**Baseline**

***Ons doel:***

Om te beginnen willen we ons doel definiëren: in zo’n kort mogelijke tijd tot de beste oplossing van Rush Hour komen met de minst mogelijke stappen. In ons geval gaat dit over stappen en geen moves, dus een auto beweegt altijd maar één vakje per keer.

***Wat hebben we al gedaan:***

We hebben al best veel tijd in de visualisatie gestopt, dit is praktisch af. Daarnaast hebben we een solver geïmplementeerd met een methode om meerdere keren een algoritme te runnen en te importeren via de CLI (command line interface). Je kan daarbij kiezen welke solver je wilt gebruiken, hoe vaak je het spel wilt laten oplossen, of je elke stap wilt visualiseren of niet en het geeft na het oplossen de statistieken van het oplossingsproces weer. Verder heeft game.py (de klasse Game en het script om de game via de CLI te spelen) de mogelijkheid om te kiezen welke game je wilt spelen. Als laatst is de README ook helemaal bijgewerkt.

***Wat we nog moeten doen:***

Het belangrijkste onderdeel waar we nog op moeten focussen is het bedenken van meer solvers. Daarbij moeten we gaan nadenken over uniforme eisen voor de solvers. Een voorbeeld kan zijn dat de ‘always improve solver’ eigenlijk onder de streep het spel maar één keer oplost omdat het algoritme de vorige oplossen gebruikt als informatie en dat is eigenlijk niet helemaal de bedoeling.

***De nieuwe eisen van onze solver:***

Wij hebben nagedacht over verschillende eisen die wij aan onze solver willen stellen:

* Bij het herhalen van een solver (steekproef) via solver.py moet de solver echt opnieuw beginnen (dus het mag geen oude output.csv gebruiken).
* Hij mag opslag gebruiken tot 2GB (peak memory usage). Hiervoor gaan we dit gebruiken: <https://www.geeksforgeeks.org/monitoring-memory-usage-of-a-running-python-program/>
* Solvers moeten ook voldoen aan style50, flake8 & type hints.
* Voor vergelijkingsmateriaal: de solver moet seed(123456789) gebruiken. Maar aangeraden wordt om bij het testen dit nog niet te doen om te kijken hoe de solver werkt in verschillende methoden.
* Resultaten zijn pas relevant als het experiment herhaald wordt tot het minstens 10 seconden duurt. Dit om te voorkomen dat we conclusies gaan trekken op basis van een vergelijking van bijvoorbeeld een solver die 400 ms duurt en een solver die 350 ms duurt.
* We willen alles vergelijken onder dezelfde omstandigheden (dus hetzelfde spel, dezelfde keer herhalen).

***Solverideeen:***

Als laatst hebben wij alvast nagedacht over verschillende solvers en heuristieken voor die solvers:

* Als de rode auto nog 1 plek naar rechts moet en dit kan: dit doen.
* Een zet direct omdraaien mag niet (dus dat een auto na 1 stap vooruit te gaan gelijk weer 1 stap achteruit gaat).
* Een zelf lerend programma (zoals de always\_improve\_solver, alleen valt dit eigenlijk nog niet onder de eisen, omdat deze solver output.csv gebruikt als info).
* Score geven aan moves (zoals bij schaken) en dan de beste score kiezen (met bepaalde kans?) -> lijkt op A\*.
* A\*, aantal stappen tot de oplossen proberen te berekenen.
* Een rode auto naar rechts bewegen als dit kan: dit doen met een hoge kans (maar de rode auto moet ook nog wel naar links kunnen bewegen).
* Bij kleine borden moeten bepaalde auto’s in een rij op één bepaalde specifieke plek staan voor een oplossing, en dan dit implementeren.
* Terug redeneren vanuit de oplossing? Je kunt dan bijvoorbeeld 4 stappen terug redeneren en dat allemaal opslaan: je weet dan welke zetten je moet doen als je in zo’n staat beland.
* Reken zo veel mogelijk uit (tot je memory op is) en stop dit in een graaf.