Project Programmeertalen

Jarre Knockaert

Inhoudsopgave

1 Introductie

Het project werd gemaakt op het besturingssysteem Windows 10. Alle programmacode werd gecompileerd met behulp van Mozart 2.

2 Code

2.1 Algemeen

De volgende namen voor variabelen worden doorheen de volledige code gebruikt:

Type stelt het type van een coördinaat in het bord voor. Deze heeft als waarde één van de volgende atomen: white, black of empty. Dit zijn tegels met resp. een witte pion, zwarte pion of een vakje zonder pion.

Color stelt de kleur voor van een speler of van een pion. Deze heeft als waarde white of black.

Board is een 2-dimensionale lijst waarin elke lijst een rij voorstelt in het bord. De rij is een lijst waarvan elke waarde een `Type` is.

Coord stelt een coördinaat in het bord voor. Dit is een record met twee features: row en col. Dit zijn elke integers.

Pawn stelt een `Coord` voor waarbij de waarde van het coördinaat in het bord een **Color** is.

Tile is een record met 3 features: row , col en type. Dit beschrijft een coördinaat en het type van deze coördinaat.

Move stelt een zet voor. Dit is een record met twee features: startcoord en stopcoord. Dit zijn beide een `Coord` en stellen het startcoördinaat en stopcoördinaat voor van de zet.

Direction stelt een richting voor. Deze is afhankelijk van de speler. De zwarte speler heeft als direction 1 en de witte speler heeft als direction -1.

2.2 Bord

Het bestand `board.oz` bevat alle functies om te interageren met het spelbord en bevat enkele basisfunctionaliteit:

CreateBoard maakt een nieuw spelbord aan met de opgegeven dimensie's.

SubmitMove bevestigt een `Move`.

RemovePawn verwijdert een `Pawn`.

BoardToTiles zet de 2-dimensionele lijst van `Type` om naar een 2-dimensionele lijst van `Tile`.

FilterTiles filtert een lijst van Tiles op basis van een `Type`.

IsInBoundaries controleert indien een `Coord` in het bord ligt.

GetNeighbouringTiles geeft de naburige `Tiles` van een `Coordinaat` op basis van een `Direction`.

Dit is geen volledige lijst, overige functionaliteit is triviaal of zijn slechts hulpfuncties.

2.3 Speler

Het bestand `player.oz` bevat slechts één functie zichtbaar naar de buitenwereld (een geëxporteerde functie), namelijk **CreatePlayer**. Deze functie aanvaardt de poort van de scheidsrechter en de kleur van deze speler. De functie maakt op zich een poort aan waarnaar berichten kunnen gezonden worden. De speler handelt in een aparte draad sequentieel alle berichten af. Deze berichten worden verstuurd vanuit de scheidsrechter. De speler kan de volgende berichten afhandelen:

chooseSize() : De speler kiest de grootte van het bord. Als grootte wordt een standaard 5x5 bord genomen.

doMove(GameBoard Strikes) : De speler berekent zijn volgende zet. De speler ontvangt hiervoor een `Board` en het aantal foutieve zetten deze beurt. Deze laatste waarde wordt verder niet gebruikt.

removePawn(GameBoard) : De speler verwijdert één van zijn pionnen.

chooseK(GameBoard) : De speler kiest een waarde voor K, het aantal te verwijderen pionnen door iedere speler. Hiervoor wordt de maximale waarde gekozen om zoveel mogelijk plaats te creëren om de finishlijn te bereiken. Dit heeft als neveneffect dat de overige speler ook eenvoudiger de overkant kan bereiken, maar het algoritme van removePawn probeert dit te verhinderen.

De afhandeling van `doMove` en `removePawn` verdienen wat extra uitleg.

2.3.1 doMove

Het berekenen van de volgende zet gebeurt op basis van een heuristiek. De waarde van elke mogelijke zet wordt berekend, en de zet met de hoogste waarde wordt uiteindelijk gekozen. Het algoritme voor het bepalen van de waarde van een `Move` wordt beschreven in algoritme 1.
Het algoritme aanvaardt de move, het bord en het kleur van de speler. Het algoritme is opgebouwd uit 2 functies: `GetValueMove` en `SimulateMove`. Het algoritme begint met de functie GetValueMove. Op lijn 9 wordt een multiplier bepaald indien de move een pion kan veroveren. Deze multiplier is groter indien de pion dicht is bij de start, aangezien de vijandige pion dan dichter bij de

