

Analytische und empirische Evaluation der Performance und Deadlock-Freiheit auf Basis des „Factorio Zugmodells“

Studienarbeit

im Studiengang Angewandte Informatik
an der Dualen Hochschule Baden-Württemberg Mannheim
von

Name, Vorname: Laurin

Name, Vorname: Hektor

Abgabedatum: 15.04.2023

Bearbeitungszeitraum: 17.10.2023 – 16.04.2024

Betreuer: Prof. Dr. Johannes Bauer

Erklärung

Wir versichern hiermit, dass wir unsere Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

1. Einleitung.....	1
2. Theoretische Grundlagen	2
2.1. Factorios Spielprinzip.....	2
2.2. Das Factorio Zugsystem	3
2.3. Deadlocks.....	7
2.3.1. Voraussetzungen und Vermeiden von Deadlocks.....	8
2.3.2. Erkennung von Deadlocks	10
2.3.3. Beheben von Deadlocks.....	12
3. Zugkreuzungen: Design-Herausforderungen, manuelle Tests und Entwurf eines automatischen Deadlock- und Performanceanalyse-Tools	13
3.1. Deadlocks im Factorio Zugsystem.....	13
3.2. Performance einer Zugkreuzung	14
3.3. Bewertung von Zugkreuzungen	16
3.4. Entwurf einer Anwendung zur Analyse der Deadlock-Freiheit und Performance von Zugkreuzungen	17
5. Implementierung eines Tools zur automatischen Analyse einer Factorio Zugkreuzung bezüglich der Deadlock-Freiheit und Performance.....	19
5.1. Aufbau der Java Anwendung	19
5.2. Das Zugstrecken-Modell	21
5.3. Erzeugung des Zugstrecken-Modells	24
5.3.1. Dekodierung der Blaupausen-Zeichenkette.....	24
5.3.2. Matrix-Generierung	24
5.3.3. Graph-Generierung	26
5.4. Deadlock-Erkennung.....	46
5.4.1. Rekursions-Algorithmus	47
5.4.2. Rekursionsdurchlauf anhand eines Beispiels	51

5.5. Performanceanalyse	56
5.6. User-interface	59
5.6.1. JavaFX-UI	59
5.6.2. Command Line Interface (CLI).....	62
6. Fazit	64
7. Literaturverzeichnis	65
8. Abbildungsverzeichnis	67

1. Einleitung

Die technischen Systeme der modernen Zeit werden immer komplexer. Es wird immer weiter optimiert und Prozesse werden immer effizienter bearbeitet. Doch dieser technische Fortschritt geht nicht immer reibungslos voran. Viele bekannte Softwareprodukte haben verschiedenste Probleme, die seit Jahren der Entwicklung nicht verhindert werden können. Ein bekanntes dieser Probleme sind Deadlocks. Sei es die Ressourcenverwaltung bei Betriebssystemen wie Windows oder bei der Umsetzung von Verhaltenskonzepten einer Anwendung, es bleibt ein immerzu aktuelles Thema, das beachtet werden muss. Das Ziel dieser Arbeit ist es, einen Überblick über die Erkennung von Deadlocks und die Messung von Performance in einem bestehenden System zu bieten. Dafür werden verschiedene Ansätze vorgestellt und die Entwicklung einer Umsetzung genauer beschrieben. Als Grundlage wird das Zugsystem in dem Spiel Factorio verwendet. Zunächst werden wichtige Begriffe und Konzepte, wie das Factorio Zugsystem und Deadlocks erklärt, um den richtigen Kontext zu schaffen. Im Anschluss wird die Umsetzung einer der vorher erläuterten Ansätze als ausführbare Applikation beschrieben, die Deadlocks in einem vom Nutzer übergegebenen Zugnetz erkennt und diesem eine Rückmeldung gibt. Außerdem wird ein Ansatz erläutert, wie die Performance dieses Zugnetzes analysiert werden kann. Zum Schluss folgt ein Fazit, inwiefern die Deadlock-Erkennung relevant ist und wie die Implementierungsphase ablief. Zunächst werden einige theoretische Grundlagen und Prinzipien erläutert, die in der fortlaufenden Arbeit für einen Lösungsansatz genutzt werden.

2. Theoretische Grundlagen

In diesem Kapitel werden Konzepte erläutert, die in dieser Arbeit vermehrt verwendet werden. Es folgt eine allgemeine Erklärung des Videospiels Factorio, dessen Zugsystem und eine Erläuterung von Deadlocks.

2.1. Factorios Spielprinzip

Das Videospiel Factorio wird von dem Unternehmen Wube Software LTD entwickelt und veröffentlicht. Es kombiniert die Elemente aus Strategie, Logistik und Automation. Der Charakter des Spielers stürzt zu Beginn mit seinem Raumschiff auf einem fremden Planeten ab. Die Hauptaufgabe des Spielers ist es Fabriken aufzubauen, um die Herstellung von verschiedenen Ressourcen zu automatisieren. Mit diesen können weitere Ressourcen, Maschinen oder Gegenstände freigeschaltet und ebenfalls automatisiert werden. Für die Freischaltung ganz neuer Ressourcen und Technologien ist es notwendig zu forschen. Dies geschieht mit Hilfe eines Laborgebäudes, welches ebenfalls automatisiert mit den benötigten Ressourcen versorgt werden kann. Das Endziel ist es mit Hilfe seiner selbst erbauten Industrie eine neue Rakete zu bauen und den Planeten wieder zu verlassen.

Im normalen Spielmodus gibt es auch gegnerische Aliens. Diese sind in bestimmten auf der Karte markierten Bereichen und werden durch die starke Luftverschmutzung durch die Fabriken aggressiv und fangen an die Gebäude des Spielers und den Spieler selbst anzugreifen. Deswegen ist es notwendig auch Verteidigungsanlagen, um seine Produktionsanlagen zu bauen oder die Aliens präventiv anzugreifen.

Da der Fokus aber auf der Automatisierung und Optimierung der Fabriken liegt, gibt es auch die Möglichkeit diese auszuschalten und sich lediglich auf die Planung der Fabriken zu konzentrieren. Hier liegt der Reiz des Spiels, da es zahllose Möglichkeiten gibt, seine Produktionsketten aufeinander perfekt abzustimmen, sodass keine Lücken in den Lieferketten entstehen.

Das Spiel setzt dabei auf eine stilistische 2D-Grafik mit einer endlosen Sandbox-Welt, die beliebig zugebaut und verändert werden kann. Die Bereitstellung von Mod-Support erlaubt es Spielern außerdem selbst Inhalte für das Spiel zu erstellen und zu veröffentlichen, weshalb die Spielmöglichkeiten deutlich über das Basisspiel erweitert werden kann. Der Multiplayer erlaubt es zusätzlich das Spiel mit seinen Freunden zusammen zu spielen. Obwohl das Spiel schon seit vielen Jahren auf dem Markt ist, erfreut es sich immer noch großer Beliebtheit. Die

durchschnittliche Spielerzahl pro Tag liegt im Jahr 2024 bei 14 Tausend gleichzeitigen Spielern.¹ Auf der Spieleverkaufsplattform Steam hat es über 142 Tausend Bewertungen wovon 97% positiv sind.² Deswegen wird auch heute noch an dem Spiel offiziell weiterentwickelt und neuer Content geliefert.³

2.2. Das Factorio Zugsystem

Die Fabriken in Factorio wachsen im Verlauf des Spieles stetig an und es werden daher mehr Ressourcen benötigt. Aufgrund dessen müssen Ressourcenquellen, die weiter entfernt sind, erschlossen werden. Innerhalb einer Fabrik werden Ressourcen und Produkte mit Transportbändern transportiert. Diese Transportbänder können im Verlaufe des Spiels mithilfe von Forschung hinsichtlich der Geschwindigkeit verbessert werden.⁴ Allerdings ist der Einsatz von Transportbändern ab einem gewissen Abstand zwischen den Ressourcen und der Fabrik nicht mehr kosteneffektiv, da die schnellsten Transportbänder teuer in der Herstellung sind, das Platzieren dieser Zeit kostet und diese nicht leicht veränderbar sind.

Um dieses Problem zu beheben, werden Züge für den Transport von Ressourcen oder Produkten verwendet. Züge werden über die Forschung freigeschalten und verbessert. Bei den Verbesserungen kann unterschiedlicher Brennstoff freigeschalten werden, der den Zug schneller fahren lässt, oder bessere Bremsen erforscht werden, um den Zug länger auf der Maximalgeschwindigkeit fahren zu lassen. Ein Zug besteht aus mindestens einer Lok und mehreren Wagons, welche Ressourcen transportieren können.

Um ein Schienennetz aufzubauen, platziert der Spieler Schienen zwischen den beiden Punkten, die er verbinden möchte. Der Spieler muss jeweils eine Auflade-Station beziehungsweise eine Ablade-Station bauen. Diese Stationen markiert er, in dem er eine Haltestelle platziert. In Abbildung 1 ist eine Ablade-Station abgebildet. Die Züge können automatisiert zwischen verschiedenen Haltestellen fahren. Hierbei wird ein Fahrplan in der Lok des jeweiligen Zuges festgelegt, der bestimmt, zu welchen Stationen der Zug wann beziehungsweise unter welchen Bedingungen fährt. Beispielsweise sollte bei einer Ablade-Station der Zug erst losfahren, wenn seine gesamte Fracht abgeladen worden ist.⁵

¹Vgl. „Factorio - Steam Charts“, zugegriffen 15. Februar 2024, <https://steamcharts.com/app/427520#7d>.

²Vgl. „Factorio on Steam“, zugegriffen 12. Februar 2024, <https://store.steampowered.com/app/427520/Factorio/>.

³Vgl. „Factorio“, Factorio, zugegriffen 12. Februar 2024, <https://www.factorio.com/>.

⁴Vgl. „Fließband-Transportsystem - Factorio Wiki“, zugegriffen 21. Februar 2024, https://wiki.factorio.com/Belt_transport_system/de.

⁵Vgl. „Eisenbahn - Factorio Wiki“, zugegriffen 21. Februar 2024, <https://wiki.factorio.com/Railway/de>.



