2019

Amazon simulatie: Onderzoek

VERBETERING BOVEN OP DE EISEN

GROEP 42

VERSIE:1.0R5

Inhoud

Inleiding

Voor ons project voor deze periode moesten wij een simulatie maken van een Amazon magazijn.
De minimale eisen voor dit project zijn als volgt:

- De geschreven software moet gebruikmaken van Java en Three.js (Javascript/Typescript).
- Het systeem bestaat uit een client en een server. De server simuleert centraal de wereld en die clients kunnen deze simulatie live zien.
- Het systeem maakt juist gebruik van het Model-View-Controller design pattern.
- Het magazijn bestaat minimaal uit 2 robots, 10 stellages en een vrachtwagen.
- Deze objecten bewegen in de wereld en kunnen met elkaar samenwerken.
- De robots kunnen hun weg door het magazijn vinden met behulp van een graaf, waarbij een kortst pad algoritme is geïmplementeerd.
- Het magazijn volgt ongeveer de opzet zoals gegeven is in de figuur hierboven.
- Vrachtwagens moeten verschijnen en bij het magazijn stoppen. Deze hebben producten (stellages) bij zich, welke moeten worden opgeslagen in het magazijn. Wanneer de vrachtwagen leeg is voorzien de robots deze van de stellages die de vrachtwagen weer mee moet nemen.

Ook hebben we een beetje hulp gekregen door middel van een basis opzet van het project.
Dit basis opzetje bevatte al een server en een client en een robot die vooruitrijdt.
Onze eerste stap was op te kijken wat er fundamenteel beter kon.
Hieruit ontstonden de volgende vragen.

Kunnen we Json methode van de basis vervangen voor een beter serializer.
Is er een beter en of efficiëntere manier van padfinding dan Dijkstra's algoritme op een graaf.

Na de juiste tools vast te stellen kwamen we bij de volgende "onderzoekjes".

- Wat is de snelste manier van serialisatie & de-serialisatie van berichten
- Pathfinding: een snellere oplossing.

Wat is de snelste manier van serialisatie & de-serialisatie van berichten

Voor ons project wouden wij iets beters gebruiken dan Json.
Omdat onze "in house" expert zij dat het beter kon.
Daarom om dit te onderzoeken en uit elkaar te trekken voor benchmarks moeten we eerst weten wat de meest voorkomende manieren zijn om dit voor elkaar de krijgen hiervoor hebben we drie mogelijkheden.

1. Json
2. Messagepack
3. Protobuf

Deze drie worden het meest gebruikt op het moment van dit onderzoek.
Hiervoor houden we drie tests met verschillende berichten groottes.
gelukkig hebben we al mooie grafieken voor door al gemaakte tests.

Test 1: Klein bericht

serializer	size (bytes)	serialization time (µs)	deserialization time (µs)
json	74	4.02	4.45
protobuf	24	21	12.1
messagepack	48	4.48	0.912

Figuur 1: Resultaten 1ste test (Silva, 2018)

	number of requests	median response (ms)	average response (ms)
small json	173,592	11	28
small protobuf	170,625	21	43
small messagepack	170,936	17	41

Figuur 2: Resultaten 1ste test response time (Silva, 2018)

Uit deze resultaten is er niet echt duidelijk te zien of er een nauw beter is dan de andere,
Daarom gaan we meerder test houden met verschillende bericht groottes.

Test 2: Medium bericht

message type	size (bytes)	serialization time (µs)	deserialization time (µs)
json	36,747	303	892
protobuf	22,160	8540	3940
messagepack	30,797	866	202

Figuur 3: Resultaten 2de test (Silva, 2018)

	number of requests	median response (ms)	average response (ms)
big json	146,927	170	213
big protobuf	161,907	71	96
big messagepack	152,764	130	156

Figuur 4: Resultaten 2de test response time (Silva, 2018)

We kunnen nu zien dat er verschil is in de serialisatie tijden zijn en dat de gemiddelde tijden wat verder uit elkaar liggen.
voor alle zekerheid houden we nog een test met een groot bericht.

