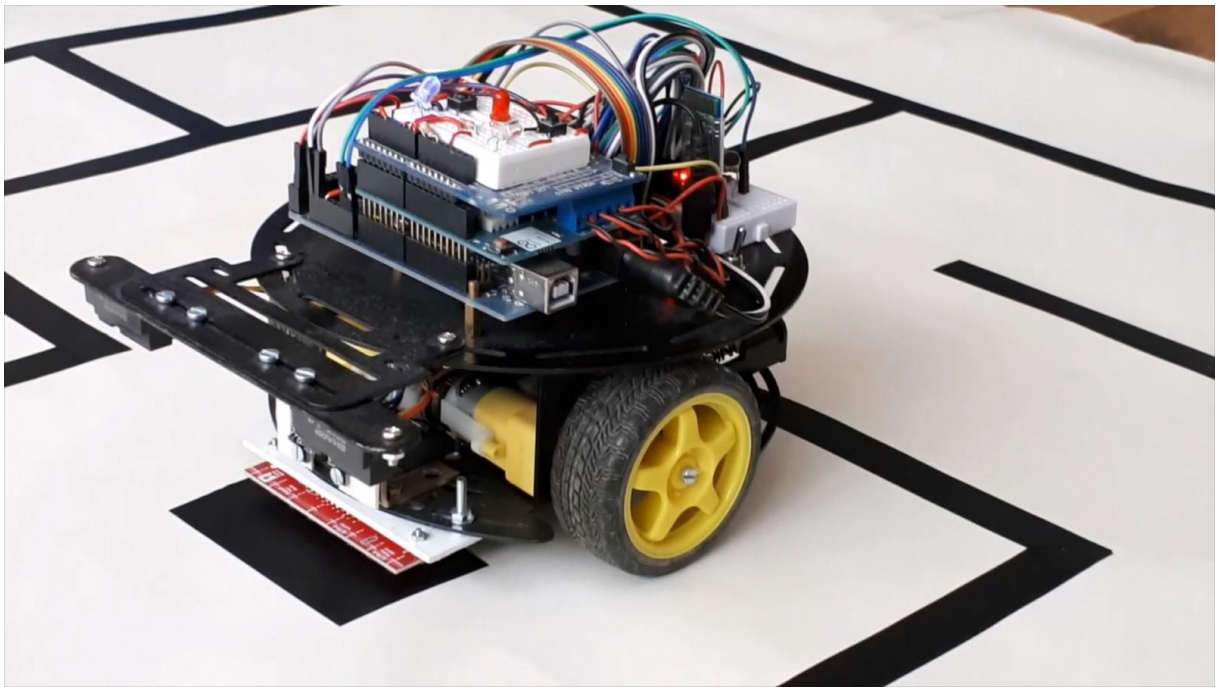
Een doolhof oplossen met een robot

In dit artikel wil ik mijn ervaringen met jullie delen met het bouwen van een doolhof robot. In de Corona periode, die mij net als iedereen in een huismus had veranderd, was ik op zoek naar een nieuw type uitdaging. Op de bijeenkomsten in de Dissel heb ik met bewondering en enige jaloezie gekeken wat enkele IG-leden konden met hun robots in een doolhof. Dat zag er erg intelligent uit, en het leek me wel wat om dat ook te proberen.

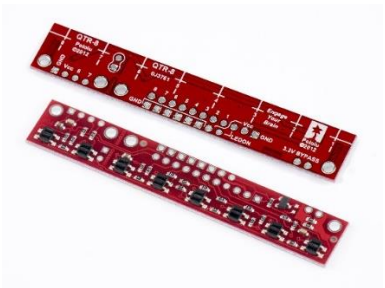


Weten jullie trouwens dat de Robotica IG een officieel Doolhof reglement kent?
https://robotica.hcc.nl/downloads/file/110-regelementen.html

Dat reglement beschrijft onder andere dat het labyrint bestaat uit een reeks van zwarte lijnen op een wit veld. Alle lijnen liggen op een (denkbeeldig) raster van 150 mm.