finish is, en dus het veroveren van deze pion belangrijk is. Op lijn 12 wordt een waarde `CanBlock` bepaald die deel zal uitmaken van de uiteindelijke `Value`. Als de pion een vijandige pion kan blokkeren door deze zet, zal deze waarde niet nul zijn. Dit betekent waarschijnlijk dat de pion zelf ook zal geblokkeerd zijn. Deze waarde wordt vermenigvuldigd met Multiplier in het kwadraat, zodat deze waarde slechts van groot belang is indien de vijandige pion het einde nadert. Op lijn 14 wordt de uiteindelijke waarde van de move mits één wijziging berekend. De waarde zal groot zijn indien de pion het einde nadert, hiernaast worden `Multiplier` en `CanBlock` toegevoegd aan de formule om te verhinderen dat de vijand de finish zou bereiken.
Als laatste wordt gecontroleerd welk effect de huidige move heeft op de waarde van de vijandige moves. Indien de waarde van de vijandige moves gemiddeld stijgt door het plaatsen van de huidige move, moet de waarde van de huidige move dalen. Om de gemiddelde waarde van de vijandige moves te berekenen wordt de functie `SimulateMove` opgeroepen. Die functie `SimulateMove` roept opnieuw `GetValueMove` op om de waardes van de vijandige moves te berekenen onder hypothese dat de move van de speler bevestigd is. Elke `GetValueMove` zal opnieuw een `SimulateMove` opnieuw oproepen. Om deze infinite loop van simulaties te vermijden wordt een diepte en maximale diepte bijgehouden die bepaalt hoe diep de simulaties mogen worden uitgevoerd. Als maximale diepte gebruik ik 2 om het rekenwerk aanvaardbaar te houden. De wijziging aan de `Value` door `SimulateMove` daalt lineair met de diepte aangezien diepere simulaties minder beduidend zijn voor de huidige move.

Algoritme 1 Heuristisch algoritme om de waarde van een move te berekenen

Input: `Board`, `Move`, `Color`

Output: Value: de waarde van de `Move`, dit is een float.

```
1: Value ← GetValueMove(Board Move Color 0 2)
2:
3: function GetValueMove(GameBoard Move Color Depth MaxDepth)
4:     N ← aantal rijen in het bord
5:     M ← afstand naar de finish
6:     Multiplier ← 1
7:     CanBlockValue ← 0
8:     if de move kan een pion veroveren then
9:         Multiplier ← 1 + 2 * M/N
10:    end if
11:    if de move kan een pion blokkeren then
12:        CanBlockValue ← N/4
13:    end if
14:    Value ← Multiplier * (N − M + Multiplier * CanBlockValue)
15:    if Depth ≤ MaxDepth then
16:        SimulationValue ← SimulateMove(GameBoard Move Color
       (Depth+1) MaxDepth) ÷ (Depth+1)/2
17:        if Depth mod 2 then
18:            Value −= SimulationValue
19:        else
20:            Value += SimulationValue
21:        end if
22:    end if
23:    return Value
24: end function
25:
26: function SimulateMove(GameBoard Move Color Depth MaxDepth)
27:    UpdatedGameBoard ← Gameboard met de gegeven Move bevestigd
28:    EnemyMoves ← de mogelijke vijandige moves.
29:    N ← Aantal EnemyMoves
30:    TotalValue ← 0
31:    EnemyColor ← het kleur van de vijand van speler met kleur Color.
32:    for all EnemyMove in EnemyMoves do
33:        Value ← GetValueMove(UpdatedGameBoard Move EnemyColor
       Depth MaxDepth)
34:        TotalValue += Value
35:    end for
36:    return TotalValue/N
37: end function
```

2.3.2 removePawn

De keuze voor het verwijderen van een pion volgt een eenvoudige heuristiek. De speler tracht pionnen te verwijderen enkel en alleen als de huidige kolom nog een vijandige pion bevat. Dit heeft als doel het verhogen van de kans dat een vijandige pion kan veroverd worden en het verlagen van de kans dat de pion geblokkeerd zal worden.

2.4 Regels

Het bestand `rules.oz` houdt alle regels van het spel bij. Het bestand exporteert 2 functies:

GetValidMoves geeft een lijst terug van alle geldige moves en maakt hiervoor gebruik van de volgende functie.

IsValidMove controleert indien een `Move` voldoet aan alle regels van het spel. Enkel indien deze functie `true` teruggeeft is een `Move` geldig en kan deze bevestigd worden.

2.5 Scheidsrechter

Het bestand `referee.oz` stelt de scheidsrechter voor. Dit bestand exporteert de functie `CreateReferee`. Deze aanvaardt de poorten van de twee spelers. Het doel van de referee is het regelen van het spel. De `CreateReferee` functie initialiseert een poort waarnaar berichten kunnen gestuurd worden. Die berichtenstroom wordt afgehandeld in een aparte draad met een accumulator loop, de berichten zijn telkens afkomstig van de spelers. De referee houdt de staat van het spel bij, om dit te realiseren wordt gebruik gemaakt van een state transition function. Deze functie retourneert telkens de nieuwe state, die op zich dan wordt gebruikt tijdens de verwerking van het volgende bericht. De referee kan de volgende berichten afhandelen:

makeBoard(N M) initialiseert een nieuw N*M bord en laat de speler een waarde kiezen voor K.

setK(K) initialiseert de waarde van K met de gegeven waarde en laat de witte speler een pion verwijderen indien K groter is dan 1. Anders mag hij onmiddelijk een move doen.

removePawn(Pawn) verwijdert de gegeven pion van het bord en laat de andere speler een pion verwijderen. Indien beide spelers K pionnen verwijderd hebben mag de witte speler een move doen.

checkMove(Move) controleert indien de gegeven `Move` geldig is. Als de `Move` niet geldig is, mag de speler een tweede poging doen (indien dit nog niet het geval was). In het andere geval wordt de `Move` bevestigd. Als de speler nu gewonnen is, eindigt het spel, in het andere geval mag de andere speler een zet doen.