Test 3: Groot Bericht

message type	size (bytes)	serialization time (ms)	deserialization time (ms)
json	105,770	0.971	2.63
protobuf	63,804	31.7	13
messagepack	88,684	2.53	0.614

Figuur 5: Resultaten 3de test (Silva, 2018)

	# of requests	# failed requests	median response (ms)	average response (ms)
3.1.a JSON	122,749	35,998	81	118
3.1.b Protobuf	131,556	44,024	10	16
3.1.c MessagePack	128,954	46,026	11	19

Figuur 6: Resultaten 3de test Response time (Silva, 2018)

	# of requests	# failed requests	median response (ms)	average response (ms)
3.2.a JSON	57,776	60	1600	2003
3.2.b Protobuf	122,733	0	510	458
3.2.c MessagePack	143,030	0	280	250

Figuur 7: Resultaten 3de test met aantal failed requests (Silva, 2018)

Bij grootte berichten gaat Json al snel “stuk” aangezien er 60 verzoeken van de 57,776 verzoeken gefaald
waarbij de andere twee geen enkele keer gefaald hebben en nog aardig snel waren.

Maar omdat wij geen validatie nodig hebben wij gekozen voor Messagepack omdat het ook kleiner is en daarom ook minder ruimte in neemt.

Pathfinding: een snellere oplossing.

Voor het uitzoeken van wat de kortste route is hebben we een zoek en pad vindt algoritme nodig.
Om een ruim voldoende te halen moet het minimaal met alleen het Dijkstra zoek algoritme werken.
Maar is dit de snelste oplossing?

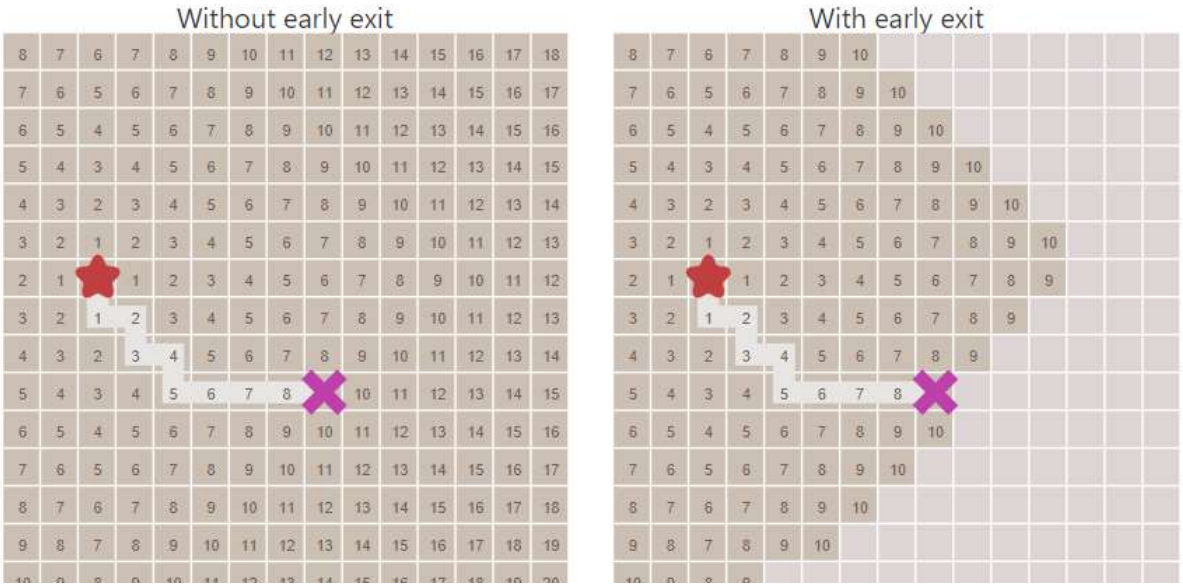
Om dit uit te vogelen moeten we eerste kijken naar wat de mogelijke opties zijn binnen zoek een pad algoritmes.

