Inleiding

De grote interesse in het domein ‘onzekerheid’ in AI resulteert in de ontwikkeling van verschillende programmeertalen. Deze talen worden probabilistische programmeertalen of PPL (Probabilistic Programming Language) genoemd. Een PPL heeft in grote lijnen 2 hoofdfuncties:

1. Het modelleren van een wereld met onzekerheid
2. Het redeneren/infereren van vragen over deze wereld

Voorheen was het zeer moeilijk als programmeur om werelden met onzekerheid te modelleren, laat staan het redeneren over deze werelden. Met de komst van PPL’s is dit probleem al een stuk makkelijker geworden.

Elke PPL heeft zijn eigen manier van implementatie en wordt vaak geïmplementeerd als extensie op een general-purpose programmeertaal. Dit zorgt ervoor dat de PPL niet gelimiteerd is aan een kleine subset van de werelden die het kan modelleren. Veel van deze PPL’s streven naar een balans tussen performantie en expressiviteit. Het is belangrijk voor een PPL om genoeg werelden te kunnen simuleren en op een aanneembare tijd te kunnen redeneren over vragen over deze wereld.

Omdat er zo veel PPL’s beschikbaar zijn de laatste jaren is het niet altijd duidelijk welke voordelen of nadelen ze hebben ten opzichte van andere PPL’s. Verschillende van deze PPL’s zijn in recente artikels vergeleken met elkaar aan de hand van eigenschappen en concepten van de taal (Probabilistic (Logic) Programming Concepts, Luc De Raedt - Angelika Kimmig). Andere artikels vergelijken PPL’s aan de hand van hun vorige iteratie of PPL’s met hetzelfde programmeerparadigma (Inference and Learning in Probabilistic Logic Programs using Weighted Boolean Formulas, LUC DE RAEDT et al). Deze artikels evalueren PPL’s met hetzelfde programmeerparadigma zoals logische probabilistische programmeertalen of functionele probabilistische programmeertalen.

In deze thesis ben ik van plan PPL’s met verschillende programmeerparadigma zoals ProbLog en Anglican te evalueren. ik ga deze PPL’s evalueren ten opzichte van elkaar aan de hand van qualitatieve en quantitatieve criteria zoals: performantie, expressiviteit, geheugengebruik, uitbreidbaarheid, tools beschikbaar, moeilijkheidsgraag,... Omdat het niet triviaal is om programmeertalen te vergelijken die totaal anders geïmplementeerd zijn maak ik gebruik van een case study. Ik begin met het verzinnen van een onzekerheidsprobleem waarna ik uitleg geef waarom ik gekozen heb voor dit probleem. Daarna evalueer ik de implementatie van het probleem in de verschillende PPL’s aan de hand van vooropgestelde criteria. Ten slotte volgt een evaluatie van de PPL’s ten opzichte van elkaar. Uiteindelijk wil ik kunnen aantonen welke PPL het best presteert in welke criteria aan de hand van het opgegeven probleem.

Achtergrond

In deze sectie geef ik de nodige achtergrondinformatie om de rest van mijn thesis te begrijpen.

Onzekerheid in artificiele intelligentie is één van de invloedrijkste domeinen van artificiele intelligentie. De reden hiervoor is omdat de wereld van nature veel onzekerheid bevat. Denk aan de volgende punten:

- Kennis (we kunnen niet alles van de wereld weten)
- Incomplete modellen (verschijnselen die niet onder het model vallen)
- Sensoren (we kunnen de wereld enkel observeren met de tools die beschikbaar zijn en deze zijn meestal nog foutgevoelig.)
- Acties (we kunnen niet elke actie uitvoeren)

Als we spreken over werken met onzekerheid bedoelen we het opstellen van een hypothese, en deze hypothese (zo goed mogelijk) bewijzen aan de hand van het gegeven bewijs van de wereld.

Er zijn 3 belangrijke kwesties in verband met het werken met onzekerheid:

- Representeren van onzekerheid
- Redeneren over onzekerheid
- Leren aan de hand van onzekerheid

Het representeren van onzekerheid gebeurt in het model. Een model is een weergave van een onzekerheidsprobleem waarbij elke wereld kan gesimuleerd worden.

Bij het redeneren over onzekerheid maken we gebruik van het model om vragen te stellen over mogelijke hypotheses. (vb. wat is de kans dat we een eerlijk muntstuk hebben als we deze 20 keer tossen en 15 keer hoofd en 5 keer munt verkrijgen). In dit voorbeeld is de hypothese “of het muntstuk eerlijk is”. Het bewijs dat we hebben is dat we 20 keer tossen en 15 keer hoofd en 5 keer munt kregen. Wat we willen is de kans dat de hypothese klopt, m.a.w. de kans dat het muntstuk eerlijk is.