Het is nu duidelijk dat de referee de flow van het spel beheert. Hij begint met het sturen van een `chooseSize()` bericht naar de speler. De speler reageert hierop met een `makeBoard(N M)` bericht. Zo verloopt het volledige spel door telkens te reageren op een bericht met een nieuw bericht. Het spel stopt indien een speler wint waardoor de scheidsrechter geen nieuwe berichten stuurt.

2.6 Spel

Het bestand `game.oz` maakt de spelers en de referee aan. Deze initialisatie is het enige nut van dit bestand.

3 Declaratieve model

Het declaratieve model is onafhankelijk (het resultaat is onafhankelijk van andere berekeningen) en deterministisch (gelijke inputs bieden telkens gelijke outputs). Daarnaast laat het declaratieve model geen neveneffecten toe. Deze drie eigenschappen minimaliseren het aantal bugs in het programma.
Het model is transparant wat ervoor zorgt dat men eenvoudig wiskundig kan redeneren over het model. Variabelen kunnen simpelweg worden gewijzigd naar hun waarde (door het determinisme) en de uitvoering zal hieronder gelijk blijven.
Als laatste heeft het declaratieve model ook als voordeel dat er kan gedefinieerd worden wat moet gebeuren i.p.v. hoe iets moet gebeuren. Dit verbetert de onderhoudbaarheid.
Het probleem met het declaratieve model is dat vele problemen niet kunnen opgelost worden met behulp van dit model. Bijvoorbeeld vele problemen vereisen het bijhouden van een staat wat niet mogelijk is met het declaratieve model. Dit kan opgelost worden door het toevoegen van extensies zoals het message-passing concurrency model.
Vooral het stateless zijn was een grote aanpassing voor mij tijdens het programmeren. Hierdoor moest mijn denkwijze vaak aangepast worden, bijvoorbeeld door het recursief oplossen van problemen. Bij het oplossen van sommige problemen zou een stateful model dus zeker enkele zaken veel makkelijker kunnen gemaakt hebben. Het objectgeoriënteerde model zou hier ook een goeie keuze zijn aangezien deze applicatie perfect kan gemodelleerd worden door het gebruik van klassen en objecten.

4 Message-passing concurrency model

Deze extensie van het declaratieve model zorgt ervoor dat er beperkte nondeterminisme mogelijk is. Echter in onze applicatie zal deze nondeterminisme niet voorkomen omdat nooit meerdere messages tegelijk kunnen toekomen in één draad.
Ook hier was het wat wennen om op deze manier om te gaan met communicatie tussen verschillende entiteiten. Deze manier van communicatie bleek echter zeer mooi te zijn door de goeie afscheiding en onafhankelijkheid van de verschillende

entiteiten. De entiteiten weten niet van elkaars bestaan, maar hebben enkel nood aan de poort om te kunnen communiceren met die entiteit. Deze manier van communicatie bleek dan ook zeer efficiënt voor de integratie van de code.

5 Toevoeging van een regel

De toevoeging van een regel bleek verrassend weinig werk. Hiervoor moesten 2 nieuwe messages kunnen afgehandeld worden in Player (chooseK, removePawn) en in Referee (chooseK, removePawn). Verder moest de staat van de state transition function 2 extra variabelen bijhouden. Dit vergde verder geen aanpassingen aan de overige code. Door het gebruiken van deze state transition function bleek dan ook de nood aan het bijhouden van een echte (globale) staat overbodig. De impact zou dus gelijk blijven bij het gebruik van dergelijke staat.

6 Integratie van de code

De interface (afhandeling van berichten) vastgesteld als groep is zeer gelijkaardig aan de interface die ik reeds aanbood. Dit zorgde ervoor dat weinig nog moest veranderen om aan deze interface te voldoen. De volgende wijzigingen moesten gebeuren:

- Hernoemen van x en y in `Coord` naar resp. row en col.
- Hernoemen van start en dest in `Move` naar resp. startcoord en stopcoord.
- Hernoemen van boodschap `setSize(N M)` naar `makeBoard(N M)`.
- Het meegeven van aantal foutieve pogingen `Strikes` van Referee naar Speler.

Alsook, door het gebruik van de message-passing concurrency model was integratie zeer makkelijk. De referee hoeft de opponent niet rechtstreeks aan te spreken, maar zendt enkel berichten naar zijn port.