Software Architectures for Enterprises
Wintersemester 2024/25
Name: Philipp Geier
Matr.Nr: 1539462

Datum: 24.03.2025
Portfolio
Übungsblatt: 3

Inhaltsverzeichnis

1 Aufgabe 1 „Nur der Schnellste gewinnt“ 2

2 Aufgabe 2 Cluster 2

3 Aufgabe 3 „Ave Caesar“ 3

3.1 Zugbasierte-Abfolge: . . . 3
3.2 Belegtheit: . . . 3
3.3 Scouten: . . . 3
3.4 Mehrere Schritte/Karten: . . . 3
3.5 Runden: . . . 4
3.6 Caesar/Punktevergabe: . . . 4
3.7 Strategie . . . 4
3.8 Aufbau/Start . . . 4
3.9 Limitierungen: . . . 5
3.10 Strecken-Generator: . . . 5

Github-Repo: [2]

1 Aufgabe 1 „Nur der Schnellste gewinnt“

Ist im Ordner Aufgabe_1 zu finden. Zum Starten muss zuerst das compose.yml ausgeführt werden. Dafür je nach Docker-Version

docker-compose up/docker compose up

ausführen. Um dann das Programm zu starten, folgendes ausführen:

python run.py parcours.json

Zum Generieren der Strecke folgendes ausführen:

python circular-course.py parcours.json

Die compose.yml ist vom Github-Repository [1] angepasst.

Die run.py hat einen Producer. Das Programm liest das beigefügte json-File ein und benötigt dabei die parcours.json Struktur. Es werden dann alle Segmente segment.py initiiert und alle Segmente vom Typ „start_goal“wird ein Token als Spieler zugesandt mit „event:start“. Die Segmente haben ein producer und ein consumer und erhalten die Nachricht und schicken in handle_message eine gleiche Nachricht an das erste Segment mit der segment_id in ihrer next_segments-Liste in der Funktion „move_token“. Dabei wird die start_time am Spieler-Token mitgeschickt, sowie die Rundenanzahl, die nach jedem „start-goal“oder „caesar“erhöht wird. Sobald ein Spieler fertig mit 3 Runden ist, wird mit seiner start-time verrechnet und ausgegeben.

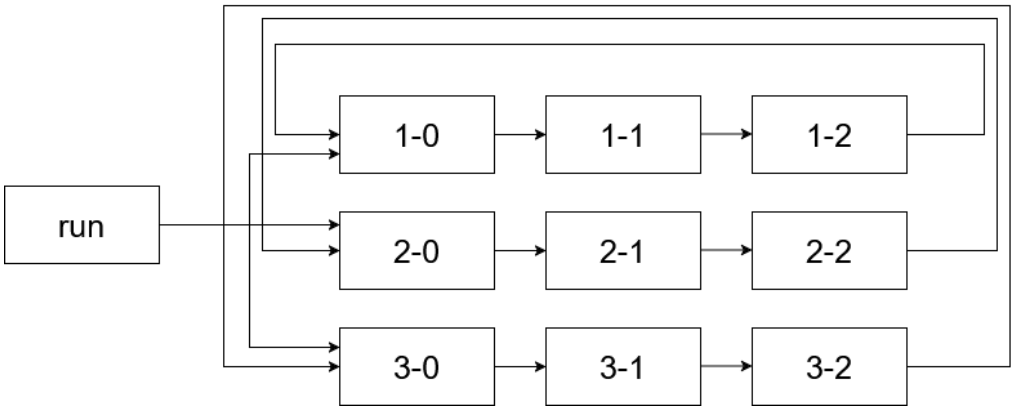

Abbildung 1: Beispielstrecke Aufgabe 1

2 Aufgabe 2 Cluster

In der Aufgabe 1 lief alles auf einer Adresse „localhost:29092“. Nun ersetzen wir das mit einem cluster aus 3 Addressen „localhost:29092, localhost:39092, localhost:49092“. Dies ist aus dem Github-Repo [1].

3 Aufgabe 3 „Ave Caesar“

3.1 Zugbasierte-Abfolge:

In Ave-Caesar wechseln sich die Spieler nacheinander ab. Dies wollen wir nun auch hier einführen. Dafür haben wir die Nachricht „turn_updates. Das verschickt die Nachricht an alle Segmente. Mit dem „event:turn-change“verstellen wir mit turnplayer die turnplayer_id in jedem segment. Jedes Segment, das ein Spieler-Token nun bekommt, wartet in einer Schleife darauf, dass die Spieler_id turnplayer entspricht. Nachdem das Segment mit dem Spieler-Token, das turn-player ist, fertig ist, verschickt es das „event:turnchange“an alle Segmente und wartet dann $0,5 \times Segmentanzahl$ lang, um zu garantieren, dass jedes Segment den turn_player umgeschaltet hat. Dann wird das Spieler-Token weiter gesandt. Es beginnt der Spieler auf Track 1 und die Reihenfolge läuft inkrementierend.