Maar waar te beginnen? Als eerste ging ik maar eens na wat voor technieken ik hierbij zou kunnen inzetten die ik al kende:
- Een rijdende robot bouwen, en omgaan met motor-encodersignalen
- Via bluetooth sensordata en interne informatie op afstand uitlezen met een HC-05 bluetooth module
- Uitlezen van een lijnvolgsensor,

Ik deed aan de Lockdown Challenges mee met een Turtle robotplatform van DFRobot, voorzien van 6v motoren met redelijk bruikbare 6V motoren met encoders (… pulsen per omwenteling). Deze is wel iets te groot om mee te dingen in een officiële competitie, maar dat mag de pret niet drukken. Het voordeel van deze robot, die ik TurboTurtle heb gedoopt, was wel dat hij al beschikte over een Pololu QTR-8a 8-voudige array van lijnvolgsensoren.



Alles wat mij te doen stond was het maken van een doolhofprogramma.

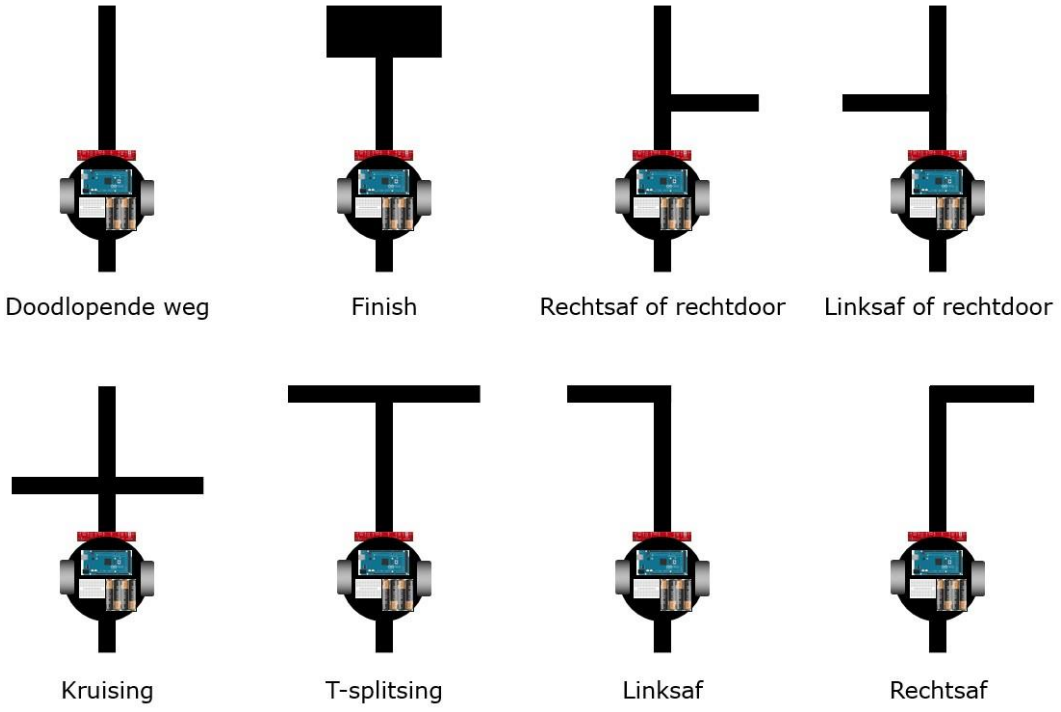
Als programmeeromgeving bleef ik bij de voor mij vertrouwde Arduino IDE. Sommigen missen hierin de geavanceerdere functies die andere ontwikkelomgevingen je kunnen bieden, maar de Arduino omgeving biedt veel libraries, ik ken er de weg in en ik heb er veel code in die ik kan hergebruiken.