Abbildung 1: Ablade-Station in Factorio

Da in einem Schienennetz mehrere Züge fahren können, müssen diese mithilfe eines Signal-Systems aufeinander achten. Wenn mehrere Züge in einem Netz ohne Signal fahren, werden diese Unfälle verursachen und sich gegenseitig zerstören. Factorio ermöglicht es dem Spieler mithilfe von Zugsignalen ein Schienennetz in Streckenabschnitte aufzuteilen und die Fahrtrichtung festzulegen. In Abbildung 2 sind Streckenabschnitte, die durch Signale definiert werden, dargestellt.



Abbildung 2: Streckenabschnitte in Factorio

Das Signal definiert auch die zugelassenen Fahrtrichtungen. Das grüne Licht eines Signals gibt die Seite des Signals an, von der Züge einfahren dürfen. Die andere Seite des Signals ist der Abschnitt, der durch dieses Signal überwacht wird. Beispielsweise gibt das linke horizontal ausgerichtete Signal in Abbildung 2 eine Fahrtrichtung von links nach rechts beziehungsweise

vom blauen Abschnitt in den pinken Abschnitt an. Wenn in beide Fahrtrichtungen gefahren werden soll, müssen die Signale auf beiden Seiten der Schiene platziert werden.

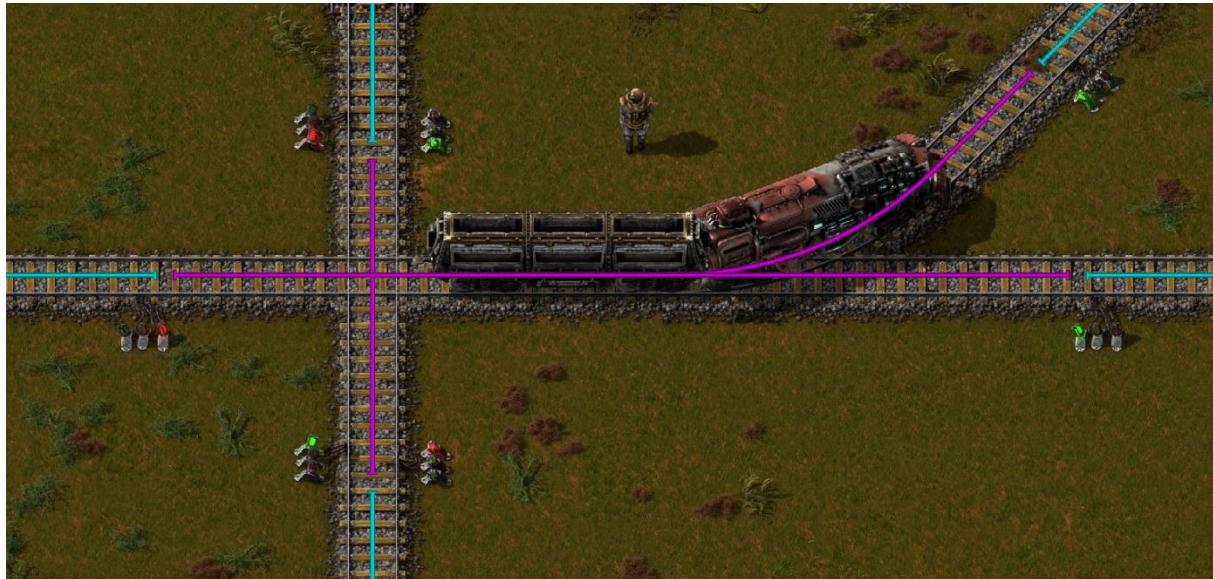


Abbildung 3: Belegter Streckenabschnitt

Ein Zugsignal gibt an, ob der sich dahinter befindende Streckenabschnitt bereits belegt, frei oder reserviert ist. In einem Streckenabschnitt darf sich nur ein Zug befinden. Befindet sich ein Zug wie beispielsweise in Abbildung 3 im pink markierten Abschnitt, markiert das Signal dies mit einem Aufleuchten der roten Lampe. Wenn ein Zug ein rotes Signal vor einem Streckenabschnitt, in den er hineinfahren will, begegnet, hält der Zug an und wartet, dass der Streckenabschnitt wieder frei. Ein gelbes Signal gibt an, dass sich von dort ein Zug nähert und der durch das Signal überwachte Abschnitt für diesen reserviert worden ist. Alle anderen Signale schalten für diesen Abschnitt aufgrund dessen auf Rot. Eine Reservierung wird vorgenommen, wenn der Zug aufgrund seines Bremsweges nicht rechtzeitig abbremsen kann, wenn das Signal auf Rot schalten würde.⁶

Neben den Zugsignalen gibt es auch die Zug-Kettensignale. Ein Zug-Kettensignal gibt an, ob der nächste und die übernächsten Abschnitte frei sind. Um zu erkennen, ob die übernächsten Abschnitte frei sind, wertet das Kettensignal den Status der Signale, die die übernächsten Abschnitte überwachen, aus. In Abbildung 4 ist ein Kettensignal dargestellt. Wenn ein Kettensignal grün leuchtet, dann sind alle übernächsten Abschnitte frei. Leuchtet es allerdings blau, dann ist mindestens einer der übernächsten Abschnitte gesperrt. Falls alle übernächsten

⁶Vgl. „Zugsignal - Factorio Wiki“, Official Factorio Wiki, zugegriffen 21. Februar 2024, https://wiki.factorio.com/Rail_signal/de.

Abschnitte gesperrt sind oder der nächste Abschnitt gesperrt ist, dann leuchtet das Kettensignal rot. Kettensignale können miteinander verkettet werden, wie in Abbildung 5 dargestellt.

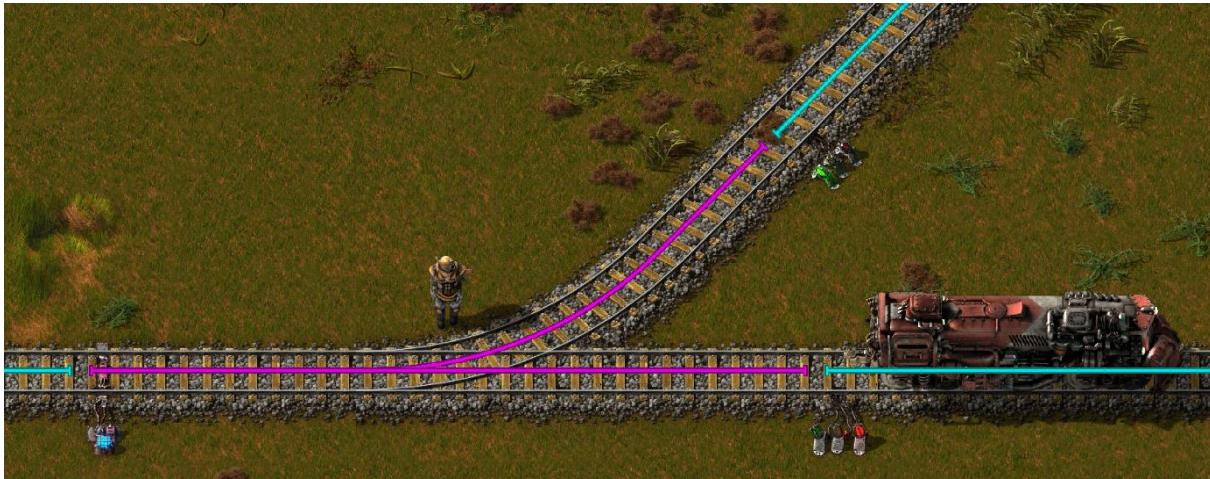


Abbildung 4: Ein Zug-Kettensignal bei einem versperrten Ausgang



Abbildung 5: Miteinander verkettete Kettensignale

Mithilfe von Kettensignalen wird bewirkt, dass Züge nur in einen unsicheren Abschnitt fahren können, wenn der Abschnitt danach auch frei ist. Damit wird verhindert, dass ein Zug in einem unsicheren Abschnitt wartet. Ein Beispiel für einen unsicheren Abschnitt ist eine Kreuzung, die in Abbildung 6 abgebildet ist. Hier wird verhindert, dass ein Zug in der gelb markierten Kreuzung wartet. Ohne ein Kettensignal würde der linke Zug in der Kreuzung warten und daher Züge, die durch diese Kreuzung von oben nach unten durchfahren, blockieren.⁷

⁷Vgl. „Zug-Kettensignal - Factorio Wiki“, Official Factorio Wiki, zugegriffen 21. Februar 2024, https://wiki.factorio.com/Rail_chain_signal/de.

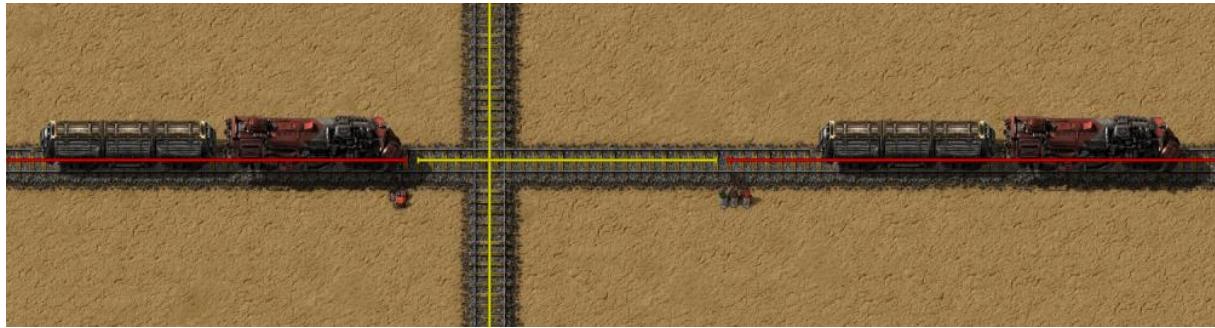


Abbildung 6: Ein unsicherer Kreuzungsabschnitt

Da in Factorio das Schienennetz aufgrund des steigenden Ressourcenbedarfs anwächst und mehr Züge fahren müssen, werden Kreuzungen mit mehreren Ein-/Ausgängen benötigt. Beim Design solcher Kreuzungen sind verschiedene Parameter wie beispielsweise die Performance oder die Deadlock-Freiheit wichtig zu beachten. Im nächsten Kapitel wird im Allgemeinen erklärt, was Deadlocks sind.

