Individueel onderdeel; technische beschrijving

Voor het individuele onderdeel heb ik diverse aanpassingen gemaakt en toegevoegd aan ons project over Rush hour. Ten eerste de functie ‘draw’ in car.py. Deze functie zorgt ervoor dat er een auto op een bestaand speelbord wordt gezet. Ook heb ik de init functie geschreven in board.py. Dit zorgt ervoor dat er een bord wordt aangemaakt om het spel op te spelen. Een opvallend stukje code die ik hier heb gemaakt, is de grootte van het speelbord uit de titel halen. De grootte van het speelbord staat namelijk in de titel van de datafiles vermeld. Daarnaast heb ik in deze class ook de functie init_cars geschreven, wat ervoor zorgt dat alle auto’s voor het spel worden geïnitialiseerd met de juiste grootte en oriëntatie (horizontaal of verticaal). Tot slot heb ik de draw functie in board.py gecodeerd. Deze zorgt ervoor dat het speelbord op de juiste manier wordt weergegeven met de juiste plek waar de rode auto ook van het bord zou moeten afrijden.

Naast het schrijven van de zojuist genoemde code heb ik het eerste experiment uitgevoerd. Dit betrof het random experiment en werd 10.000 keer uitgevoerd op het kleinste speelbord. Hier hebben we veel data uit verkregen en dit hebben Esmee en ik samen in een grafiek gezet. De belangrijke informatie zoals het gemiddelde en de top 5 beste oplossingen heb ik hierbij ook bewaard zodat we dit later kunnen vergelijken met de prestatie van onze eigen algoritmes.

Er hoeft eigenlijk niet echt meer iets te gebeuren aan de classes die we hebben geschreven. We staan nu voor de opdracht om verschillende algoritmes uit te gaan werken om de best mogelijke oplossing te gaan vinden. We willen dit doen door zelfstandig allemaal aan een ander algoritme te werken om zo uiteindelijk verschillende algoritmes te kunnen testen.
Ik ga zelf werken aan een greedy algoritme. Het greedy algoritme wil ik als volgt vormgeven. Het gaat erom dat het algoritme ‘greedy’ is om de rode auto uit het speelbord te laten rijden. Dit betekent dat als de rode auto naar voren kan, deze altijd naar voren wordt geschoven. Als dat niet kan dan wordt er gekeken welke auto er in de weg staat en wordt die verschoven als dit mogelijk is. Kan dit niet? Dan wordt er gekeken naar de auto die daar weer voor staat enzovoorts. We vinden het leuk om een heuristiek te onderzoeken die wij tijdens het spelen van het spel hebben gevonden. Deze heuristiek houdt in dat de rode auto eigenlijk altijd eerste naar achter moet om het spel zo snel mogelijk uit te spelen. We willen dus in het greedy algoritme de resultaten van deze twee heuristieken met elkaar vergelijken; de rode auto moet naar voren als deze naar voren kan en de rode auto moet eerst naar achter.

Om de greedy te implementeren wil ik beginnen met het aanroepen van het bord. Vervolgens is de eerste stap om te gaan kijken of de rode auto vooruit kan worden gezet.
Kan de rode auto vooruit? Als deze vooruit kan dan gaat die vooruit, als dat niet kan dan wordt met de functie obstructed_by gekeken welke auto er in de weg staat en returnt dit car object. In het lijstje van get_moves staat of deze auto te verplaatsen is. In het geval dat de auto niet kan verplaatsen dan zit de auto of tegen muur en return none of hij is ook obstructed door een andere auto (of aan beide obstructed door twee auto’s). als de auto helemaal niet kan verplaatsen wordt er door middel van random gekeken of er naar de auto aan de voor of achterzijde gekeken gaat worden. Dit doen we om bias te voorkomen. Bij een stap die kan om een obstructie weg te halen, moet die move niet in de weg staan VOOR de rode auto. Nu wordt er een nieuwe get_moves aangeroepen. Er wordt weer naar de auto’s gekeken in tegengestelde richting of de auto’s nu met de nieuwe move wel kunnen bewegen.

Recursive functie: obstructing car en obstructed car.
Obstruction – length or obstruction + 1