Vervolgens ging ik mij verdiepen in het algoritme om een doolhof op te lossen. En daar kwam de HCC Robotica-groep natuurlijk heel goed van pas. Al snel wezen vriendelijke helpers mij op de linkerhandregel: als je je hand steeds tegen de linkerwand houdt, zul je uiteindelijk het hele doolhof bezoeken. Google op 'linkerhandregel doolhof' of 'left hand rule maze' levert een heleboel leuke en nuttige voorbeelden op. De linkerhandregel heeft wel een beperking: De oplossing werkt alleen voor 'eenvoudig verbonden' doolhoven. Dat is de benaming voor een doolhof waarbij alle muren met elkaar zijn verbonden. In een doolhof dat bestaat uit zwarte lijnen komt dit erop neer dat er geen lussen in het doolhof voorkomen. Maar dat geldt in de praktijk voor een heleboel doolhoven.

Het oplossen van een doolhof bestaat in de basis uit 2 stappen. De eerste stap is de verkenning van het doolhof om de locatie van de finish te vinden. Hierbij volgt TurboTurtle de linkerhandregel. In de tweede stap wordt TurboTurtle op de oorspronkelijke beginpositie gezet, waarna hij zelfstandig de kortste weg door het doolhof naar de finish kiest.

HET PROGRAMMA VOOR FASE 1: VERKENNING VAN HET DOOLHOF

Om de robot een lijnendoolhof te laten rijden , moet je natuurlijk ook met je robot een lijn kunnen volgen. Daarvoor besloot ik om een PID-regeling toe te passen. Op zich al weer een kleine challenge, want die had ik nog nooit echt gebruikt (behalve als kant-en-klaar demonstratie programmaatje). Dat had makkelijker gekund, want een rechte lijn volgen zou best eenvoudiger kunnen dan met PID. Hoe die programmacode eruit ziet is misschien iets voor een ander artikel. De volgende uitdaging is het afhandelen van de punten waarop de weg verandert. Als je daar over nadenkt zul je ontdekken dat er acht mogelijke vormen voor een knooppunt zijn:



Hoe bepaalt TurboTurtle met welk knooppunt hij te maken heeft?:

De eerste detectiestap die de robot doet is of de weg is doodgelopen. Het is logisch om als eerste op deze situatie te testen omdat hij meteen eenduidig te herkennen is. De lijnvolgsensor registreert namelijk ineens 0 hits. En de actie als je een doodlopende weg detecteert is ook duidelijk: Stop en keer om.

Voor de overige situaties zal de robot wat vervolgonderzoek moeten doen. Want zie je bijvoorbeeld een aftakking naar rechts, dan kan je misschien ook nog rechtdoor op dat punt. En zie je zowel een aftakking naar links als naar rechts, dan kan dat zowel een kruising als een T-splitsing zijn.

Als de testroutine het type knooppunt heeft bepaald dan reageert de robot volgens het volgende schema, wat in feite de uitwerking van de linkerhandregel is:

Knooppunt	Code voor de situatie	Actie
Doodlopende weg	*B*	*Keer om*
Finish	*EOM*	*Stop en initieer Fase 2*
Rechtsaf of rechtdoor	*RS*	*Rij rechtdoor*
Linksaf of rechtdoor	*LS*	*Sla linksaf*
Kruising	*LRS*	*Sla linksaf*
T-splitsing	*T*	*Sla linksaf*
Linksaf	*L*	*Sla linksaf*
Rechtsaf	*R*	*Sla rechtsaf*

De betekenis van de lettercodes:

L = Links

R = Rechts

B = Back (rechtsomkeert)

S = Straight (rechtdoor)

T = T-splising

EOM = End Of Maze (einde doolhof)

Hoe vertaalt zich dit dan in programmacode? Laat ik beginnen met te zeggen dat de programmacode van TurboTurtle werkt volgens het principe van een State machine. De meeste tijd verkeert de robot in de State 10: volg de lijn. Tijden het volgen van de lijn test de robot voortdurend of er zich een verandering voordoet