2.3. Deadlocks

Ein Deadlock ist ein Zustand, welcher einen zyklischen Wartezustand beschreibt, der ohne Eingriff nicht enden kann. Dies ist ein wichtiges Thema bei Betriebssystemen oder Datenbanken und entsteht, wenn jeder Prozess aus einer Menge von Prozessen, eine Ressource anfordert, die von einem anderen Prozess der Menge blockiert wird. So wartet jeder Prozess auf einen anderen und keiner ist in der Lage weiterzuarbeiten.⁸

Ein anschaulicheres Beispiel für einen Deadlock ist das „Dining-Philosopher“ - Problem von Dijkstra. Es beschreibt eine Szene, wobei fünf Philosophen an einem Tisch sitzen, um zu essen. Es gibt also fünf Teller, die auf einem runden Tisch in einem Kreis angeordnet sind. Zwischen den Tellern liegt jeweils eine Gabel. Damit gegessen werden kann braucht ein Philosoph eine Gabel in der linken und in der rechten Hand. Es gilt dabei, dass erst die linke Gabel aufgehoben werden muss, vor der rechten. So ergibt sich ein Deadlock, da die rechte Gabel immer die linke eines anderen Philosophen ist.⁹

⁸Vgl. „betriebssysteme-11-23-05-25-Wdh.pdf“, zugegriffen 11. Februar 2024, https://moodle.dhbw-mannheim.de/pluginfile.php/535400/mod_resource/content/1/betriebssysteme-11-23-05-25-Wdh.pdf.

⁹Vgl. „Referat ‚Deadlocks‘ im Proseminar ‚Computer Science Unplugged‘“, o. J.

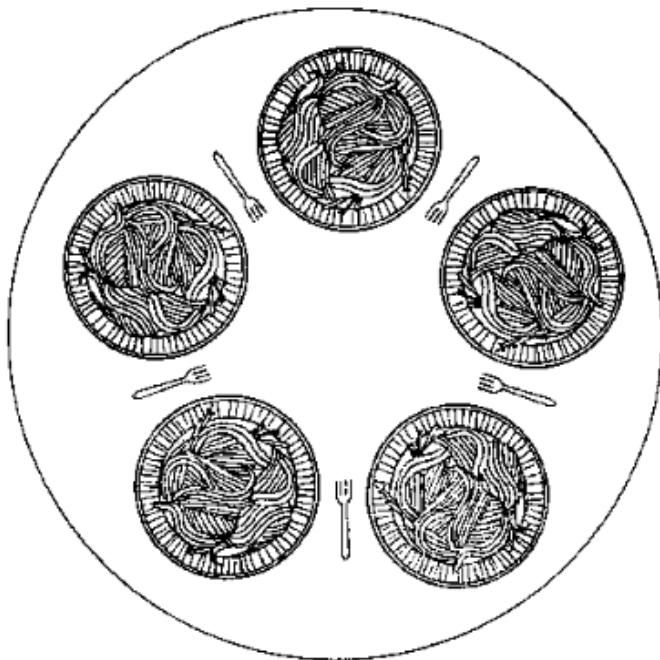


Abbildung 7: Das „Dining-Philosopher“ Problem

Einen ähnlichen Begriff zum Deadlock gilt es zu unterscheiden. Das Verhungern oder auch Starvation beschreibt einen ähnlichen, aber dennoch unterschiedlichen Zustand. Während Deadlocks ein zyklisches Warten beschreiben, wartet ein Prozess beim Verhungern unendlich lange. Das ist zum Beispiel der Fall, wenn in einem Betriebssystem ein Prozess mit höherer Priorität einem Prozess mit niedriger Priorität vorgezogen wird. Wenn während der Bearbeitung dieses Prozesses, neue Prozesse mit hoher Priorität entstehen, werden diese dem Prozess mit niedrigerer Priorität wieder vorgezogen. Das kann theoretisch unendlich lange fortgeführt werden und dieser eine Prozess mit niedriger Priorität wartet unendlich lange, obwohl hier im Gegensatz zum Deadlock auf keine spezifische Ressource eines anderen Prozesses gewartet wird.¹⁰

2.3.1. Voraussetzungen und Vermeiden von Deadlocks

Damit ein Deadlock auftreten kann, müssen vier Bedingungen gleichzeitig gelten. Die Erste ist der gegenseitige Ausschluss der Ressourcen. Das bedeutet, wenn ein Prozess eine Ressource angefordert hat, weil diese für den Abschluss benötigt wird, kann währenddessen kein anderer

¹⁰Vgl. „Deadlock v/s Starvation - Coding Ninjas“, zugegriffen 11. Februar 2024, <https://www.codingninjas.com/studio/library/deadlock-vs-starvation>.

Prozess diese Ressource nutzen, obwohl diese hier ebenfalls für den Abschluss benötigt wird. Eine weitere Bedingung ist die „Hold-and-Wait-Bedingung“. Diese trifft zu, wenn Prozesse, die bereits Ressourcen reserviert haben, weitere Ressourcen anfordern können. Die dritte Bedingung ist die Ununterbrechbarkeit von Prozessen. Das bedeutet, dass eine Ressource nicht gewaltsam von einem Prozess genommen werden kann, sei es durch einen anderen Prozess oder einer übergeordneten Instanz. Die Ressource muss explizit von den Prozessen, die diese nutzen, freigegeben werden. Die vierte Bedingung ist eine zyklische Wartebedingung. Das trifft auf eine zyklische Kette von Prozessen zu, wobei jeder dieser Prozesse auf eine Ressource wartet, die einem anderem Prozess gehören. Alle Prozesse befinden sich hier in einem Wartezustand.¹¹

Um Deadlocks zu vermeiden, muss das System mindestens eine dieser Bedingungen vermeiden. Der gegenseitige Ausschluss ist in der Regel nicht zu verhindern. Sollten zwei oder mehr Prozessen gleichzeitig dieselbe Ressource nutzen, entsteht abhängig von der Nutzung ein Durcheinander und die Prozesse werden eventuell nicht beendet. Ein Beispiel hierbei ist, wenn ein Prozess die Ressource zu einem Zweck nutzt und ein zweiter Prozess soll diese Ressource löschen. Die „Hold-and-Wait-Bedingung“ kann ebenfalls meistens nicht verhindert werden. Es gibt verschiedene Ansätze diese Bedingung zu umgehen, beispielsweise damit, dass alle Prozesse alle benötigten Ressourcen reservieren, bevor sie beginnen. Sollten zwei Prozesse dieselbe Ressource anfordern, wird es erneut versucht. Hier kann aber eventuell keine Lösung erzielt werden. Manchmal kann ein Prozess aber auch nicht wissen, wie viele Ressourcen benötigt werden, weil sich das während dem Prozess ergibt. Diese Bedingung kann allerdings umgangen werden, indem die Ununterbrechbarkeit ebenfalls wegfällt. Sollte die dritte Bedingung nicht greifen, kann die „Hold-and-Wait-Bedingung“ umgangen werden, indem ein Prozess einem anderen die Ressource wegnehmen kann. Diese Vorgehensweise wäre eine Möglichkeit, um Deadlocks zu verhindern. Die zyklische Wartebedingung kann aber ebenfalls vermieden werden. Eine Möglichkeit wäre es, die Ressourcen aufsteigend zu nummerieren. Ein Prozess kann nur dann eine neue Ressource anfordern, wenn diese einen höheren Wert hat als die vorher von ihm belegte. Bei einem geringerem muss die Ressource

¹¹Vgl. „betriebssysteme-11-23-05-25-Wdh.pdf“, zugegriffen 11. Februar 2024, https://moodle.dhbw-mannheim.de/pluginfile.php/535400/mod_resource/content/1/betriebssysteme-11-23-05-25-Wdh.pdf.

mit dem höheren Wert erst freigegeben werden, bevor die neue mit geringerem Wert benutzt wird.¹²

2.3.2. Erkennung von Deadlocks

Es gibt verschiedene Möglichkeiten Deadlocks in einem System zu finden. Im Folgenden werden zwei Verfahren zur Deadlock-Erkennung erklärt.

Das erste Verfahren nutzt einen Wartegraphen. Dieser zeigt alle Wartebeziehungen zwischen den Prozessen. Zunächst wird ein Ressourcenzuweisungsgraph mit allen Ressourcen und Prozessen erstellt.

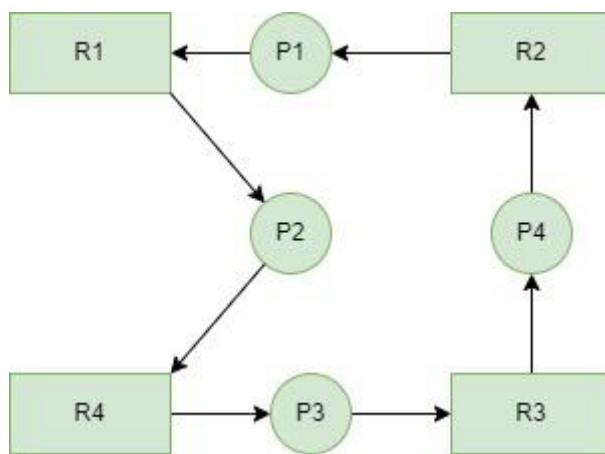


Abbildung 8: Ein Ressourcenzuweisungsgraph

Die eckigen Kästen stellen Ressourcen dar und die Kreise Prozesse. Wenn ein Pfeil von einer Ressource auf einen Prozess zeigt, bedeutet das, der Prozess nutzt diese Ressource. Ein Pfeil von einem Prozess auf eine Ressource zeigt an, dass der Prozess diese Ressource benötigt. Zum Beispiel zeigt hier ein Ausschnitt des Graphen, dass Ressource zwei von Prozess eins benutzt wird, dieser aber zusätzlich Ressource eins anfordert, welche aber von Prozess zwei genutzt wird. Dieser Graph kann nun vereinfacht werden, da im Endeffekt nur die Wartebeziehungen zwischen den Prozessen relevant sind. Zusammengefasst bedeutet das eben genannte Beispiel, dass Prozess eins darauf wartet, dass Prozess zwei fertig wird, damit die benötigte Ressource freigegeben wird und Prozess eins weiterarbeiten kann.

Genau dasselbe wird mit dem gesamten Graph getan, woraus sich folgender neuer Graph ergibt.

¹²Vgl. Subham Datta, „Deadlock: What It Is, How to Detect, Handle and Prevent? | Baeldung on Computer Science“, 20. Juni 2021, <https://www.baeldung.com/cs/os-deadlock>.

