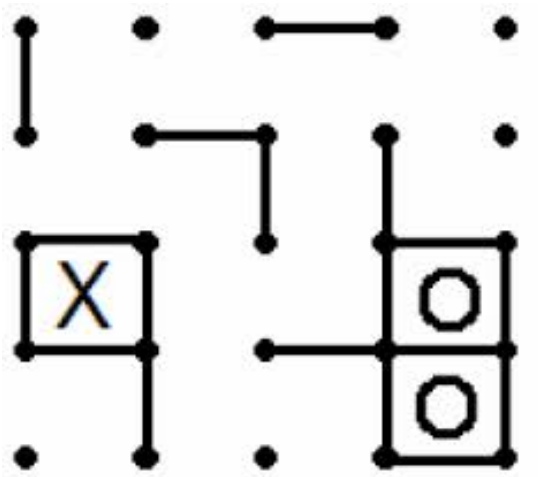
Universiteit van Amsterdam

Programmeertalen

Erlang

Erik Kooistra - Jordy Perlee - Stephan van Schaik - Robin de Vries - Seyla Wachlin - Damian Frölich

Kamertje Verhuur



Deadline: Individueel: 4 maart 2020 15:59
Team: Zondag 8 maart 17:59

Introductie

Deze week, die in het teken staat van Erlang, krijg je als eerste een individuele opgave om het spel boter kaas en eieren te implementeren, vervolgens volgt een team opgave waar je een spel gaat maken dat gespeeld wordt door twee spelers. Hierbij worden er voor de spelers computer processen gebruikt om de code parallel, m.a.w. naast elkaar, uit te kunnen voeren. Aangezien deze opdracht een vrij aardige uitdaging is, staan de logische stappen om de opdracht tot een goed einde te brengen in dit document beschreven. Voor de tests wordt `eunit` gebruikt, welke automatisch geïnstalleerd wordt bij het installeren van Erlang. We gebruiken de versies van Ubuntu 18.04.

Boter, Kaas en eiren

Voor deze opdracht is een skeleton file aangeleverd die op canvas te vinden is in het `tar.gz` bestand in de map individueel en de skeleton file heet `tictactoe.erl`. Dit maakt gebruik van het concept `gen_server` in erlang. Voor meer informatie zie `http://erlang.org/doc/man/gen_server.html` Het `gen_server` model implementeert een basis server in het client server model. Daarnaast biedt `gen_server` automatisch mogelijkheden om fault tolarant code te schrijven, om bijvoorbeeld automatische failovers en restarts mogelijk te maken als de server, waar jouw programma op draait, crashed. De basis van jouw programma zal zitten in de funtie `handle_call/3` deze wordt elke keer aangeroepen als jouw proces een bericht krijgt.

0.1 generic server

Omdat erlang van nature een gedistribueerde taal is moet je op een manier je state bijhouden omdat je geen global variablen hebt die je continue moet updaten. Binnen de `gen_server` is dat als volgende geimplementeerd. Elke aanroep aan `handle_call/3` krijgt 3 waardes mee, het bericht, wie het gestuurd heeft en wat de huidige state is van de server. Wanneer het bericht afgehandeld hebt en iets wilt terugsturen moet je een tuple teruggeven met de volgende waardes `{reply,Mesg,State}`, waar `Mesg` de waarde is die de caller terug krijgt en `State` de nieuwe interne staat van de server.

0.2 De implementatie

Het doel is om het spel boter kaas en eieren te spelen zodat je tegen een computer het spel kan spelen, en het bord kan weergeven. De bot tegen wie je speelt hoeft niet intelligent te zijn en mag ook telkens een random positie kiezen. Hiervoor moeten jullie de volgende calls en functies implementeren.

De volgende calls en bijbehorende functies moeten geïmplementeerd worden.