3.2 Belegtheit:

Im Spiel kann jedes Segment nur eine Spielfigur fassen und blockiert damit auch Wege für andere Spieler.
Dafür hat jedes Segment das Attribut „occupied“und „occupied_by“, um sicher zu stellen, dass ein belegtes Segment kein Spieler-Token bekommt.
Vor dem Weiterschicken eines Spieler-Tokens müssen wir scouten, ob der Weg frei ist. Sobald ein Spieler-Token auf einem Segment ankommt, wird dieses blockiert. Die Blockierung wird aufgehoben, wenn das Token dieses verlässt.

3.3 Scouten:

Zum Scouten, ob der Weg frei ist, schicken wir das „event:scout“nacheinander an die Segmente in der next_segments-Liste und warten in einer Schleife auf eine Antwort vom Segment mit „event:availability“. Dieses stellt die nötigen Variablen ein, sowie markiert es, dass eine Antwort kam. Dabei wird bei einer positiven Antwort das Segment schon mal als besetzt markiert. Ist ein Segment von keinem Spieler-Token besetzt, so ist die Antwort positiv.

3.4 Mehrere Schritte/Karten:

Jedes Spieler-Token hat wie im Spiel 3 Karten dabei, die anfangs zufällig zwischen 1 und 6 gesetzt werden.
Damit die Karten richtig funktionieren, wird das Scouting rekursiv aufgerufen und falls ein Segment offen ist, wird dieses vom Scouting mit „next_free_segment_id“zurückgeschickt.
Wird mit allen 3 Karten keines gefunden, so muss der Zug übersprungen werden, indem das Segment weiterhin blockiert bleibt und eine Nachricht vom Segment an sich selbst mit den selben Spieler geschickt wird, nachdem der turnplayer geändert wurde, damit der Spieler wieder auf seinen nächsten Zug wartet. Sobald eine Karte verbraucht wurde, wird diese durch „used_card_index“ersetzt. Mit „round_cnt“wird durch die gescoutete Route die Anzahl an Rundenerhöhung gezählt und die Rundenanzahl im Spieler-Token erhöht. Das Spieler-Token wird dann auf direktem Wege an „next_free_segment_id“gesandt. Mit „next_segment“schauen wir nur, ob die nächsten verfügbar waren.

3.5 Runden:

Die Runden speichert jedes Spieler-Token für sich selbst. Nach dem Vorbeilauf an einem „start-goal“oder „caesar“Segment wird diese um „round_cnt“erhöht. Sobald die fertige Rundenanzahl 3 erreicht wurde, wird die ID des Spieler-Tokens in „finished_players“in jedem Segment mit „turn_updates event:race_finished“geschickt und gewartet für jedes Segment. Dann wird ausgegeben wie viele Punkte der Spieler bekommen hat, was abhängig davon ist, ob caesar besucht wurde.

3.6 Caesar/Punktevergabe:

Im Spiel ist man im Zugzwang und muss daher jeden Zug einer seiner Karten ausspielen, aber auf der Kaisergasse muss man entweder in der ersten oder der zweiten Runde halten und den Kaiser grüßen. Im Programm bekommt man deshalb nur Punkte, wenn man in der Kaisergasse gehalten hat. Jedoch bekommt man |Segmente| viele Punkte für Platz 1 und mit jedem weiteren Platz ein Punkt weniger. Der Nicht-Besuch von Caesar führt dazu, dass man diese Punkte nicht bekommt. Jedoch bekommt auch kein anderer diese Punktzahl. Ist ein Spieler-Token fertig so wird seine Punktzahl einmal angezeigt und danach seine Züge übersprungen. Zusätzlich wird das Segment als nicht belegt markiert, damit andere Spieler-Token es überspringen können.

3.7 Strategie

Im Programm verfolgt jedes Spieler-Token die selbe Strategie. Es wird Greedy gespielt, indem jedes Spieler-Token die Karte mit der höchsten Zahl spielen will, um am weitesten zu kommen. Wenn dies nicht klappt, dann für die zweithöchste und sonst für die kleinste. Passt keine Karte, wird der Zug übersprungen. Die Besonderheit liegt darin, dass die Strategie sich beim Finden des Caesar-Segments ändert. Wird dieses gefunden, so wird es fokussiert und geschaut, ob dieses belegt ist und ob es mit den eigenen Karten erreichbar ist, um darauf zu halten. Ist das möglich, geht das Spieler-Token direkt dahin und verwendet die entsprechende Karte, aber nur wenn es die erste oder zweite Runde ist. Da das Caesar-Segment immer als segment-0-0 generiert wird, nehmen wir den ersten gefundenen Weg mit der höchsten spielbaren Karte, damit die Spieler-Token eher auf dem Caesar-Segment landen können.