Een manier om te berekenen wat de kans is of een munt eerlijk is, is het gebruik maken van de Bayes’s rule:

$$P(Hypothese|Bewijs) = \frac{P(Bewijs|Hypothese)P(Hypothese)}{P(Bewijs)}$$

Bayes' rule geeft de kans dat een hypothese waar is in een wereld waar er al dan niet bewijs is over deze wereld. Voor meer informatie verwijs ik naar het boek (Bayesian Reasoning and Machine Learning). PPL's kunnen hetzelfde berekenen aan de hand van een inferentie proces dat de taal implementeert. Elke PPL heeft een methode om de inferentie te bereken. Hoe ze dit doen verschilt voor elke PPL.

Het berekenen van de inferentie is een zeer krachtig, maar een zeer kostelijk proces qua rekenkracht. Veel PPL's zoeken een balans tussen hoe efficient ze inferentie kunnen berekenen en welke problemen ze kunnen modelleren (hoe expressief de taal is).

Omdat het modelleren van een onzekerheidsprobleem, het redeneren over dit probleem en het leren aan de hand van de redeneringen een intensief proces is, is het beter om hiervoor computerkracht te gebruiken. Hierdoor werden er Probabilistische programmeertalen ontworpen.

Probabilistische programmeertalen

Probabilistische programmeertalen (of PPL's van Probabilistic Programming Languages) zijn programmeertalen met 2 hoofdfuncties:

- Het modelleren van een wereld met onzekerheid
- Het redeneren/infereren van vragen over deze wereld

Er zijn PPL implementaties die een volledig nieuwe taal hebben geïmplementeerd speciaal voor het modelleren en redeneren, maar de meesten zijn geïmplementeerd als extensie op een bestaande general-purpose programmeertaal. In deze thesis maak ik gebruik van 2 PPL's, namelijk: ProbLog2 en Anglican.

ProbLog2

ProbLog2 is een PPL die gebaseerd is op de Sato's distribution semantics (Sato 1995). Het kan beschouwd worden als een Prolog programma met probabilistische aspecten geïmplementeerd. Een ProbLog programma bestaat uit feiten geannotteerd met kansen.

0.5 :: heads(C) :- coin(C).
coin(c1).

In het bovenstaande voorbeeld hebben we 1 feit: *coin(c1).* en 1 probabilistisch predicaat: *0.5 :: heads(C) :- coin(C).* wat dit programma zegt is dat het predicaat *heads(C)* waar is in de wereld met 50% kans voor elke munt die geïnstantieerd is als *coin(C).*

We kunnen ook gebruik maken van "annotated disjunction" wat eigenlijk syntactische suiker is om hetzelfde te bereiken. Als we het vorige voorbeeld schrijven met "annotated disjunction" komen we tot:

0.5 :: heads(C, true); 0.5:: heads(C, false) :- coin(C).
coin(c1).

In dit voorbeeld geeft het heads predicaat voor een bepaalde munt true voor 50% van de tijd en false voor 50% van de tijd. De totale kans voor het predicaat heads is 100%. Stel dat we *heads(C,false).* een kans geven van 0.4, dan hebben we 50% kans op true als parameter voor heads, 40% kans op false als parameter voor heads en 10% kans dat het predicaat zelf false terug geeft. Voor dit klein programma lijkt dit onnodig maar voor grotere probabilistische problemen kan "annotated disjunction" zeer handig zijn om een beter inzicht te krijgen over wat de probabilistische predicaten doen.

De combinatie van deze regels en het logic programmeer systeem prolog geeft ons de mogelijkheid om zeer uitgebreide probabilistische werelden te modelleren. We kunnen vragen stellen aan deze modellen en aan de hand van de inferentie machine van ProbLog worden deze vragen opgelost.

Inferentie

Om vragen te stellen over het model maken we gebruik van queries en bewijzen. Als we willen berekenen hoeveel kans we hebben dat we 2 munten tossen en ze beiden op hoofd landen kunnen we dit doen aan de hand van het volgende programma:

```
0.5 :: heads1.
0.5 :: heads2.
two_heads :- heads1, heads2.

query(two_heads).
```

Dit geeft het resultaat 0.25 wat uiteindelijk de kans is van $P(heads1) * P(heads2)$.

We kunnen het resultaat van de query manipuleren door bewijs te geven aan dit model. Stel dat we weten dat de eerste munt zeker hoofd als resultaat heeft, dan kunnen we dit als bewijs meegeven aan het model op de volgende manier:

```
0.5 :: heads1.
0.5 :: heads2.
two_heads :- heads1, heads2.

Evidence(heads1, true).
query(two_heads).
```

Dit geeft het resultaat 0.5. Omdat we weten dat de eerste munt zeker in hoofd resulteert is de uitkomst gewoon de kans dat de tweede munt op hoofd resulteert. Dit is voor de munt in het model 0.5. Voor meer voorbeelden verwijs ik naar de tutorials van de ProbLog website: https://dtai.cs.kuleuven.be/problog/tutorial.html

De inferentiemachine in ProbLog is gebasseerd op “Knowledge Compilation”. Dit houdt verschillende stappen in:

1. Het “gronden” van het programma. Dit wil zeggen alle variabelen in het programma vervangen door de termen die deze variabelen kunnen bevatten. Dit houdt enkel rekening met de queries dus predicaten in het systeem die niet aangesproken worden, worden ook niet gegrond.
2. Het gegronde programma converteren naar een equivallente booleaanse formula.
3. Het bewijs en gewogen functies gebruiken om de booleaanse formula te converteren naar een gewogen booleaanse formula.