1. `get_board/0`, dit geeft het huidige board terug, deze moet je kunnen omzetten naar een format string met `show_board/1`
2. `show_board/1`, dit zet een board om naar een format string representatie, welke je met `fwrite/1` kan printen. **Let op**: `show_board/1` moet precies terug geven wat wij verwachten, kijk daarom in de tests voor wat voorbeelden.
3. `restart/0` en `restart/1`, dit reset het spel, zodat je weer een nieuw potje kan spelen. Waarbij `restart/1` het mogelijk maakt om met een bepaalde configuratie van het bord te spelen.
4. `move/2`, dit neemt een x en y coördinaat als argument en plaatst daar een rondje(hoofdletter o) als speler, mits de speler niet heeft gewonnen plaatst dit ook voor de computer(hoofdletter x) een vakje. Als een speler of computer een move maakt in een vakje dat al bezet is moet je de atom `not_open` terug geven. Als een speler of computer een move maakt in een ongeldige locatie, bijvoorbeeld (-10, -10), dan moet je de atom `not_valid` terug geven. Wanneer het spel is gewonnen moet er de volgende tuple terug gegeven worden,`{won,1}` als jij gewonnen hebt en `{won,2}` als de computer gewonnen heeft. In alle andere gevallen kan je `ok` teruggeven.
5. `is_finished/0`, dit geeft aan of het spel afgelopen is.

Bij deze functies moet de volgende interne representatie gebruikt worden voor het bord en de zetten. Het bord is een lijst met daarin tuples van de volgende vorm: `{X-position,Y-position,PlayerID}`, waar de `PlayerID` 1 is voor jouw en 2 voor de computer. Stel, we hebben een bord met daarin een zet van jouw op plek $(0, 2)$, dus de x positie is 0 en de y positie is 2. Ook hebben we een zet van de computer op plek $(0, 1)$. Dan hebben we het volgende bord: `[{0,2,1},{0,1,2}]`, waarbij de eerste zet(die van ons) de eerste tuple in de lijst is. Die `[{0,2,1},{0,1,2}]` is dus wat de functie `get_board/0` moet terug geven.

Dit bord kan dan geprint worden met `print_board/1`, welke het volgende resultaat moet geven:

Shell snippet

```
1> c(tictactoe).
{ok,tictactoe}
2> tictactoe:print_board([{0,2,1},{0,1,2}]).
 | |
------
X| |
------
O| |
ok
3>
```

Vervolgens kan je het spel spelen in de erl shell door de server die `tictactoe` aan te roepen met de berichten die je hier boven geïmplementeerd hebt.

Kamertje verhuur

Voor deze opgave ga je het spel kamertje verhuur maken met behulp van computer processen. Om het spel te kunnen spelen heb je een raster nodig bestaande uit m bij n punten. De beide spelers bouwen om de beurt een muur, die voorgesteld wordt als een lijn tussen twee naburige punten, met als doel een vierkant te vormen, m.a.w. een kamer te bouwen. Iedere keer dat er een kamer vervolledigd wordt, krijgt de speler die als laatste een lijn gezet heeft een punt. Als uiteindelijk alle kamers verhuurd zijn, dan wint de speler met de meeste punten.

De spelregels zijn dus als volgt:

- Het spel wordt gespeeld op een raster van zes bij zes groot.
- Iedere speler zet om de beurt een lijn tussen twee naburige punten.
- Als een hokje van één bij één gevormd is, dan levert dit een punt op voor de speler die als laatste een lijn heeft gezet.
- Als een speler met één lijn, twee kamers kan vervolledigen, dan levert dit twee punten op.
- Als een speler een punt heeft behaald, dan mag deze nog een zet doen.
- Als alle kamers verhuurd zijn, dan wint de speler met de meeste kamers.

Voor meer informatie, zie `http://nl.wikipedia.org/wiki/Kamertje_verhuren_(spel)`

De spelers

Om de twee spelers te simuleren wordt er gebruik gemaakt van computer processen. Deze processen moeten zich als volgt gedragen:

- Als er binnen een beurt een kamer vervolledigd kan worden, dan zet de speler hier zijn lijntje neer om een punt te behalen.

- Als er geen kamer vervolledig kan worden, dan zet de computer een lijn op een willekeurige onbezette plek in het speelveld.

Opdracht 1: het speelveld (Ontwikkelend)

De eerste stap is het implementeren van een zinnige data structuur om het raster met de muren in op te slaan. Een handige manier om de muren te representeren is bv. door de coördinaten van de cellen waar de muur tussen ligt op te slaan in een tuple. bv. de tuple `{{4,3},{5,3}}` stelt de muur voor ten oosten van de cel op coördinaat $(4, 3)$ of ten westen van de cel op coördinaat $(5, 3)$. Implementeer de functie `get_wall(X,Y,Dir)` waarbij `Dir` een van de volgende atoms is: `north`, `east`, `south`, `west`. Deze functie geeft een tuple terug als volgt:

Shell snippet

```
1> c(rooms).
{ok,rooms}
2> rooms:get_wall(1, 3, north).
{{1,2},{1,3}}
3> rooms:get_wall(4, 3, east).
{{4,3},{5,3}}
4> rooms:get_wall(0, 0, north).
{{0,-1},{0,0}}
```

Het raster kunnen we voorstellen als een tuple met de breedte van het veld, de hoogte van het veld, en een lijst van gezette muren. Implementeer de `new_grid(Width,Height)` functie die een leeg raster aanmaakt.