- Hard Coded (voldoende maar niet netjes en autonoom)
- Breath first (zeer inefficiënt)
- Breath first Early exit (minder inefficiënt)
- Greedy Breath first (inefficiënt met paden die gewicht hebben)
- Dijkstra algoritme (ruim voldoende maar niet zeer efficiënt met grote aantallen robots)
- A* algoritme (efficiënt voor grote aantallen robots)

De opties hierboven zijn het makkelijkst om uit te werken en te raffineren in een korte periode van tijd.
Nou is er geen juiste keuze behalve Hard ge-code want, dat is niet autonoom omdat het niet zijn eigen pad maakt en of zoekt.

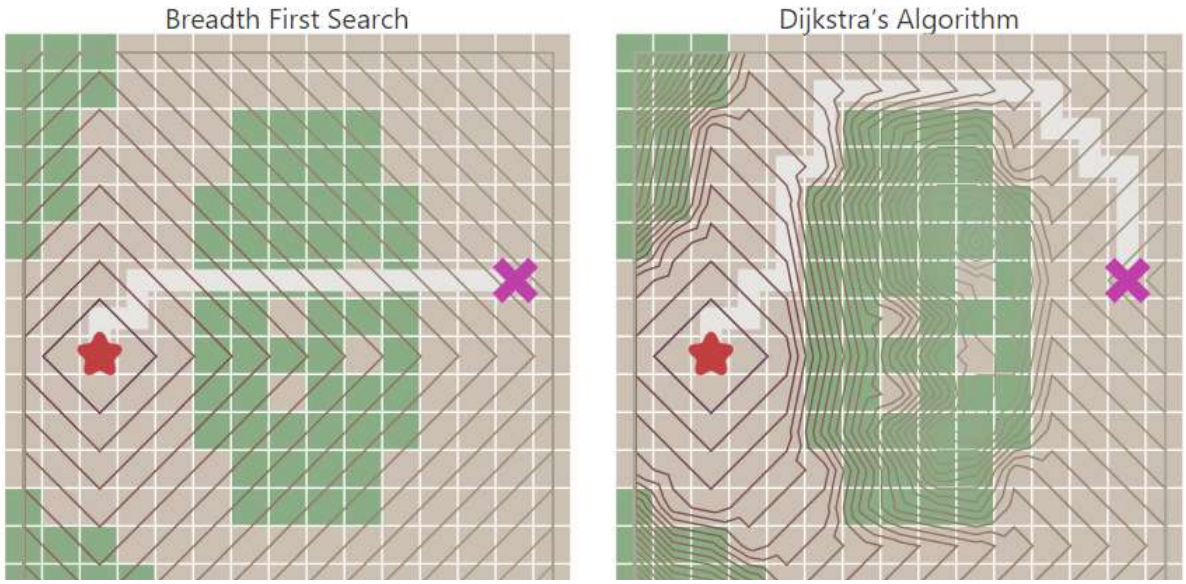
Breath First met early exit

Het eerste verschil is al snel te zien met het volgende plaatje waarbij "Breath First" met en zonder een eerder exit conditie zijn uitgewerkt.
Dit zorgt ervoor dat niet alle Nodes gecontroleerd hoeven te worden.



Figuur 8: Breath first met en zonder exit (redblobgames, 2014)