Om de succes probabiliteit (SUCC), de conditionele probabiliteit (MARG) en de meest waarschijnlijke probabiliteit (MPE) te verkrijgen gebruiken we de gewogen booleaanse formula in combinatie met verschillende algoritmes zoals bijvoorbeeld “Weighted Model Counting (WMC)” om deze te berekenen. Meer informatie over het ProbLog Systeem en de inferentiemachine kunt u terugvinden in het artikel (Inference and Learning in Probabilistic Logic Programs using Weighted Boolean Formulas).

De API van het ProbLog systeem kan ook gebruikt worden via Python. Hierdoor kunnen we gebruik maken van Python om het inferentie process te manipuleren. Dit zorgt voor extra uitbreidbaarheid van verschillende problemen en dus extra expressiviteit van het systeem in totaal. De API is beschikbaar via de volgende URL: https://problog.readthedocs.io/en/latest/api.html

Uitwerking

Probleem verzinnen

Als onzekerheidsprobleem heb ik gekozen om een spel te modelleren dat onderheven is aan probabilistische aspecten. Om het spel te spelen heb ik 4 spelstrategiën ontwikkeld die elks ook onderheven zijn aan probabilistische aspecten.

Spel

Het spel bestaat uit een bord van 10 op 10 blokken. Wanneer het spel gestart wordt krijgen de blokken een random kleur toegewezen maar er kunnen geen 3 van dezelfde blokken op een rij staan (enkel verticaal en horizontaal). Er zijn 4 kleuren in totaal: rood, groen, geel, blauw. In figuur 1 ziet u een voorbeeld van een begin bord.

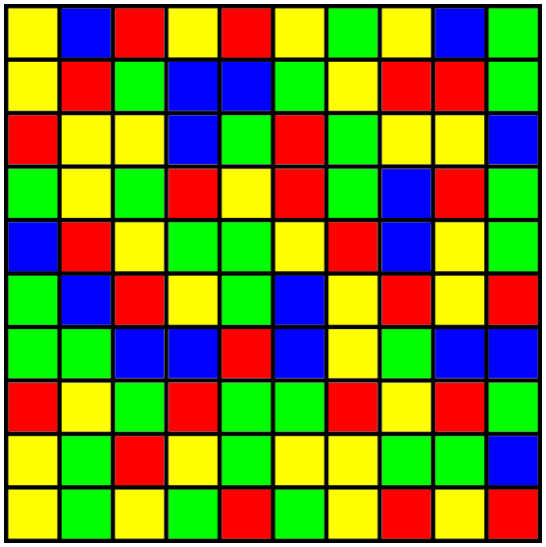


Figure 1 voorbeeld van een intieel bord

De speler kan op elk van de blokken op het bord drukken. Als de speler op een blok drukt verandert deze van kleur. De kleur waar de blok in veranderd hangt af van de probabilistische distributie. Om het simpel te houden gebruik ik hier een uniforme distributie:

Verandert in / Block kleur	Rood	Groen	Blauw	Geel
Rood	0	1/3	1/3	1/3
Groen	1/3	0	1/3	1/3
Blauw	1/3	1/3	0	1/3
Geel	1/3	1/3	1/3	0

Figure 2 probabilistische distributie voor het veranderen van kleuren.

In woorden betekent dit dat als er op een rode blok wordt gedrukt er 1/3 kans is dat deze blok in een groene verandert, 1/3 kans in een blauwe verandert en 1/3 kans in een gele verandert. Voor een groene, blauwe en gele blok is dit analoog.

Als er drie of meer blokken van dezelfde kleur ofwel horizontaal naast elkaar liggen ofwel verticaal naast elkaar liggen verdwijnen ze en dit levert punten op. De blokken die zich boven de verdwenen blokken bevinden vallen naar beneden tot ze op een andere blok belanden ofwel op de bodem van het spelbord belanden. Voor elke blok die verwijderd wordt krijgt de speler 1 punt. De bedoeling van het spel is om in 5 beurten zoveel mogelijk punten te behalen waarin de speler in elke beurt 1 blok van kleur mag veranderen. De beurt eindigt wanneer er geen 3 blokken van dezelfde kleur meer op een rij staan. In figuur 3 ziet u het verloop van een beurt in een 10x10 bord.