Implementeer ook de functie `has_wall(X,Y,Dir,Grid)`, die `true` geeft als een bepaalde muur gebouwd is, en de functie `add_wall(X,Y,Dir,Grid)`, die een nieuw raster teruggeeft met de gebouwde muur.

Shell snippet

```
1> c(rooms).
{ok,rooms}
2> Grid = rooms:new_grid(6, 6).
{6,6,[]}
3> Grid2 = rooms:add_wall(4, 3, east, Grid).
{6,6,[{{4,3},{5,3}}]}
4> rooms:has_wall(5, 3, west, Grid).
false
5> rooms:has_wall(5, 3, west, Grid2).
true
```

> Bekijk voor het maken van deze opdracht de documentatie van de `lists` (`http://www.erlang.org/doc/man/lists.html`) module.

Opdracht 2: het speelveld visualiseren (vaardig)

Nu dat we een data structuur hebben gemaakt voor het raster, moeten we het raster kunnen visualiseren. Dit gaan we doen door muren weer te geven met streepjes, waarbij '`--`' gebruikt wordt voor horizontale muren en '`|`' voor verticale muren. De knooppunten doen we met plusjes ('`+`'). Implementeer een `print_grid(Grid)` functie die het raster print. Voor deze functie moet je twee helpers schrijven, namelijk `show_hlines(Row,Grid)` en `show_vlines(Row,Grid)` die respectievelijk de horizontale en de verticale lijntjes omzetten naar een format string. Je kan deze format strings dan printen in de `print_grid(Grid)` functie. De `show_hlines/2` en `show_vlines/2` functies worden getest. Het is daarom, net zoals in de individuele opgave, belangrijk dat het gegeven formaat precies gevolgd wordt. Vergeet dus niet op tijd op Codegrade te testen en de lokale tests te gebruiken.

Shell snippet

```
1> c(rooms).
{ok,rooms}
```

```
2> Grid = rooms:new_grid(3, 3).
{3,3,[]}
3> Grid2 = rooms:add_wall(1, 0, west, Grid).
{3,3,[{{0,0},{1,0}}]}
4> Grid3 = rooms:add_wall(0, 1, east, Grid2).
{3,3,[{{0,0},{1,0}},{{0,1},{1,1}}]}
5> Grid4 = rooms:add_wall(1, 0, south, Grid3).
{3,3,[{{0,0},{1,0}},{{0,1},{1,1}},{{1,0},{1,1}}]}
6> Grid5 = rooms:add_wall(0, 1, north, Grid4).
{3,3,
 [{{0,0},{0,1}},{{0,0},{1,0}},{{0,1},{1,1}},{{1,0},{1,1}}]}
7> rooms:print_grid(Grid5).

Game board:

+  +  +  +
   |
+--+--+  +
   |
+  +  +  +

+  +  +  +

ok
8> Grid6 = rooms:add_wall(0, 2, north, Grid5).
{3,3, [{{0,0},{0,1}}, {{0,0},{1,0}}, {{0,1},{0,2}}, {{0,1},{1,1}}, {{1,0},{1,1}}]}
9> rooms:print_grid(Grid6).

Game board:

+  +  +  +
   |
+--+--+  +
   |
+--+  +  +

+  +  +  +

ok
```

> Bekijk voor het maken van deze opdracht de documentatie van de `io` (`http://www.erlang.org/doc/man/io.html`) en `string` (`http://www.erlang.org/doc/man/string.html`) modules.

Opdracht 3: willekeurige muren plaatsen (competent)

Het gedrag van de AI bestaat uit twee deelstappen. De eerste deelstap, het plaatsen van een muur naar willekeur, gaan we nu behandelen. Om die deelstap te implementeren gaan we stapsgewijs functies implementeren totdat we een willekeurige muur kunnen bouwen.