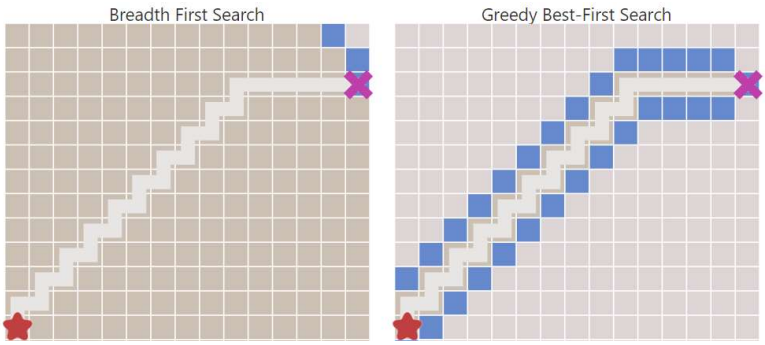
Een groter verschil is zichtbaar wanneer je van "Breath First" overstapt naar Dijkstra. Dijkstra maakt gebruik van gewicht op Nodes, dit zorgt ervoor dat er een efficiënte route wordt gemaakt naar het einddoel.



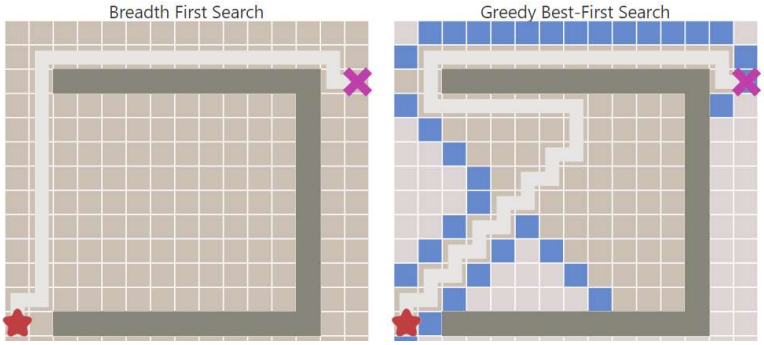
Figuur 9: Dijkstra Vs Breath First (redblobgames, 2014)

Breath First VS Greedy Breath First

Nou is een warenhuis meestal vlak en heeft het geen heuvels waardoor er geen gewicht is.
Waardoor er ook veel te veel mogelijke route opties bestaan die de efficiëntie en snelheid snel omlaag trekken.
voor gewichtloze maar betere oplossing dan "Breath First" is er ook nog "Greedy Breath First" deze manier is directer dan "Breath First" en werkt alleen goed op stelsel zonder gewicht.
dit werkt wel met een robot maar als je een hele hoop robots hebt, dan hebben zij een vrij grote kans om door elkaar heen te gaan.



Figuur 10 Greedy zonder muren (redblobgames, 2014)



Figuur 11: Greedy Breath First met muren (redblobgames, 2014)

De drie vergeleken

De “Greedy” manier controleert minder Nodes, maar vindt nog wel op een zeer directe manier de kortste route.
Dit is in simpele lay-outs van warenhuizen goed werkend maar wanneer er meer muren of andere blokkades in de weg staan zijn de routes niet altijd even efficiënt.
Dit is in figuur 12 goed te zien.

Figuur 12: Vergelijking tussen de drie (redblobgames, 2014)

Conclusie:

Wat is de snelste manier van serialisatie & de-serialisatie van berichten

Uit alle serializer tests blijkt en na eigen debat in de groep hebben we gekozen voor Messagepack.
Omdat wij geen “checks” nodig hebben die aan beiden kanten controleren of de data correct is.
Dit is daardoor ook sneller en kost minder tijd om te implementeren.
Plus het is kleiner dan protobuf en beter dan gewoon Json heen en weer te sturen.

Pathfinding: een snellere oplossing.

De minimale eis was dat de paden hard in gecodeerd werden en voor een ruim voldoende moest het minstens Dijkstra bevatten.
Uit onze vinding voor de meest optimale manier voor het vinden van een pad kwam er A* algoritme die op de meest consistente manier een pad vindt en werkt voor meeste situaties.

Bronnen:

Redblobgames. (2014, Mei 26). *Redblobgames*. Retrieved from redblobgames: https://www.redblobgames.com/pathfinding/a-star/introduction.html

Silva, H. V. (2018, Augustus 2). *Medium*. Retrieved from medium.com: https://medium.com/unbabel/the-need-for-speed-experimenting-with-message-serialization-93d7562b16e4