1. *Loopt het pad dood? Ga naar de State 180 voor rechtsomkeert maken*
2. *Doet er zich een aftakking voor? Bepaal dan het type kruising*
3. *Heb je het einde van het doolhof bereikt? Stop en stap over naar programmafase 2 (State 300)*
4. *Als zich een aftakking voordoet, kun je dan rechtdoor? (situatie RS) Behoud de huidige State, en blijf gewoon rechtdoor rijden.*

Eindconditie State 10:

```
if (hits == 0) {                                    // test op doodlopende weg
  Motors(0,0);
  Serial3.println(F("  State 10: Doodlopende weg, keer om!"));
  State = 180;
} else if(sensorValues[1] > QTRZWART || sensorValues[NUM_SENSORS-2] > QTRZWART) {
                                                    // test op afslag, splitsing of kruising
  Richting = RichtingStore.read();
  TestJ_uitslag = testJunction(sensorValues);   // bepaal type kruising
  if (TestJ_uitslag == JNC_EOM) {
    State = 300;                                    // Einde doolhof bereikt
  } else if (TestJ_uitslag != JNC_RS) {
    State = 11;
  }
}
```

De resterende testen, die in de functie testJunction() zijn geïmplementeerd, verlopen dan als volgt:

Als het knooppunt dus niet een doodlopende weg is , gaat de testfunctie lopen die het type kruising bepaalt en retourneert. Die test verloopt in de volgende stappen:

1. *kijk met de lijnsensor of er links en rechts afslagen zijn,*
2. *rijd een stukje rechtuit,*
3. *kijk met de lijnsensor of het pad rechtdoor mogelijk is.*
4. *Bepaal d.m.v. de verzamelde informatie het type kruising en retourneer die waarde.*

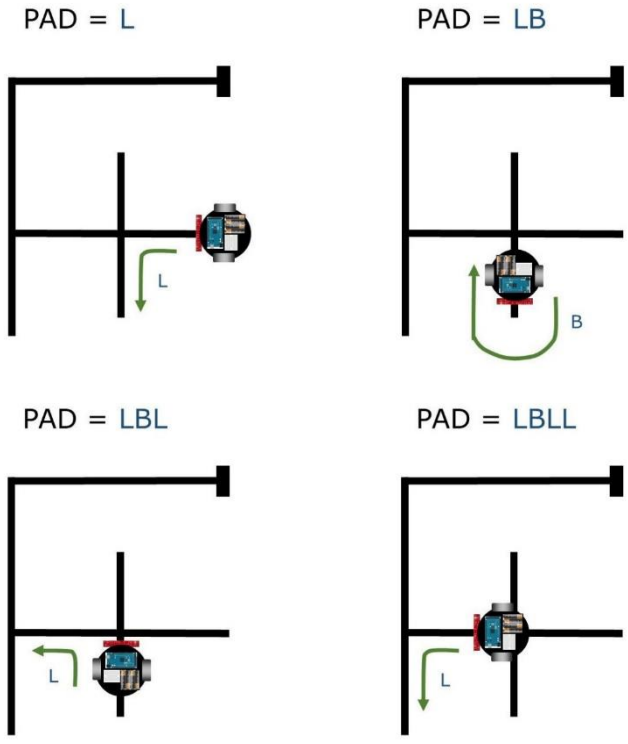
Is het niet een van de reeds genoemde gevallen, dan weet je dat je in ieder geval een afslag links of rechtsaf zult moeten nemen. Daarom wordt eerst een 'tussen'state 11 uitgevoerd om een klein stukje rechtdoor te rijden om goed uit te komen bij het maken van de draai. De State Machine kiest vervolgens aan de hand van de geretourneerde waarde met behulp van een switch-case statement de volgende State.