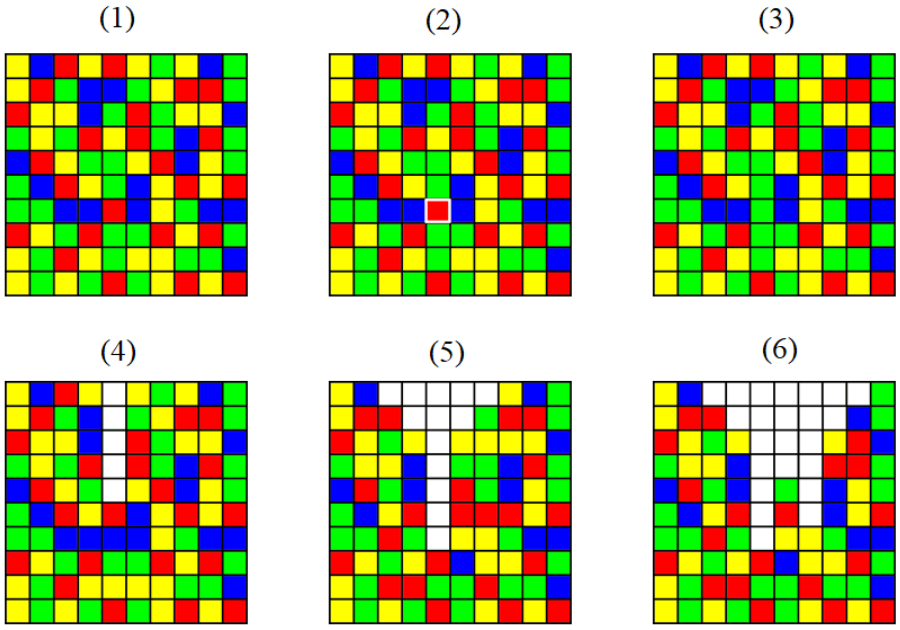


Figure 3 er wordt een blok gekozen om op te drukken, in dit geval een rode blok. De blok verandert met een 1/3 kans in een groene blok. Omdat er meer als 2 blokken van dezeflde kleur op een rij staan worden deze verwijderd en de bovenstaande blokken vallen naar beneden. Dit wordt herhaald tot er niet meer als 2 blokken van dezelfde kleur op een rij staan

Strategiën

Ik heb 4 strategiën ontwikkeld om het spel te spelen.

- Uniforme strategie
- Kleuren ratio strategie
- Mogelijke score strategie
- Gewogen score strategie

Deze strategiën kunnen gebruikt worden om het spel te spelen op een bepaalde wijze. Elke strategie kiest altijd 1 blok uit de mogelijke blokken die beschikbaar zijn. Welk blok dit is hangt van de strategie af.

Uniforme strategie

Als de uniforme strategie wordt toegepast is de kans dat een blok gekozen wordt uniform voor elk blok in het spelbord. Voor een 10x10 bord is de kans dat een blok wordt gekozen $\frac{1}{10*10} = \frac{1}{100}$. In figuur 4 zien we alle mogelijke keuzes die de uniforme strategie kan kiezen in het gegeven bord.

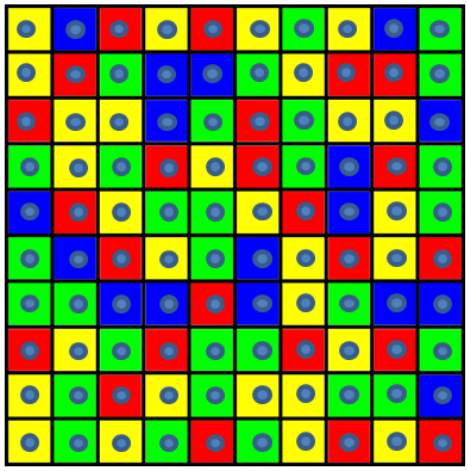


Figure 4 in de uniforme strategie kunnen alle blokken gekozen worden met een uniforme kans verdeling. De kans is 1/100 voor elke blok in dit geval

Kleuren ratio strategie

Voor de kleuren ratio strategie worden eerst alle blokken met dezelfde kleur opgeteld. Uit de kleur met het minst aantal blokken wordt uniform een blok gekozen. In figuur 5 zien we dat de blauwe blokken in de minderheid zijn. De strategie zorgt ervoor dat er uniform een blauwe blok wordt gekozen.

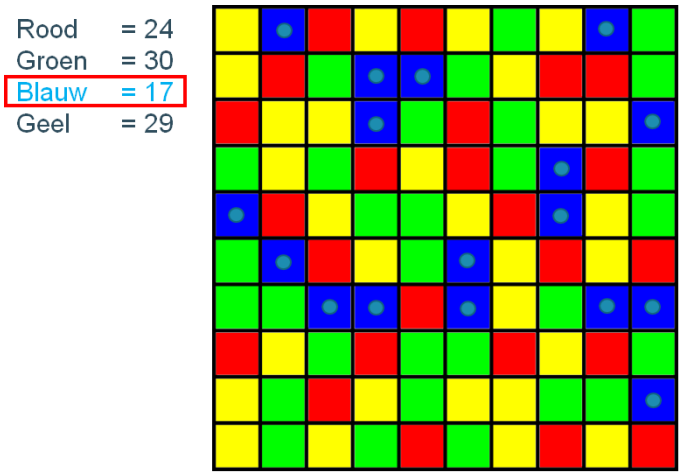


Figure 5 In bovenstaande figuur is de kleuren ratio voor de blauwe blokken het minst. Hier wordt uniform een blauwe blok gekozen met een 1/17 kans voor elke blok