We willen eerst kunnen achterhalen welke muren er nog gebouwd kunnen worden. Om dit te doen willen we eerst beginnen met een eenvoudige hulpfunctie `get_cell_walls(X,Y)` die voor een gegeven cel de lijst van alle mogelijke muren om die cel heen geeft. Met deze hulpfunctie kan dan `get_all_walls(W,H)` geïmplementeerd worden die voor een $(W \times H)$ grid alle mogelijke muren geeft (let op dat je lijst geen dubbele muren bevat). Door de muren die gebouwd zijn van de lijst van alle muren af te trekken kan dan een lijst van alle muren die nog gebouwd moeten worden gevonden worden. Implementeer dit als de functie `get_open_spots(Grid)` die een lijst van alle muren die nog gebouwd moeten worden teruggeeft.

Nu dat we een lijst van alle muren die nog gebouwd moeten worden kunnen krijgen, kunnen we de `choose_random_wall(Grid)` functie implementeren. Deze functie geeft een willekeurige plaats terug waar nog geen muur gebouwd is.

Tenslotte kan dan de functie `build_random_wall(Grid)` geïmplementeerd worden die een willekeurige muur bouwt voor een gegeven grid.

> Bekijk voor het maken van deze opdracht de documentatie van de `random` (`http://www.erlang.org/doc/man/rand.html`) module. Met name de `rand:uniform` functie.

Opdracht 4: kamers vervolledigen (gevorderde)

De tweede deelstap van de AI is om kamers te kunnen vervolledigen. Hiervoor moet er voor iedere kamer bepaald kunnen worden of er nog maar één muur gebouwd hoeft te worden om deze te vervolledigen. Implementeer de functie `get_open_cell_walls(X,Y,Grid)` die voor een gegeven cel de plekken geeft waar nog muren gebouwd kunnen worden.

Nu kunnen we voor iedere kamer kijken of deze vervolledigbaar is door te kijken of er nog één enkele muur gebouwd hoeft te worden. Implementeer de `get_completable_walls(Grid)` functie die een lijst van muren geeft die een kamer vervolledigen als ze gebouwd worden.

Implementeer vervolgens de `get_completable_wall(Grid)` functie die een muur teruggeeft die als deze gebouwd wordt een kamer vervolledigt.

Combineer deze twee deelstappen door de `build_wall(Grid)` functie te implementeren, die muur bouwt om een kamer te vervolledigen of een willekeurige muur als er geen kamer vervolledigd kan worden binnen een enkele beurt.

Opdracht 5: de spelers (gevorderde)

Nu dat we het grid en de logica om muren te plaatsen geïmplementeerd hebben, kunnen we `player()` processen voor iedere individuele speler aanmaken. Hierbij is het belangrijk dat ieder proces, inclusief het `start()` proces, de pseudo random number generator seed met een willekeurige waarde. Hiervoor kan de onderstaande code gebruikt worden:

Seeding the pseudo random number generator.

```
<<S1:32, S2:32, S3:32>> = crypto:strong_rand_bytes(12),
rand:seed(exs1024,{S1, S2, S3}),
```

Bij het aanmaken van de processen is het belangrijk om na te denken wat voor een berichten je heen en weer wilt sturen en welke state je daarbij wilt meegeven. Let er vooral op dat de proces ID's van de processen naar waar berichten gestuurd kunnen worden bekend zijn. Als je wilt kan je kijken om het te implementeren als een `gen_server`, maar denk dan goed na hoe je het allemaal vorm geeft en zorg ervoor dat de tests uit vorige delen nog steeds slagen.

Opdracht 6: meer spelers (expert)

Normaal gesproken speel je kamertje verhuren met twee spelers. Maar het is natuurlijk veel leuker als dat met N spelers kan. Voeg aan `start()` toe dat hij vraagt om hoeveel spelers er nodig zijn. Gebruik de functies `io:get_line/1` en `erlang:list_to_integer/1` om het aantal spelers op te vragen. Daarnaast is het ook leuker als je het spel met je vrienden kan spelen in plaats van twee of meer computers tegen elkaar. Bereid het spel uit dat bij het opstarten van het spel de gebruiker per speler de keuze krijgt of dit een AI is of dat de gebruiker zelf de muren plaatst. Gebruikt de x, y coördinatoren van de vakken in combinatie met de richtingen n, w, z, o voor de locatie van de muren.