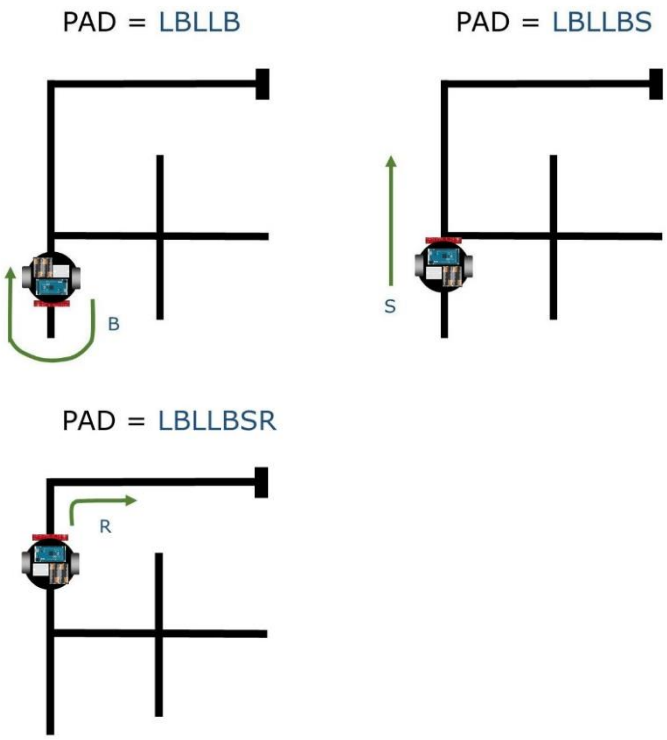
Eindconditie State 11:

```
switch (TestJ_uitslag) {
  case JNC_LS  :
  case JNC_LRS :
  case JNC_T   :
  case JNC_L   :
    {
      State = 90;                          // 90 graden linksaf
      break;
    }
  case JNC_R   :
    {
      State = 100;                         // 90 graden rechtsaf
      break;
    }
  }    // end switch
```

VOORBEELD WERKING LINKERHAND ALGORITME

Als je dit algoritme toepast op een eenvoudig voorbeeld-doolhof levert dat het volgende resultaat op:





OK, nu de robot dit voor elkaar heeft hebben we een pad. Het pad wordt in het geheugen opgeslagen door op iedere kruising de actie op te slaan die de robot neemt. Daarvoor maken we de volgende variabele (array) aan:

```
char path[100] = " ";
```

We declareren ook 2 index variabelen voor gebruik bij de array:

```
unsigned char pathLength = 0; // de lengte van het pad
int pathIndex = 0;      // index om een specifiek element van de array te kunnen benaderen.
```

Als we bovenstaand voorbeeld uitvoeren eindigen we met:

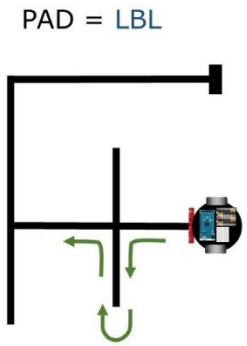
path = [LBLLBSR]
en pathLengh = 7

FASE 2: HET OPTIMALISEREN VAN HET PAD

De volgende uitdaging is het algoritme om dit pad te vereenvoudigen tot een korter pad.

Laten we kijken naar de eerste foute afslag die we namen. Een "B" (voor "Back") betekent dat de robot rechtsomkeert maakte - hij had dus een doodlopende afslag genomen. We kunnen het pad dus optimaliseren door op een of andere wijze die "B" door iets anders te vervangen.

Neem de eerste 3 acties in het pad van het voorbeeld. Die zijn "LBL". In totaal ziet dat er als volgt uit:



In plaats van linksaf te slaan, rechtsomkeert te maken en weer linksaf te slaan, had de robot dus beter meteen rechtdoor kunnen gaan. Dus kunnen we stellen dat LBL => S. Deze vervanging is wat de robot gebruikt om het pad te verkorten. De hele lijst van substituties ziet er als volgt uit:

LBL → S
LBR → B
LBS → R
RBL → B
SBL → R
SBS → B
LBL → S

Als je zin hebt, kun je ze op papier uittekenen om te zien hoe ze werken.