Mogelijke score strategie

In de mogelijke score strategie wordt voor elke blok apart nagegaan of deze een mogelijke score kan hebben. Een blok kan een mogelijke score hebben als deze blok kan veranderen in een kleur die een score oplevert. In figuur 6 ziet u alle blokken aangeduid die een mogelijke score kunnen opleveren. Er wordt 1 blok uit deze blokken gekozen met een uniforme kansverdeling.

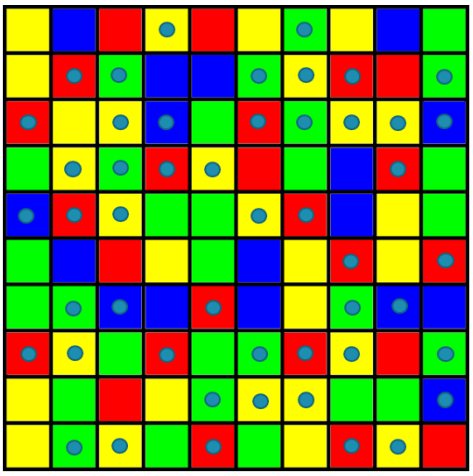


Figure 6 In een mogelijke score strategie wordt er uniform een blok gekozen uit alle blokken met een mogelijke score. In dit geval zijn er 48 blokken met een mogelijke score dus de kansverdeling is 1/48 voor elke blok

Gewogen score strategie

De gewogen score strategie is een uitbreiding op de mogelijke score strategie waar we niet enkel naar de mogelijke score zien, maar naar de gewogen score. Elke blok kan veranderen van kleur aan de hand van een kansverdeling. De gewogen score strategie houdt rekening met deze kansverdeling. De gewogen score wordt berekend aan de hand van de score als een blok in Rood/Groen/Blauw/Geel verandert gewogen met de kans dat de blok verandert in Rood/Groen/Blauw/Geel.

Implementatie

De implementatie van het spel gebeurt in 2 verschillende PPL's, namelijk ProbLog2 en Anglican. Tijdens het begin van de implementatie heb ik goed nagedacht welke aspecten ik allemaal zou moeten implementeren. Hieronder vindt u een chronologische lijst met alle aspecten die ik wou implementeren in het model.

1. Kleuren van blok veranderen in ProbLog
2. Spel predicaten om het spel te spelen
3. Uniform kiezen van een blok om op te drukken (uniforme strategie)
4. Mogelijke bordconfiguraties kunnen samplen
5. Implementatie van kleuren ratio strategie en mogelijke score strategie voor testen
6. ProbLog API gebruiken via python
7. ProbLog "knowledge compilation" process manipuleren aan de hand van python
8. Sampling implementeren met behulp van python
9. Compile-once Evaluate-many process hanteren op het ProbLog process

10. ProbLog als meta-taal gebruiken in python (implementatie van gewogen score strategie)

(1) Kleuren van blok veranderen in ProbLog

Om een kleur van een blok te veranderen in ProbLog heb ik gebruik gemaakt van “annotated disjuntion”.

1/3::color_change(red,green); 1/3::color_change(red,blue); 1/3::color_change(red,yellow).
1/3::color_change(green,red); 1/3::color_change(green,blue); 1/3::color_change(green,yellow);
1/3::color_change(blue, red); 1/3::color_change(blue, green); 1/3::color_change(blue,yellow);
1/3::color_change(yellow, red); 1/3::color_change(yellow, green); 1/3::color_change(yellow,blue);

Deze code geeft een distributie over de kans dat een gegeven kleur kan veranderen in een andere kleur zoals gegeven in de volgende tabel.

Verandert in / Block kleur	Rood	Groen	Blauw	Geel
Rood	0	1/3	1/3	1/3
Groen	1/3	0	1/3	1/3
Blauw	1/3	1/3	0	1/3
Geel	1/3	1/3	1/3	0

Deze code lijkt goed te werken maar het bevat 1 grote fout als we rekening houden met de werking van het spel. ProbLog maakt namelijk gebruik van stochastische memoization. Dit wilt zeggen dat ProbLog onthoudt welke uitkomst waar is in een gegeven wereld. Stel we hebben 2 blokken met een rode kleur. Hier aangeduid met een lijst *[block(red),block(red)]*. We drukken eerst op de eerste blok en daarna op de tweede blok. Als er op de eerste blok wordt gedrukt worden er 3 werelden gecreëerd met hun respectievelijke kans, namelijk:

- *[block(green), block(red)] : 1/3*
- *[block(blue), block(red)] : 1/3*
- *[block(yellow), block(red)] : 1/3*

Als we nu drukken op de tweede blok zou u denken dat voor elk van de drie werelden nog eens drie werelden worden aangemaakt met elks een kans van 1/9 (de kans dat de wereld bestaat maal de kans van de kleur waar de tweede blok in kan veranderen). De uitkomst van de tweede beurt is:

- *[block(green), block(red)] : 1/3*
  - *[block(green), block(green)] : 1/3*
  - *[block(green), block(blue)] : 0*
  - *[block(green), block(yellow)] : 0*
- *[block(blue), block(red)] : 1/3*
  - *[block(blue), block(green)] : 0*
  - *[block(blue), block(blue)] : 1/3*
  - *[block(blue), block(yellow)] : 0*

- *[block(yellow), block(red)] : 1/3*
    - *[block(yellow), block(green)] : 0*
    - *[block(yellow), block(blue)] : 0*
    - *[block(yellow), block(yellow)] : 1/3*

Wat gebeurt hier nu? Dit is de stochastische memoization dat aan het werk is. Om uit te leggen wat hier gebeurt gebruik ik de wereld waar er gedrukt is op de eerste blok en deze is in een groene blok veranderd *[block(green), block(red)]*.

Als de eerste rode blok in een groene blok veranderd is, onthoudt de inferentie machine dat het predicaat *color_change(red,green)* waar is in deze wereld. Bij het gebruik van "annotated disjunction" kan er altijd maar 1 predicaat juist zijn. In dit geval is het predicaat *color_change(red,green) waar en* de predicaten *color_change(red, blue), color_change(red,yellow)* zijn niet waar in de wereld. Als we nu op de tweede rode blok drukken spreken we terug het predicaat color_change(red,Color) aan, maar omdat de inferentie machine weet dat enkel color_change(red,green) waar is met een kans van 100% wordt er enkel een wereld gecreëerd waar de tweede blok ook groen is. De rest van de werelden hebben 0% kans om te bestaan omdat deze niet in rij staan met de regels van het model.

Dit is natuurlijk een groot probleem voor het model van het spel. Omdat het spel gebruik maakt van beurten willen we er natuurlijk zeker van zijn dat het inferentie alle juiste werelden kan creëeren. Als we het spel voor 1 beurt zouden spelen zou dit geen probleem zijn, maar vanaf we het spel twee beurten laten spelen kan de inferentie machine een foute uitkomst geven.

Om dit op te lossen maak ik gebruik van een soort van ID voor het predicaat. Als ID gebruik ik "welke beurt wordt momenteel gespeeld". De code verandert dan in:

1/3::color_change(red,green,T); 1/3::color_change(red,blue,T); 1/3::color_change(red,yellow,T).
1/3::color_change(green,red,T); 1/3::color_change(green,blue,T); 1/3::color_change(green,yellow,T);
1/3::color_change(blue, red,T); 1/3::color_change(blue, green,T); 1/3::color_change(blue,yellow,T);
1/3::color_change(yellow, red,T); 1/3: color_change(yellow, green,T); 1/3::color_change(yellow,blue,T);

T staat hier voor de beurt dat momenteel gespeeld wordt. Als we dit gebruiken met het voorbeeld van de twee rode blokken gaat de eerste rode blok zien naar het predicaat *color_change(1,red,Color1)* en de tweede rode blok naar het predicaat *color_change(2,red,Color2).* De ID zorgt er dus voor dat stochastisch memoization niet gebruikt wordt.

(2) spel predicaten om het spel te spelen

Over het implementeren van de spelregels ga ik niet veel zeggen. Vooral omdat dit gewoon prolog code is die uitgevoerd wordt. Het belangrijkste predicaat dat ik wil laten zien is het board\4 en het score_of_turn\2 predicaat

```
board(0,Board,0,[]) :-
        initial_board(Board).
board(T,Board,Score,Positions) :-
        T > 0,
```

TT is T - 1,
board(TT,PreviousBoard,PreviousScore,PreviousPositions),
press(PreviousBoard,X,Y,Color,TT,true),
append(PreviousPositions,[[X,Y]],Positions),
change_color(Color,NewColor,TT),
change_color_in_board(PreviousBoard,X,Y,NewColor,ColorChangedBoard),
remove_and_drop(ColorChangedBoard, Board, CurrentScore),
Score is PreviousScore + CurrentScore.

score_of_turn(T,Score) :-
board(T,Board,Score,Positions).