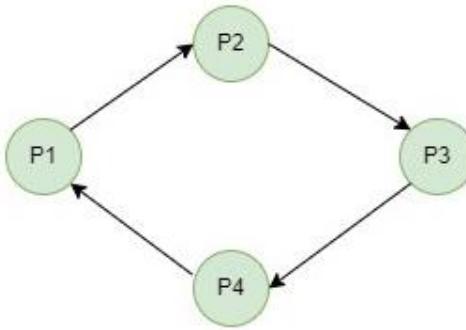


Abbildung 9: Ein Wartegraph

Dieser Graph zeigt nun alle Wartebedingungen zwischen den Prozessen aus dem ersten Ressourcenzuweisungsgraph. Das bedeutet in diesem Fall wartet Prozess eins auf Prozess zwei, welcher auf Prozess drei wartet, welcher wiederum auf Prozess vier wartet, der auf Prozess eins wartet. Es ergibt sich ein Kreis im Graphen und damit eine endlose Wartesituation, das System befindet sich also in einem Deadlock.¹³ ¹⁴

Eine andere Möglichkeit mit Deadlocks umzugehen, besteht darin einem Prozess verschiedene Timeouts mitzugeben. Die genaue Umsetzung kann hier verschieden angegangen werden. Als Beispiel kann jedem Prozess eine gewisse Zeit mitgegeben werden, nach dessen Ablauf überprüft werden soll, ob der Prozess sich in einem Deadlock befindet. Wenn nicht, kann entweder unendlich lange gewartet werden, bis die benötigte Ressource freigegeben wird oder es gibt einen weiteren Timer, der den Prozess nach einer weiteren Zeitspanne beendet.

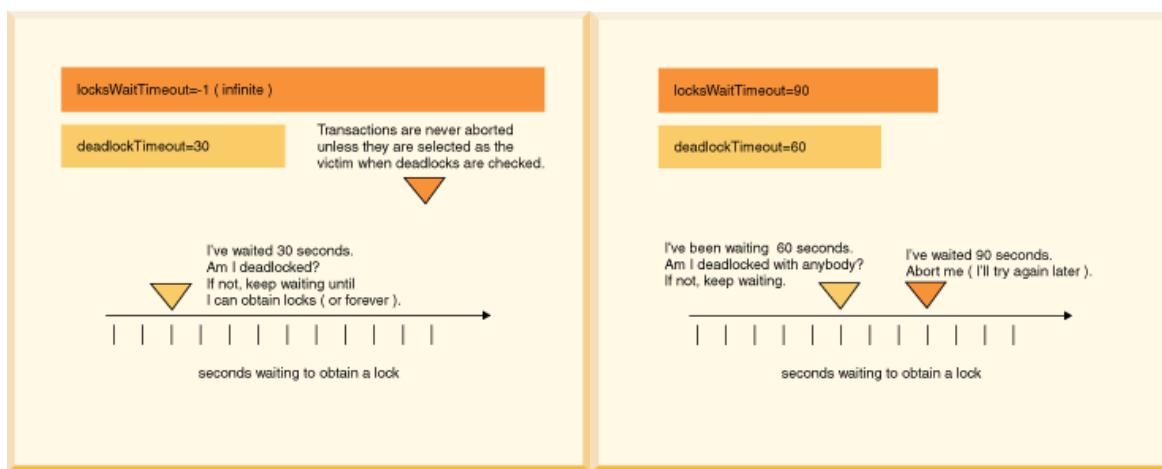


Abbildung 10: Timeout Ansatz zur Deadlock-Erkennung

¹³Vgl. „Resource Allocation Graph | Deadlock Detection | Gate Vidyalay“, zugegriffen 11. Februar 2024, <https://www.gatevidyalay.com/resource-allocation-graph-deadlock-detection/>.

¹⁴Vgl. „Wait For Graph Deadlock Detection in Distributed System“, GeeksforGeeks (blog), 25. April 2022, <https://www.geeksforgeeks.org/wait-for-graph-deadlock-detection-in-distributed-system/>.

Genauso kann auch der Timer für die Deadlock-Überprüfung weggelassen werden und der Prozess kann nach einer gewissen Zeitspanne beendet oder neu gestartet werden, wenn diese höher ist, als der Prozess benötigen sollte. Die Umsetzung ist von den System- und Anwendungsanforderungen abhängig, wie auch vom Prozess.¹⁵

2.3.3. Beheben von Deadlocks

Das Beheben von Deadlocks ist häufig schwierig umsetzbar. Wenn sich ein System in einem Deadlock-Zustand befindet, gibt es zwei grundsätzliche Vorgehensweisen.

Einerseits gibt es den Vogel-Strauß-Algorithmus. Dieser schreibt vor den Deadlock zu ignorieren. Die betroffenen Prozesse warten dadurch unendlich lange oder werden von außen, beispielsweise durch menschlichen Eingriff aus dem Zustand befreit. Normalerweise erscheint ein derartiges Vorgehen als ineffizient, sollte aber der Aufwand zur Verhinderung des Deadlock-Zustandes groß sein im Vergleich zur Wahrscheinlichkeit, dass dieser Zustand eintritt, wird diese Methodik als sinnvoll angesehen.^{16 17}

Eine weitere Möglichkeit Prozesse aus dem Deadlock-Zustand zu befreien, ist durch das Eingreifen des Systems selbst. Es kann beispielsweise eine Ressource einem Prozess gewaltsam entzogen werden oder das System setzt sich auf einen vorherigen funktionsfähigen Zustand zurück. Die genaue Implementierung ist hier von Art und Funktionsweise des Systems und den Prozessen abhängig. Ein derartiger Eingriff des Systems auf sich selbst hat allerdings starken Einfluss auf die Funktionsfähigkeit, weswegen grundsätzlich gilt, dass die Entstehung von Deadlocks durch die vorher genannten Bedingungen im Voraus vermieden werden sollten, sodass eine Möglichkeit zur Behebung irrelevant wird.¹⁸

¹⁵Vgl. „Configuring Deadlock Detection and Lock Wait Timeouts“, concept, zugegriffen 11. Februar 2024, <https://db.apache.org/derby/docs/10.9/devguide/cdevconcepts16400.html>.

¹⁶Vgl. „betriebssysteme-11-23-05-25-Wdh.pdf“, zugegriffen 11. Februar 2024, https://moodle.dhbw-mannheim.de/pluginfile.php/535400/mod_resource/content/1/betriebssysteme-11-23-05-25-Wdh.pdf.

¹⁷Vgl. „Vogel-Strauß-Algorithmus“, Academic dictionaries and encyclopedias, zugegriffen 11. Februar 2024, <https://de-academic.com/dic.nsf/dewiki/1471161>.

¹⁸Vgl. „Wait For Graph Deadlock Detection in Distributed System“.

3. Zugkreuzungen: Design-Herausforderungen, manuelle Tests und Entwurf eines automatischen Deadlock- und Performanceanalyse-Tools

In diesem Kapitel werden zuerst die grundlegenden Herausforderungen beim Design einer Zugkreuzung erklärt. Danach erfolgt ein Einblick, wie Spieler manuell die Deadlock-Freiheit und Performance einer Kreuzung testen können. Zum Schluss wird der Aufbau und die Funktionen einer Applikation beschrieben, die automatisiert eine Zugkreuzung bezüglich der Deadlock-Freiheit und Performance analysiert.

3.1. Deadlocks im Factorio Zugsystem

Die Züge in Factorio können mit den Prozessen eines Betriebssystems verglichen werden. Ein Prozess besitzt Ressourcen und kann weitere Ressourcen anfordern, während dagegen der Zug sich in einem Abschnitt befindet und in einen anderen hineinfahren möchte. Ein Deadlock tritt daher beispielsweise auf, wenn ein Zug sich in Abschnitt A befindet und in Abschnitt B einfahren möchte, der allerdings durch einen weiteren Zug bereits besetzt ist, welcher sich wiederum nach A begeben möchte. Dieser Sachverhalt ist in Abbildung 11 dargestellt.