3.8 Aufbau/Start

Die Strecke muss von circular-course.py generiert worden sein. Beim Start von run.py wird die Strecke eingelesen, das caesar_segment gesucht und alle Segmente gestartet. Schließlich bekommen die „start-goal“-Segmente ein Spieler-Token mit „event:start“geschickt. Dieses belegt das betroffene Segment und schickt das Spieler-Token an das selbe Segment mit „event:run“. Die Spieler-Token werden leicht versetzt geschickt. Zum Starten starte erst den Docker-Container mit

docker-compose up/docker compose up

je nach Docker Version.
Um das Programm zu starten, nutze entweder

python run.py parcours.json

oder

python run.py parcours.json logging

um mehr geloggt zu bekommen.

3.9 Limitierungen:

Nicht alles vom Spiel wurde implementiert. Dass der vorderste Spieler keine 6 spielen darf, wurde nicht implementiert, da man dafür sich den ersten Spieler merken und vergleichen müsste. Es gibt nur eine Rennstrecke statt 2 wie im Spiel. Aber wir können diese beliebig groß machen. Daher können auch mehr als 6 Gespannführer teilnehmen. Das Kartendeck ist auch unendlich groß. Eine weitere Limitierung ist, dass jeder Spieler die selbe „Greedy"-Strategie verfolgt und es somit zufällig ist, wer gewinnt. Dazu zählen die fertigen 3 Runden für jeden Spieler erst nach der Folgerunde, da dies am Anfang von „event:run"abgefragt wird. Eine Alternative habe ich probiert, aber nicht hinbekommen. Derzeit ist es so, als würde man in einem Zug über die Ziellinie fahren und erst im nächsten die Spielfigur entfernen. Dies schadet etwas der Ähnlichkeit zum Spiel. Auch ist eine Limitierung in meiner Behandlung vom Scouting bei sehr kleinen Strecken, wenn ein Streckenstück beim Scouting doppelt durchlaufen wird, ist die rekursive Antwort der Segmente fehlerhaft, da ein Segment auf zwei Antworten wartet, aber eine Nachricht für beide akzeptiert und dann der nächste Spieler dran ist. Da dies dazu führte, dass die Rundenanzahl des Scouting nicht richtig summiert wurde, habe ich die Regel hinzugefügt, dass jeder Track in der Rennstrecke minimal 6 Streckensegmente lang sein muss. Dadurch muss „round_cnt"nicht mehr summiert werden. Und für die Durchsetzung der Regel ist die Generierung zuständig. Beim Starten von run.py klappt das Programm aus mir nicht erklärlichen Gründen nicht und muss geschlossen und nochmals gestartet werden.

3.10 Strecken-Generator:

Die Strecke wird zufällig generiert. Zuerst wird ein zweidimensionales Array mit den Optionen „bottleneck", „wall-down"und „normal"generiert. Dabei füllt bottleneck entweder die ganze Spalte oder kommt in der Spalte gar nicht vor und wall-down kann nicht an unterster Stelle in der Spalte vorkommen. Daraus werden dann die next_segments aller Segmente generiert.

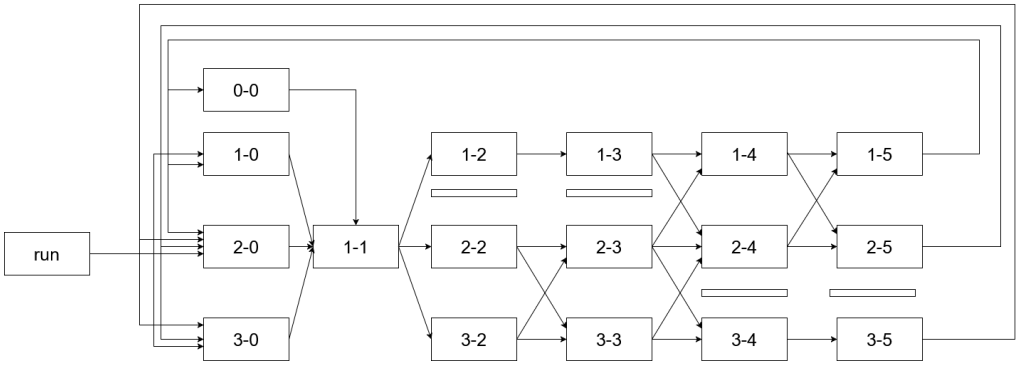

Abbildung 2: Beispielstrecke Aufgabe 3

Literatur

[1] `https://github.com/apache/kafka/blob/trunk/docker/examples/docker-compose-files/cluster/combined/plaintext/docker-compose.yml`

[2] `https://github.com/phgeie/SA4E_Ave_Caesar`