, [Col, NewRow])    :- Row <
    ↪ MR, NewRow is Row + 1.
181 next_bottom_position(Col, Row, _, MR, [NewCol, NewRow]) :- Row <
    ↪ MR, Col > 1, NewCol is Col - 1, NewRow is Row + 1.
182 next_bottom_position(Col, Row, MC, _, [NewCol, Row]) :- Col < MC,
    ↪ NewCol is Col + 1.
```

```
183 next_bottom_position(Col, Row, _, _, [NewCol, Row])  :- Col > 1,
    ↪  NewCol is Col - 1.
```

Referenties

[1] https://github.com/perkola/matrix/blob/master/matrix.pl *Understanding a Prolog Matrix*

[2] https://stackoverflow.com/questions/5807455/matrix-operations-prolog *Transpose of a matrix*

[3] https://github.com/jaunerc/minimax-prolog *Minimax Versie 1*

[4] http://colin.barker.pagesperso-orange.fr/lpa/tictac.htm *Minimax + alpha beta pruning*

[5] https://www.cpp.edu/ jrfisher/www/prolog_tutorial/5_3.html *Minimax + alpha beta pruning 2*