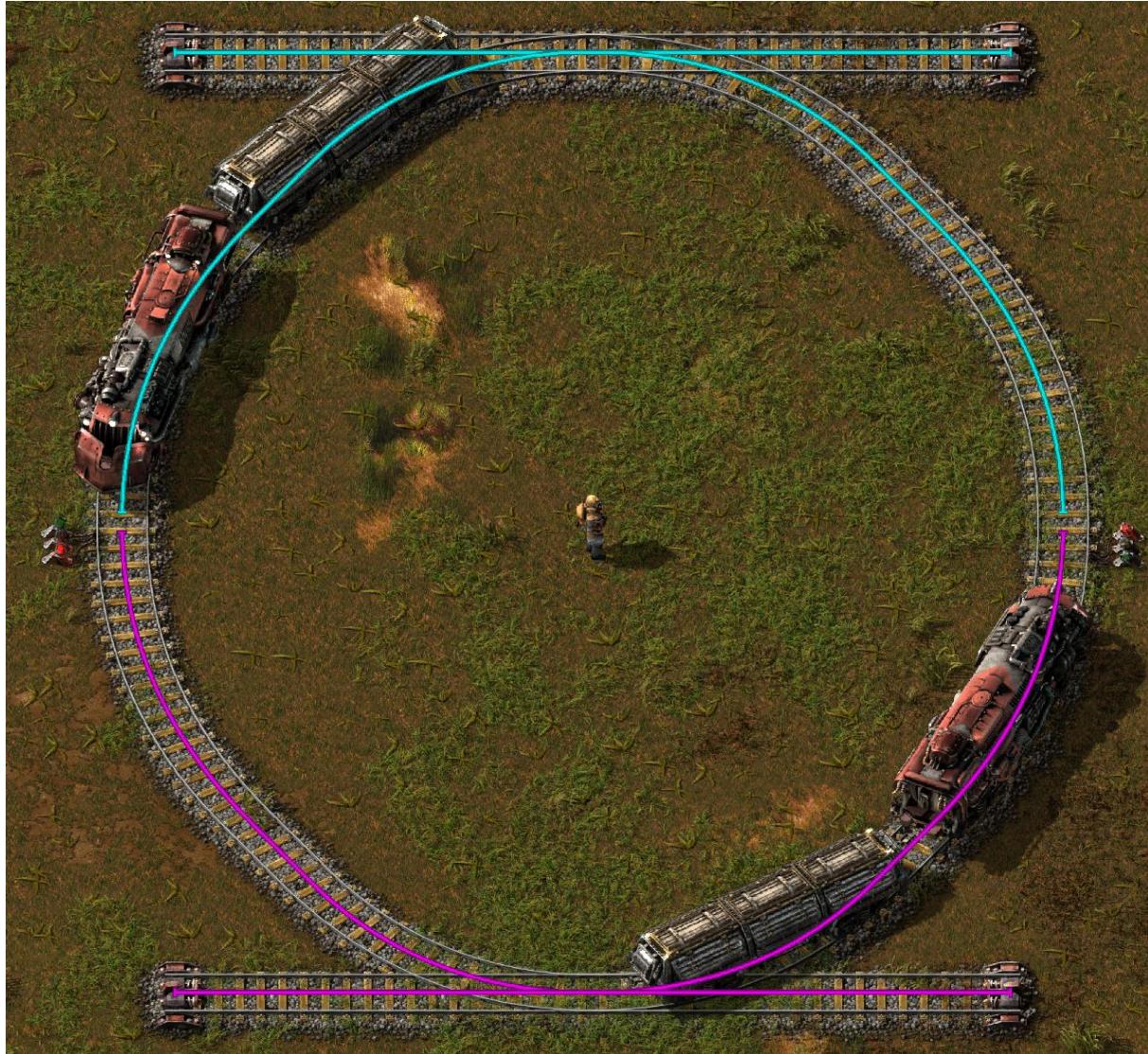


Abbildung 11: Deadlock im Factorio Zugsystem

Wie in einem vorhergehenden Kapitel angesprochen worden ist, gibt es zwei Möglichkeiten für die Applikation, um einen Deadlock zu erkennen. Es kann zum einen ein Wartegraph verwendet werden, um einen Deadlock zu erkennen, zum anderen können die simulierten Züge mit einem Timeout versehen werden. Wenn ein Zug sich nach Ablauf desTimeouts noch in der Kreuzung aufhält, kann analysiert werden, ob ein Deadlock vorliegt. Welcher Ansatz bei der Implementierung hierbei verwendet wird, wird in einem späteren Kapitel erläutert.

3.2. Performance einer Zugkreuzung

Die Performance einer Zugkreuzung wird mit dem Durchsatz an Zügen in einem konkreten Zeitabstand gemessen. Für einen fairen Vergleich ist es wichtig, dass die Züge eine feste Länge in der Messung des Durchsatzes besitzen. Eine Kreuzung, die es schafft, 45 Züge pro Minute durchzuleiten, ist performanter als eine, die nur 10 Züge pro Minute durchleitet.

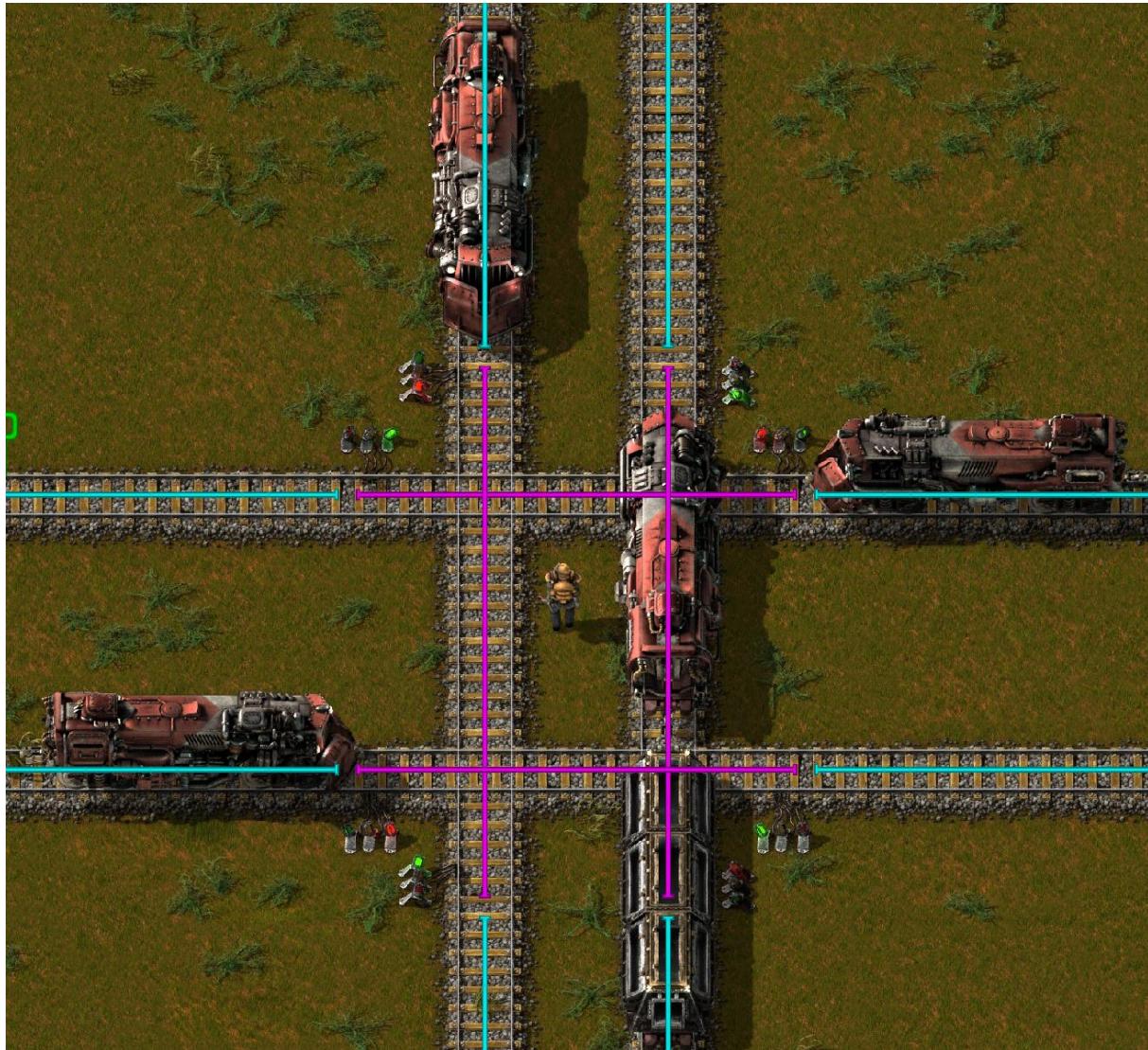


Abbildung 12: Deadlockfreie aber inperformante Zugkreuzung

In Abbildung 12 ist eine deadlockfreie Kreuzung abgebildet. Das Problem an dieser Kreuzung ist, dass diese aufgrund der gesamten Sperrung des pinken Abschnittes auch Züge blockiert, die sich auch sicher durch diese Section bewegen könnten. Beispielsweise könnte der obere linke Zug in Abbildung 12 sich weiterbewegen, ohne mit dem anderen Zug im pinken Abschnitt zu kollidieren. Um diesen Flaschenhals zu entfernen, wird der pinke Abschnitt wie in Abbildung 13 in 4 weitere aufgeteilt. Diese Verbesserung verdoppelt den theoretischen Durchsatz dieser Kreuzung, da statt 3 Züge nur 2 Züge im Idealfall blockiert werden.

Das Problem bei dieser Verbesserung ist, dass ein Deadlock auftreten kann, wenn in jedem pinken und gelben Abschnitt sich ein Zug befindet.

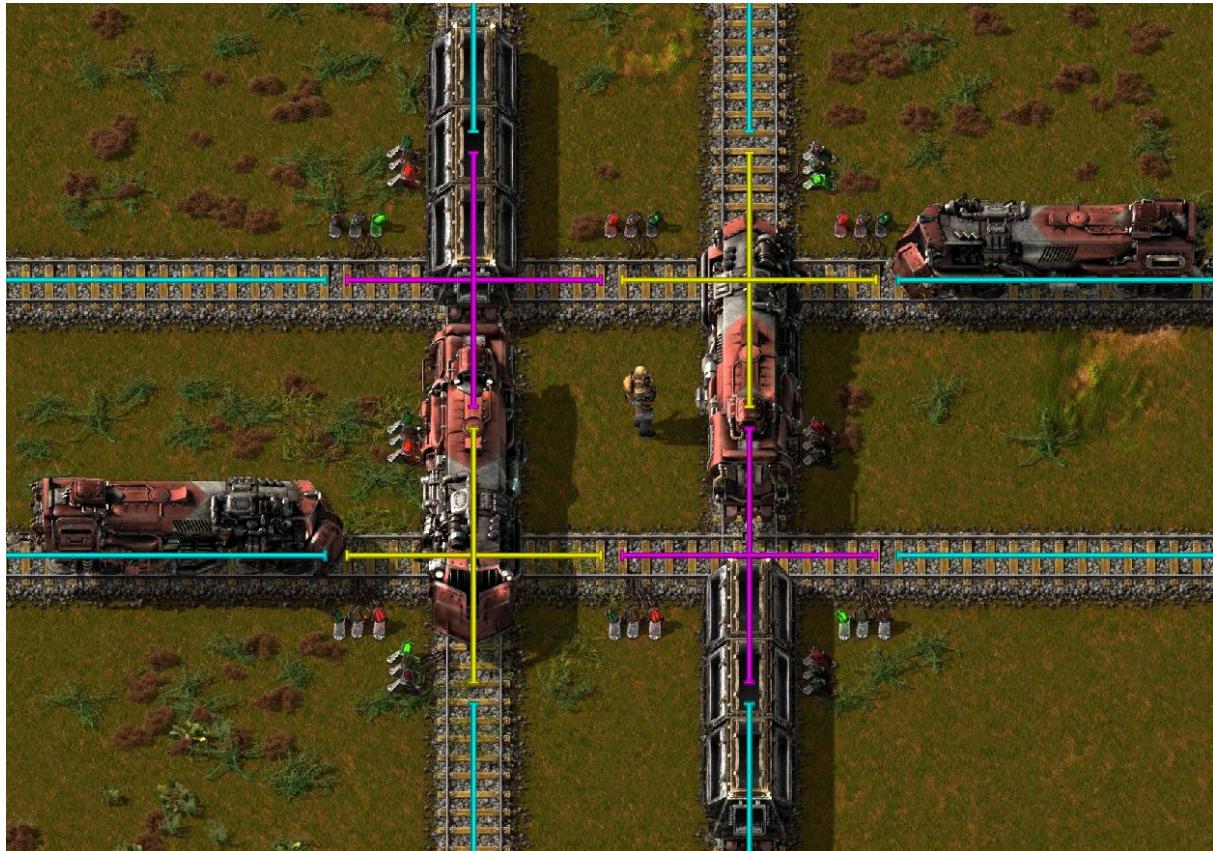


Abbildung 13: Performantere aber nicht deadlockfreie Kreuzung

Der Spieler muss daher bei Performanceverbesserungen von Zugkreuzungen darauf achten, dass diese weiterhin frei von Deadlocks bleiben. Im nachfolgenden Kapitel werden die Herausforderungen an den Spieler beim Kreuzungsdesign näher beleuchtet.