Het board\4 predicaat ligt aan het hart van het model. Dit is het predicaat dat de strategie toepast op elke beurt en de kleuren verandert. Zoals u kunt zien in de bovenstaande code worden er elke beurt sequentiële stappen gevolgd:

1. Als we de score, posities en bord willen vinden van beurt 0 krijgen we het initieel bord. Elke beurt later dan beurt 0 gaat recursief terug naar het beginbord om vanaf daar te spelen. (*board(TT,PreviousBoard,PreviousScore,PreviousPositions)*)
2. Het press\6 predicaat past een gegeven strategie toe en geeft de X en Y coördinaten terug van de blok die gekozen is door de strategie. (***press(PreviousBoard,X,Y,Color,TT,true)***)
3. De gevonden coördinaten worden als een positie in een chronologische manier opgeslagen in een lijst zodat we uiteindelijk kunnen zien welke blokken er gekozen zijn. (*append(PreviousPositions,[[X,Y]],Positions)*)
4. We zoeken de nieuwe kleur waar de gevonden blok in kan veranderen. (***change_color(Color,NewColor,TT)***)
5. We veranderen deze kleur in het bord van de vorige beurt. (*change_color_in_board(PreviousBoard,X,Y,NewColor,ColorChangedBoard)*)
6. We checken of er meer als 2 blokken met dezelfde kleur op een rij staan en als dit zo is verwijderen we de blokken en laten we de bovenstaande blokken vallen. Dit herhalen we tot er niet meer als 2 blokken van dezelfde kleur op een rij staan. Dit levert ook de uiteindelijke score op van een beurt, (*remove_and_drop(ColorChangedBoard, Board, CurrentScore)*)
7. We tellen de score op van het vorige bord met de score van deze beurt. (*Score is PreviousScore + CurrentScore.*)

Als we vragen aan de inferentie machine om dit predicaat op te lossen krijgen we voor elk van de werelden die kunnen gecreëerd worden hun respectievelijke kans dat ze bestaan. Dit is leuk om te testen maar eigenlijk willen we alleen maar weten welke strategie het beste presteert op een gegeven bord. Dus variabelen zoals Board en positions zijn niet nodig om deze te berekenen. Het goede aan ProbLog is dat als we een query opstellen voor een bepaald aantal variabelen, het enkel rekening houdt met deze variabelen. De andere variabelen die toevallig aanwezig zijn in een ander predicaat worden verwaarloosd. Om dit te benutten maak ik gebruik van het predicaat score_of_turn\2. Dit predicaat roept gewoon board\4 op maar houdt geen rekening met hoe het bord er uitziet in elke wereld en welke posities hiervoor gebruikt zijn. Het wilt enkel de mogelijke scores weten op een gegeven beurt.

Dus stel dat we een query opstellen voor score_of_turn\2 met als beurt 1.

query(score_of_turn(1,X)).

Wat we nu vragen is wat is de kans dat we score X behalen op 1 beurt. De uitkomst van deze query kan er uitzien zoals dit:

score_of_turn(1,0) : 0.88888888889
score_of_turn(1,3) : 0.11111111111

Dit wilt zeggen dat als we een gegeven strategie toepassen op een gegeven bord en dit spelen voor 1 beurt hebben we 89% kans op een score van 0 en 11% kans op een score van 3.

(3) Uniform kiezen van een blok om op te drukken (uniforme strategie)

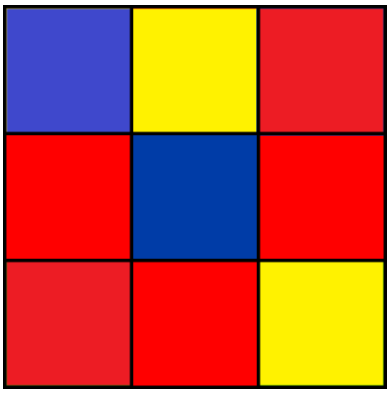
Tijdens de implementatie van de uniforme strategie was het de eerste keer dat ik met een bug te maken had die rare resultaten gaf. De eerste keer dat ik deze strategie implementeerde zag de code er als volgt uit:

```
P::uniform_block(Board,X,Y,Color,T,uniform) :-
        find_block_in_board(block(Color,X,Y),Board),
        pressable_color(Color),
        how_many_blocks_with_color(Board, N),
        P is 1 / N.
```

Stel we hebben een 3x3 bord met geen verwijderde blokken. Dit geeft een predicaat zodat uniform_block waar is met een kans van 1/9 = 11%. Voor 89% is het predicaat niet waar in deze wereld. Dit is natuurlijk een probleem omdat we willen dat de uniforme strategie altijd een blok terug geeft, niet maar 11% van de tijd. Zoals men zegt een rookie-mistake. De reden dat ik dit aangeef is omdat de uitkomst van mijn queries een oplossing gaven die correct was als ik de volgende query uitvoerde:

```
board(0,Board,0) :-
        initial_board(Board).
board(T,Board,Score) :-
        T > 0,
        TT is T - 1,
        board(TT,PreviousBoard,PreviousScore),
        uniform_block (PreviousBoard,X,Y,Color,TT),
        change_color(Color,NewColor,TT),
        change_color_in_board(PreviousBoard,X,Y,NewColor,ColorChangedBoard),
        remove_and_drop(ColorChangedBoard, Board, CurrentScore),
        Score is PreviousScore + CurrentScore.
query(board(1,_,_)).
```