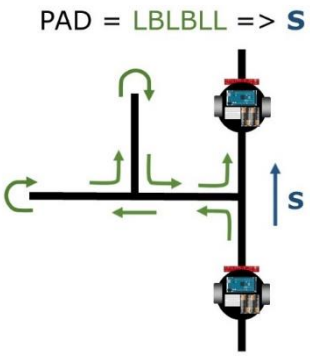
Voor het codevoorbeeld, kun je de volgende webpagina raadplegen:
https://create.arduino.cc/projecthub/mjrobot/maze-solver-robot-using-artificial-intelligence-4318cf

Hoe vertaalt zich dit dan weer in Arduino C-code? Daarvoor bevat het programma een functie die verantwoordelijk is voor de conversies. Hoewel het robotprogramma eromheen eigen ontwikkeling is, kon ik deze functie letterlijk overnemen van het codevoorbeeld. Daarin zul je niet direct de letteromzettingen terug zien komen. De bedenker heeft namelijk een slimme manier verzonnen om de conversies te doen. Hij vertaalde de draaien links, rechts en rechtsomkeert naar een hoek in graden van 90, 270 resp. 180. Als je die optelt modulo 360 rolt de gewenste draaihoek er uit, die zich weer terugvertaalt naar L, R, of B.

```
void simplifyPath()
{
  // only simplify the path if the second-to-last turn was a 'B'
  if(pathLength < 3 || path[pathLength-2] != 'B')
    return;

  int totalAngle = 0;
  int i;
  for(i=1;i<=3;i++)
  {
    switch(path[pathLength-i])
    {
    case 'R':
      totalAngle += 90;
      break;
    case 'L':
      totalAngle += 270;
      break;
    case 'B':
      totalAngle += 180;
      break;
    }
  }

  // Get the angle as a number between 0 and 360 degrees.
  totalAngle = totalAngle % 360;

  // Replace all of those turns with a single one.
  switch(totalAngle)
  {
  case 0:
    path[pathLength - 3] = 'S';
    break;
  case 90:
    path[pathLength - 3] = 'R';
    break;
  case 180:
    path[pathLength - 3] = 'B';
    break;
  case 270:
    path[pathLength - 3] = 'L';
    break;
  }

  // The path is now two steps shorter.
  pathLength -= 2;
}
```

Het is natuurlijk prima mogelijk om een eigen versie van simplifyPath() implementeren die hetzelfde doet op basis van het herkennen van de letterreeksen. Ik was daar echter te lui voor ;-).

Wanneer wordt de functie simplifyPath() nu in het programma aangeroepen? Als de robot het doolhof verkent, kijkt hij bij elk knooppunt of hij het pad kan optimaliseren, door te testen of de laatste drie gevonden knooppunten voldoen aan één van deze substitutieregels. De werking van het algoritme kun je volgen aan de hand van het volgende stukje doolhof:



Het gewenste eindresultaat van de optimalisatie is natuurlijk dat de robot de gehele aftakking negeert en bij het eerste knooppunt rechtdoor gaat. Het pad evolueert als volgt bij toepassen van het algoritme:

L
LS
LSB
*L**SBL** → simplifyPath() → L**R***
LRB
*L**RBL** → simplifyPath() → L**B***
LBL** → simplifyPath() → **S

En voilà, het resultaat is bereikt. Als dit eenmaal geprogrammeerd is kan de robot ook heel ingewikkeld ogende doolhoven aan. Als er maar geen lussen in voorkomen.

Het uitvoeren van Fase 2 is dan in feite slechts het volgen van het geoptimaliseerde pad.

Tot slot van dit artikel een link naar een Youtube-filmpje van mijn doolhofrobot in actie:
https://youtu.be/qSSniuzVxpq

Ik hoop dat jullie het interessant om vonden dit te lezen. Ik moet er wel bij zeggen dat het er op papier heel logisch uit ziet, maar in de praktijk bleek er nog aardig wat 'tuning' bij te komen kijken om de robot succesvol bij de finish te laten aankomen. Meer over die praktijkervaringen in een volgend artikel.

Februari 2021, Ewoud Hüttner