3.3. Bewertung von Zugkreuzungen

Wenn ein Spieler eine Kreuzung entwirft, muss er darauf achten, dass diese zum einen deadlockfrei und zum anderen performant ist. Das Spiel bietet dem Spieler keinerlei Funktionalitäten an, um eine Kreuzung bezüglich der Deadlock-Freiheit und Performance zu analysieren. Aufgrund dessen muss der Spieler selbst mögliche Szenarien für eine von ihm gebaute Kreuzung durchspielen und ermitteln, ob das Design ausreichend ist. Bei Kreuzungen mit einer geringeren Anzahl an Ein- und Ausgängen sind solche Probleme erkennbar. Wenn aber die Anzahl der Ein- und Ausgänge steigt und Züge von einem beliebigen Eingang in jeden beliebigen Ausgang fahren sollen, dann steigt die Komplexität dieser Aufgabe an. Der Spieler kann die Analyse seines Kreuzungsentwurf auch vernachlässigen. Dies hat zur Folge, dass beim Auftreten eines Deadlocks der Spieler diesen manuell beheben muss. Da in Factorio die Automatisierung das Ziel ist, ist dies keine zufriedenstellende Lösung.

Um daher die steigende Komplexität beim Designen von Kreuzungen dem Spieler gegenüber zu bewältigen, wird in dieser Arbeit eine Applikation entworfen, die die Analyse der entworfenen Kreuzung übernimmt und dem Spieler Feedback zu seinem Entwurf liefert. Im nachfolgenden Abschnitt wird der Entwurf dieser Applikation erläutert.

3.4. Entwurf einer Anwendung zur Analyse der Deadlock-Freiheit und Performance von Zugkreuzungen

Die erste Herausforderung für die zu entwerfende Applikation ist es Informationen über die Beschaffenheit der zu analysierenden Kreuzung zu erhalten. Hierbei bietet sich an, die Blaupausenmechanik des Spiels zu verwenden. Factorio ermöglicht das Teilen und das Wiederverwenden von bereits gebauten Industrieanlagen mittels Blaupausen. Eine Blaupause ist eine Zeichenkette, die für Factorio alle wichtigen Informationen bezüglich der Platzierung verschiedener Gebäude beinhaltet. Das Spiel bietet die Möglichkeit an eine Blaupause als String zu exportieren oder andere Blaupausen in das Spiel zu importieren.¹⁹ Von einer Zugkreuzung kann auch eine Blaupause erstellt werden. Aufgrund dessen muss die Anwendung den exportierten Blaupausen String einlesen und Informationen aus diesem erhalten können.

Für die Interaktion zwischen dem Nutzer mit der Anwendung gibt es verschiedene Optionen. Ein Ansatz ist es, dass der Nutzer mit der Anwendung über ein Command Line Interface (CLI) interagiert. Die Anwendung wird in einer Konsole gestartet und der Nutzer gibt beim Start den Dateipfad zu einer Textdatei mit dem Blaupausen-String ein. Nach der Eingabe einer validen Blaupause erzeugt die Applikation eine json-Datei mit den Ergebnissen.

Das Anzeigen einer grafischen Oberfläche ist eine weitere Option. Der Nutzer startet die Anwendung und gibt im erscheinenden Fenster seine Blaupause ein. Die Anwendung zeigt in der Oberfläche die eingelesene Blaupause an, damit der Nutzer verifizieren kann, ob die Blaupause erfolgreich eingelesen worden ist. Falls ein Deadlock auftreten kann, wird dieser über der angezeigten Blaupause eingebendet. Die Einblendung des Deadlock hilft dem Nutzer zu erkennen, an welcher Stelle der Kreuzung der Deadlock auftritt und wie er diesen beheben kann. Bei der Performanceanalyse werden die simulierten Züge angezeigt und nach erfolgreichem Abschluss der Simulation der gemessene Durchsatz angezeigt.

¹⁹Vgl. „Blaupause - Factorio Wiki“, Official Factorio Wiki, zugegriffen 21. Februar 2024, <https://wiki.factorio.com/Blueprint/de>.

Das Anbieten eines Webservices ist ein weiterer Ansatz zur Darstellung der Informationen. Das System teilt sich hierbei in ein Backend und ein Frontend auf. Ersteres übernimmt die Deadlock- und Performanceanalysen und sendet die ermittelten Werte als Antwort zurück an den Aufrufer. Es wird eine API angeboten, die einen Blaupausen String erhält und als Antwort den Durchsatz und die Deadlock-Freiheit zurückgibt. Diese API wird von dem Frontend verwendet, um die eingegebenen Blaupausen analysieren zu lassen. Auf der Client-Seite wird die Anzeige der Informationen veranlasst. Das Frontend kann der Nutzer in seinem Webbrower öffnen, ohne etwas auf seinem Gerät zu installieren. Auch kann die API von anderen Anwendungen aufgerufen werden und ermöglicht somit, dass diese beispielsweise in Mods für Factorio verwendet werden kann.

Für die funktionalen Aspekte ist es wichtig, dass der Nutzer auch eingibt, wie die Züge, die auf der Kreuzung fahren, beschaffen sind. Ein Zug mit einer hohen Anzahl an Wagons sperrt bei der Performanceanalyse mehr Abschnitte als ein kurzer Zug, was zur Folge hat, dass mehr Züge auf Freigabe des jeweiligen Abschnittes warten müssen. Der Nutzer sollte eine Menge an Zuglängen angeben, die diese Kreuzung befahren werden, damit die Analysen für diesen Anwendungsfall korrekte Ergebnisse liefern.

Funktional soll die Anwendung unter anderem erkennen, ob Deadlocks in der gegebenen Kreuzung auftreten können. Es stehen in diesem Fall zwei Handlungsoptionen zur Verfügung: Entweder wird die Analyse gestoppt, sobald ein Deadlock erkannt wird, oder es wird so lange analysiert, bis alle möglichen Deadlock-Kombinationen ermittelt worden sind.

Auch soll die Anwendung die Performance der gegebenen Kreuzung messen können. In diesem Fall werden in den verschiedenen Eingängen der Kreuzung einfahrende Züge simuliert und gemessen, wie viele durch die Kreuzung von einem Eingang zu einem Ausgang in einem festen Zeitintervall durchfahren.

All die zuvor genannten Punkte sind Anforderungen und mögliche Ansätze für die entwickelte Anwendung gewesen. Im nachfolgenden Kapitel wird im Detail die Beschaffenheit und die Funktionen der Applikation beschrieben und welche dieser Ansätze umgesetzt worden sind.

5. Implementierung eines Tools zur automatischen Analyse einer Factorio Zugkreuzung bezüglich der Deadlock-Freiheit und Performance

Dieses Kapitel beschreibt die Umsetzung der zuvor beschriebenen Anforderungen an die zu entwickelnde Applikation. Zuerst wird der Aufbau des Projektes beschrieben. Im Anschluss daran wird das Zugstrecken-Modell erläutert, auf dem alle Algorithmen der Anwendung ansetzen. Darauf aufbauend wird beschrieben, wie dieses Modell aus der eingegebenen Blaupause generiert wird. Danach wird der Algorithmus zur Erkennung von Deadlocks erläutert. Zum Schluss erfolgen Erklärungen bezüglich des Algorithmus zur Performance-Analyse.

5.1. Aufbau der Java Anwendung

Die Anwendung ist in Java entwickelt worden, da die Sprache Portabilität bietet. Die Portabilität erlaubt es, dass die Anwendung auf allen Computern laufen kann, die die JRE installiert haben. Auch ist die Entscheidung für Java aufgrund der persönlichen Präferenzen der Entwickler gefallen.

Für die Benutzerschnittstelle wird ein CLI verwendet, da die anderen Optionen in dem vorgeschriebenen Zeitraum nicht umsetzbar sind. Bei der Deadlock-Erkennung ist nur die Blaupause relevant. Die Länge der Züge kann vernachlässigt werden, da kleine Züge, welche nur aus einer Lok bestehen, dieselben Deadlocks im verwendeten Algorithmus erzeugen können, wie längere Züge mit Wagons. Zwar gibt es Fälle, wie in Abbildung 14, in dem ein Zug sich selbst blockieren sollte. Allerdings hält der Zug im Spiel nicht an und kollidiert trotz der platzierten Signale mit sich selbst. Aufgrund dessen wird dies als Edge Case betrachtet und bei der Deadlock-Erkennung nur mit dem kleinstmöglichen Zug gerechnet.

Da ein CLI verwendet wird, muss bei der Ausführung des Programms ein Pfad zu einer txt-Datei, die die Blaupause enthält, angegeben werden. Das Programm gibt die Ergebnisse der Berechnungen in Form einer json-Datei zurück. In dieser Json sind alle möglichen Deadlock-Pfade und das Zugstrecken-Modell, das das Programm erstellt hat, gespeichert. Aufgrund von mangelnder Zeit ist die Performance-Analyse nicht implementiert worden. Nichtsdestotrotz wird eine konkreter Implementierungsansatz im entsprechenden Abschnitt diskutiert.