Dit gaf mij alle mogelijke borden met de correcte score en de correcte kans dat deze wereld bestaat in het model. Een voorbeeld van een resultaat dat deze query gaf voor een 3x3 bord is:



- board(1,[[block(blue,0,2), block(yellow,1,2), block(red,2,2)], [block(red,0,1), block(blue,1,1), block(red,2,1)], [block(red,0,0), block(red,1,0), block(blue,2,0)]],0): 0.11111111
- board(1,[[block(blue,0,2), block(yellow,1,2), block(red,2,2)], [block(red,0,1), block(blue,1,1), block(red,2,1)], [block(red,0,0), block(red,1,0), block(green,2,0)]],0): 0.11111111
- board(1,[[block(blue,0,2), block(yellow,1,2), block(red,2,2)], [block(red,0,1), block(green,1,1), block(red,2,1)], [block(red,0,0), block(red,1,0), block(yellow,2,0)]],0): 0.11111111
- board(1,[[block(blue,0,2), block(yellow,1,2), block(red,2,2)], [block(red,0,1), block(yellow,1,1), block(red,2,1)], [block(red,0,0), block(red,1,0), block(yellow,2,0)]],0): 0.11111111
- board(1,[[block(green,0,2), block(yellow,1,2), block(red,2,2)], [block(red,0,1), block(blue,1,1), block(red,2,1)], [block(red,0,0), block(red,1,0), block(yellow,2,0)]],0): 0.11111111
- board(1,[[block(white,0,0), block(white,1,0), block(white,2,0)], [block(blue,0,1), block(yellow,1,1), block(red,2,1)], [block(red,0,2), block(red,1,2), block(yellow,2,2)]],3): 0.11111111
- board(1,[[block(white,0,0), block(white,1,0), block(white,2,0)], [block(blue,0,1), block(yellow,1,1), block(white,2,1)], [block(red,0,2), block(blue,1,2), block(white,2,2)]],5): 0.11111111
- board(1,[[block(white,0,0), block(yellow,1,0), block(red,2,0)], [block(white,0,1), block(blue,1,1), block(red,2,1)], [block(white,0,2), block(red,1,2), block(yellow,2,2)]],3): 0.11111111
- board(1,[[block(yellow,0,2), block(yellow,1,2), block(red,2,2)], [block(red,0,1), block(blue,1,1), block(red,2,1)], [block(red,0,0), block(red,1,0), block(yellow,2,0)]],0): 0.11111111

Dus op het eerste zicht had ik zelfs niet door dat er iets mis was. Het was pas als ik gebruik maakte van de volgende query dat ik door had dat er iets mis was:

```
score_of_turn(T,Score) :-
        board(T,Board,Score).
query(score_of_turn(1,_)).
```

Deze query gaf als resultaat:

- score_of_turn(1,0): 0.51028807
- score_of_turn(1,3): 0.18518519
- score_of_turn(1,5): 0.11111111

Wat er niet klopt aan dit resultaat, is dat de kansen dat deze werelden bestaan nu niet meer optellen tot 1. Dit wilt dus zeggen dat er sommige werelden zijn die niet kunnen bestaan omdat er een predicaat niet waar is in de query.

Ik geef dit als voorbeeld omdat het de eerste keer was dat ik vast zat met een BUG. Omdat de query over het *board\3* predicaat de juiste oplossing gaf en de query over het *score_of_turn\2* predicaat niet klopte, wist ik niet meteen waar het probleem lag. Dit was de eerste keer dat ik dacht, als ik nu een debugger of iets dergelijks ter beschikking had, kon ik dit waarschijnlijk meteen oplossen. Spijtig genoeg is een debugger maken voor een inferentie machine gebasseerd op eerste orde logica moeilijk of al dan niet onmogelijk om te maken.

(4) Mogelijke bordconfiguraties kunnen samplen

(5) Implementatie van kleuren ratio strategie en mogelijke score strategie voor testen

(6) ProbLog API gebruiken via python

(7) ProbLog “knowledge compilation” process manipuleren aan de hand van python

(8) Sampling implementeren met behulp van python

(9) Compile-once Evaluate-many process hanteren op het ProbLog process

(10) ProbLog als meta-taal gebruiken in python (implementatie van gewogen score strategie)

Gerelateerd werk

- Inference and Learning in Probabilistic Logic Programs using Weighted Boolean Formulas
- Probabilistic Logic Programming Concepts
- Exploiting Local and Repeated Structure in Dynamic Bayesian Networks
- Bayesian Reasoning and Machine Learning
- https://dtai.cs.kuleuven.be/problog/tutorial.html

Conclusie

Planning

1. Verdere evaluatie van ProbLog2
2. Implementatie van model in Anglican
3. Evaluatie van Anglican
4. Evaluatie van ProbLog2 en Anglican ten opzichte van elkaar
5. (optioneel) Stap 2 tot 4 herhalen voor een andere PPL.

In de tussentijd wordt er gewerkt aan mijn tekst.