Abbildung 14: Zug kollidiert mit sich selbst trotz Signale

Das Projekt ist mit Gradle erstellt worden und wird mit diesem Tool gebaut. Gradle ist ein Build-Werkzeug für die JVM.²⁰ Im Projekt ist der Code im Basis-Package „factorio.train.analyser“ in weitere Packages aufgeteilt, welche in Abbildung 15 einzusehen sind.

```

1      analyser/
2      decoder/
3      graph/
4      jsonmodels/
5      App.java
6      Matrix.java

```

Abbildung 15: Ordnerstruktur der Applikation

Im „factorio.train.analyser.analyser“ Package sind alle Klassen, die für die Deadlock-Erkennung und die Performancemessung benötigt werden. Die Klassen zur Decodierung der Blaupausen Zeichenkette sind im „factorio.train.analyser.decoder“ Package enthalten. Im „graph“ Package befinden sich die Klassen, die das im nachfolgenden Abschnitt beschriebene Zugstrecken-Modell aus der Matrix erzeugen. Im „jsonmodels“ sind Klassen, die für das Parsen des dekodierten Strings in ein Objekt benötigt werden. Die „App.java“ Klasse startet das Programm

²⁰Vgl. „Gradle User Manual“, zugegriffen 28. Februar 2024,
<https://docs.gradle.org/current/userguide/userguide.html>.

und ruft die entsprechenden Klassen auf, um mit den Analysen zu beginnen. Die „Matrix.java“ Klasse erzeugt eine Matrix aus dem Json-Objekt, das in „jsonmodels“ angelegt worden ist. Wie bereits angesprochen, erfolgt im nachfolgenden Kapitel eine Beschreibung des Zugstrecken-Modells, das benötigt wird, um die entsprechenden Funktionalitäten zu implementieren.

5.2. Das Zugstrecken-Modell

Um das Factorio Zugmodell in das Modell unserer Java Applikation zu übertragen, sind verschiedene Begrifflichkeiten definiert worden. Im Folgenden werden alle in dieser Arbeit verwendeten Begriffe definiert und es wird ein eigenes Modell vorgestellt.

Ein Entity ist ein Element aus der Blaupause. Ein Entity kann jedes Objekt sein, dass durch den Spieler platziert worden ist und somit in der Blaupause enthalten ist. Für dieses Projekt sind allerdings die Signale und die Schienen von Relevanz.

Ein Track ist ein Entity, das die Bezeichnung „track“ im Namen beinhaltet. In Abbildung 16 sind alle möglichen Tracks in Factorio abgebildet. Auch sind Signale Entities, die im Namen „(chain-)rail-signal“ tragen.



Abbildung 16: Alle Track-Varianten in Factorio

Eine Node ist ein möglicher Weg, den ein Zug innerhalb eines Abschnittes nehmen kann. Eine Node setzt sich aus mindestens einem Track zusammen. Eine Node verweist auf welche weiteren Nodes diese führt. In Abbildung 17 sind die Nodes mit N1, N2, N3 und N4 bezeichnet worden. Beispielsweise verweist N1 aufgrund des Signals, dass von dort aus nach N2, N3 oder N4 gefahren werden kann. N2, N3 und N4 verweisen dagegen nicht auf N1, da es aufgrund des Zugsignals nicht möglich ist von diesen aus N1 zu erreichen. Darüber verweist eine Node, ob diese durch eine andere Node über ein Kettensignal beschützt ist. Auch weiß eine Node, ob es innerhalb einer Blaupause selbst ein möglicher Eingang beziehungsweise Ausgang ist, in

welchem der Zug hinein- oder hinausfahren kann. Diese Eigenschaft ist wichtig, da in der Applikation nur Kreuzungen analysiert werden soll. Hierbei wird für die Deadlock- und Performanceanalyse ein Einstiegspunkt benötigt, wo einfahrende Züge simuliert werden können. Daher werden Blaupausen, die keine Eingangs- und Ausgangs-Nodes beinhalten von der Applikation abgewiesen.

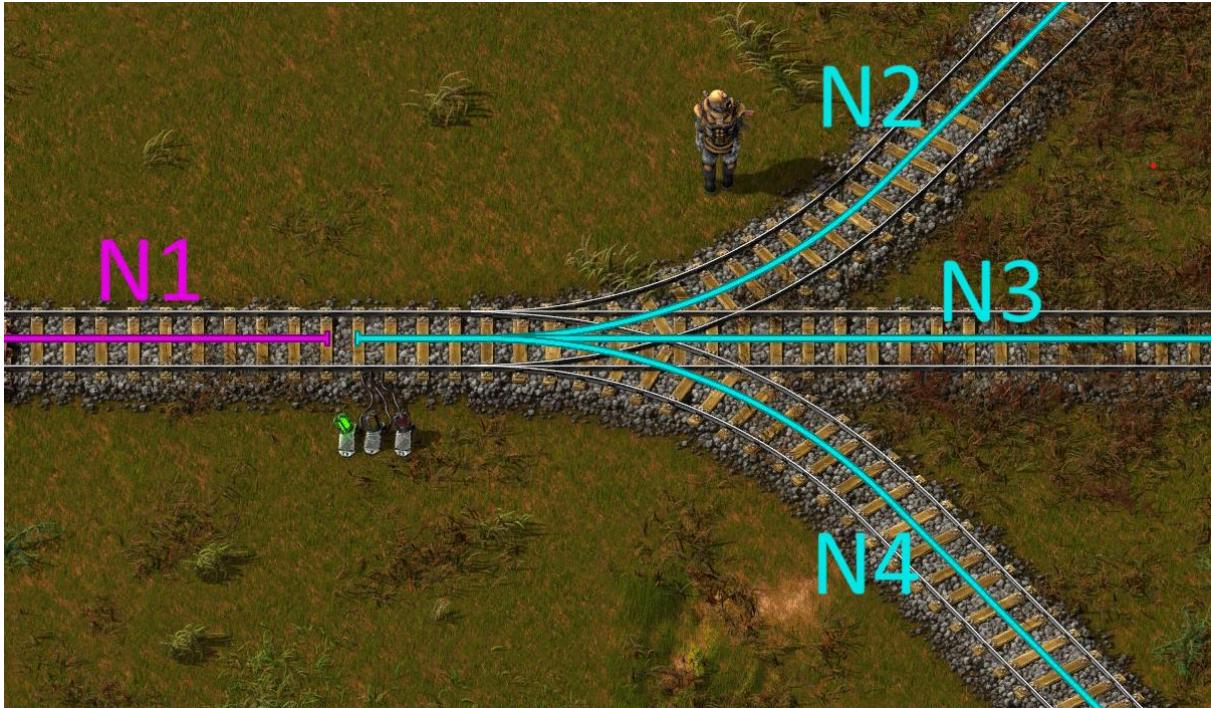


Abbildung 17: Nodes im Zugstrecken-Modell

Eine Section, welche in vorhergehenden Kapiteln als Abschnitt bezeichnet worden ist, besteht aus mehreren Nodes, die aufgrund von den gegenseitigen Überschneidungen und Verbindungen miteinander zusammengefasst worden sind. In Abbildung 17 gehören N2, N3 und N4 in dieselbe Section. Die Definition einer Section ist wichtig, um das gegenseitige Beeinflussen von Nodes untereinander nachzubilden. Wenn also N2 durch einen Zug belegt worden ist, dann wird die gesamte Section als belegt markiert und der Algorithmus erkennt, dass sich dort nicht weitere Züge befinden dürfen. Das heißt, dass beispielsweise der Algorithmus keine Züge in N3 simuliert, da dies von der Spiellogik unmöglich ist.

Der Graph, den unsere Anwendung erzeugt, um Factorios Zugmodell nachzubilden, beinhaltet alle erkannten Sections einer Blaupause. Die Analysealgorithmen nutzen den Graphen und arbeiten sich durch die Sections und ihre jeweiligen Nodes durch. Durch die Verweise der Nodes untereinander entsteht ein Netz an Abhängigkeiten, die die das Factorio Zugmodell nachbilden.

Nachdem diese Begriffe und das eigene Zugstrecken-Modell vorgestellt worden sind, wird im nachfolgenden Kapitel erklärt, wie aus einer Blaupause das Zugstrecken-Modell in der Anwendung erzeugt wird.

5.3. Erzeugung des Zugstrecken-Modells

Um ein Zugstrecken-Modell zu erzeugen, muss zuerst die Blaupause dekodiert und in ein Matrix-Objekt umgewandelt werden. Darauf basierend werden die verschiedenen Nodes und Sections generiert. In den nachfolgenden Kapiteln wird dieser Prozess beschrieben.

5.3.1. Dekodierung der Blaupausen-Zeichenkette

Eine Blaupausen-Zeichenkette ist eine base64-verschlüsselte und komprimierte Json-Zeichenkette.²¹ Die Java Applikation erzeugt ein Objekt der Klasse Decoder aus dem „factorio.train.analyser.decoder“ Package und ruft auf diesem Objekt die Methode „String decode()“ auf. In dieser Methode wird der übergebene base64-String entschlüsselt und dekomprimiert. Als Rückgabewert erhält der Aufrufer den entschlüsselten und dekomprimierten Json-String.

Dieser Json-String muss allerdings in ein Java-Objekt geparsed werden, um mit den dort enthaltenen Informationen arbeiten zu können. Im Package „factorio.train.analyser.jsonmodels“ sind Klassen deklariert worden, die von der Gson Bibliothek benötigt werden, um die Einträge des Json-Strings in Objekte zu übertragen. Hierbei wird in der Methode „create()“ der Klasse JsonBuilder mit dem entschlüsselten Json-String aufgerufen. In der Methode wird die Gson Bibliothek verwendet, um aus den Einträgen der Json ein Objekt der Klasse Json zu erzeugen, welches alle Informationen des Json-Strings enthält.

Nachdem die Informationen für das Programm nutzbar gemacht worden sind, wird im nächsten Kapitel beschrieben, wie aus dem Objekt der Klasse Json eine Matrix generiert wird.

5.3.2. Matrix-Generierung

Um bei der späteren Generierung des Graphens effizienter auf die Entities beziehungsweise Tracks zugreifen zu können, werden diese in einer Matrix gespeichert. Das Speichern in der 2D-Matrix ermöglicht es positionsbezogene Informationen zu erhalten. Für eine besseres

²¹Vgl. „Blaupause - Factorio Wiki“, Official Factorio Wiki, zugegriffen 21. Februar 2024, <https://wiki.factorio.com/Blueprint/de>.

Verständnis der folgenden Beschreibung der Matrixgenerierung ist in Abbildung 18 das Klassendiagramm der Matrix abgebildet.

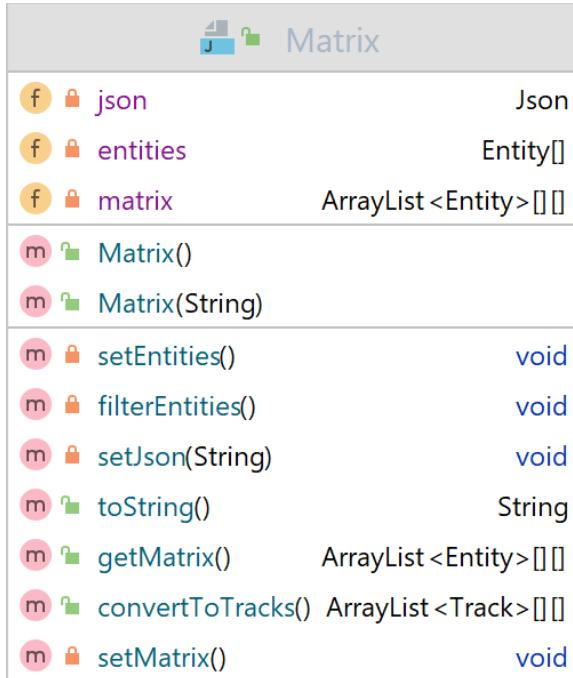


Abbildung 18: UML-Klassendiagramm der „Matrix.java“

Ein Objekt der Klasse Matrix wird mit dem Aufruf des Konstruktors „Matrix()“, bei dem der encodierte Blaupausen-String übergeben wird, erzeugt. Der Konstruktor erzeugt mithilfe des JsonBuilder ein Objekt der Klasse Json und kopiert sich das Array im Feld „entities“ in das eigene Feld „entities“. Bevor aus diesem Array die 2D-Matrix erzeugt wird, werden die „entities“ gefiltert. Hierbei werden alle Entities, die kein Track oder Signal sind, entfernt. Danach wird das Feld „matrix“ mithilfe des „entities“ Array erzeugt. Die Entities werden nach ihrer Position in die 2D-Matrix eingefügt. Die Position besteht aus den x- und y-Koordinaten des Entities. Da diese Koordinaten negativ oder auch Nachkommastellen beinhalten können, werden die x und y Werte entsprechend angepasst. Um beispielsweise Nachkommastellen zu vermeiden, werden alle Werte mit 2 multipliziert. Da auch mehrere Entities an derselben Stelle sich befinden können, wird eine Arrayliste an den entsprechenden Positionen gespeichert, um Elemente dynamisch hinzufügen zu können.

Nachdem die Entities in einer Matrix gespeichert worden sind, können diese für die Graph-Generierung verwendet werden, die im nachfolgenden Abschnitt beschrieben wird.

5.3.3. Graph-Generierung

Die Erstellung einer entsprechenden Datenstruktur, die die Wechselwirkungen nachbildet, welche in Factorio auftreten, ist für die Analysen unerheblich. Der Graph bildet das in einem vorhergehenden Kapitel besprochene Zugstrecken-Modell ab.

Um den Graphen zu erzeugen, müssen folgende Schritte durchgeführt werden.

Zuerst müssen die Tracks, welche miteinander verbunden sind, in eine Node zusammengefasst werden. Nachdem die Nodes erzeugt worden sind, erfolgt eine Zusammenfassung der Nodes, die sich gegenseitig überschneiden, zu einer Section. Bei dem Zusammenfassen der verschiedenen Elemente müssen allerdings die Beziehungen untereinander erhalten bleiben. Nodes müssen die Information, in welche Nodes sie münden, beibehalten. Im Folgenden wird dieser beschriebene Algorithmus genauer erläutert.

Die für die Umsetzung benötigten Klassen befinden sich im „factorio.train.analyser.graph“ Package des Projektes. In Abbildung 19 ist der Aufbau des Packages abgebildet.

```
1      Graph.java
2      LookUp.java
3      Node.java
4      Section.java
5      Track.java
```

Abbildung 19: Aufbau des „factorio.train.analyser.graph“ Package

Die Track Klasse, die in Abbildung 20 einzusehen ist, bildet das Grundgerüst, auf dem das Zugstrecken-Modell aufbaut. Ein Track verweist mit dem Attribut „nodes“ auf die Node, von denen der Track ein Teil ist. Ein Track hat eine feste Länge, die für die spätere Performanceanalyse benötigt wird. Diese Länge wird bei der Erstellung eines Track-Objektes anhand des übergebenen Namens ermittelt. Ein „curved-rail“ hat die Länge 3, während dagegen jede andere Art an Track die Länge 1 hat. Dieser Wert wird im „length“ Attribut persistiert.

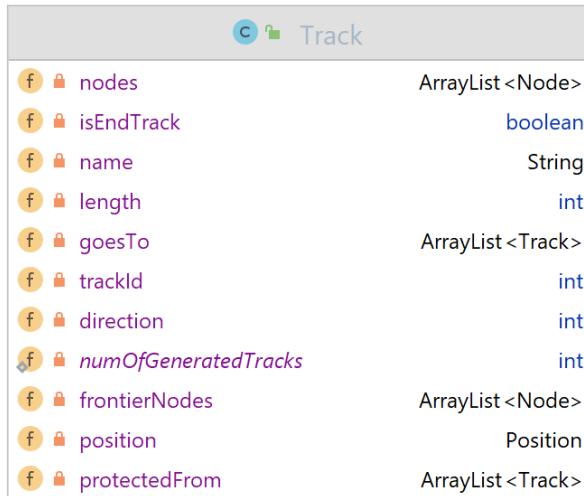


Abbildung 20: UML-Diagramm der Track Klasse

Das Feld „*isEndTrack*“ wird gesetzt, wenn beim Parsen des Tracks erkannt wird, dass der entsprechende Track an mindestens einer Stelle mit keinem anderen Track verbunden ist. In Abbildung 21 ist bei den pink markierten Tracks und dem gebogenen Track dieses Feld auf true gesetzt.



Abbildung 21: Tracks mit gesetztem „*isEndTrack*“ Flag

Das „*direction*“ Feld der Klasse Track spiegelt die Richtung des Tracks in Factorio wider. Die Richtung des Tracks wird bei der Erkennung, welche Tracks sich gegenseitig überschneiden beziehungsweise miteinander verbunden sind, benötigt. Beim Parsen der einzelnen Tracks zu einer Node wird das „*frontierNodes*“ Attribut benötigt. Das „*position*“ Feld gibt die Position des Tracks an. Hierbei ist wichtig zu beachten, dass hier die bereits angepassten Koordinaten der Matrix verwendet werden. Auch das „*position*“ Feld ist für die Erkennung von verbundenen beziehungsweise sich überschneidenden Tracks wichtig. Um die Abhängigkeiten der Nodes untereinander widerzuspiegeln, muss beim Parsen der Tracks diese erhalten bleiben. Hierfür werden die „*goesto*“ und „*protectedFrom*“ Felder benötigt.

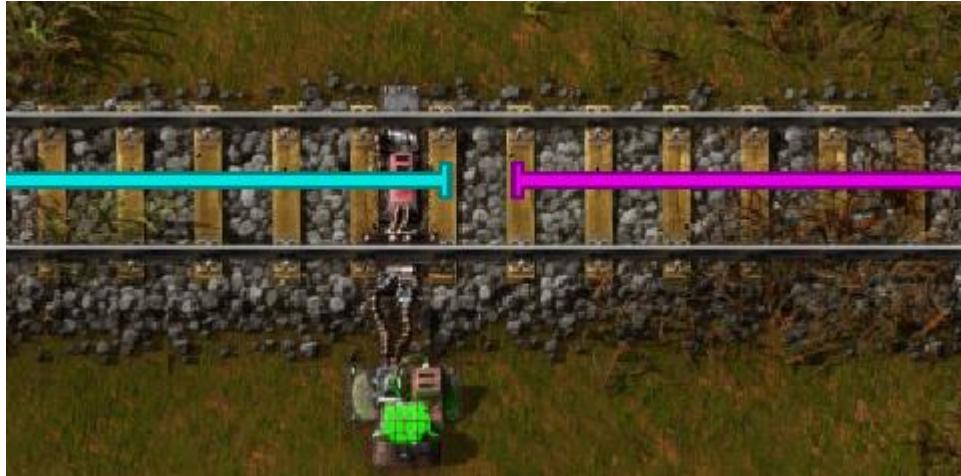


Abbildung 22: Beispiel für die „goesTo“ und „protectedFrom“ Attribute

Der in Abbildung 22 blaumarkierte Track verweist in seinem „goesTo“ Attribut auf den pinkmarkierten Track. Dagegen gibt der pinke Track im „protectedFrom“ Feld an, dass er vom blaumarkierten Track durch ein Kettenignal beschützt wird. Diese Felder bilden die Abhängigkeiten wieder und werden später auf die Eltern-Nodes der entsprechenden Tracks übertragen.

Eine Node setzt sich aus mehreren Tracks zusammen. Der Aufbau der Node Klasse ist in Abbildung 23 abgebildet. Eine Node gehört immer zu einer Section, welche wiederum beliebig viele Nodes, die sich gegenseitig überschneiden, zusammenfasst. Eine Node verweist auf die Section im „section“ Attribut.

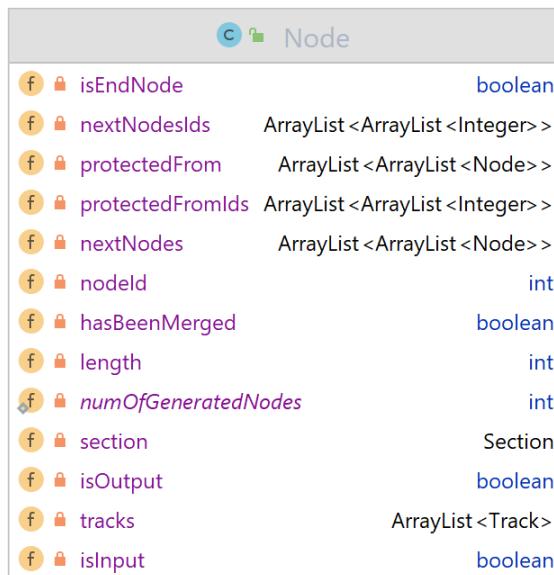


Abbildung 23: UML-Diagramm der Node Klasse

Das Flag „hasBeenMerged“ wird beim Zusammenfügen mehrerer Nodes zu einer Section benötigt. Wie genau das Zusammenfassen verschiedener Nodes zu einer Section funktioniert, wird in einem späteren Abschnitt beschrieben. Eine Node verweist mit dem „tracks“ Feld auf die Tracks, aus denen sie zusammengesetzt ist. Die Länge einer Node wird aus der Summe der Länge der Tracks im „tracks“ Feld errechnet und im „length“ Attribut gespeichert. Das „nextNodes“ Feld gibt an, welche Nodes ein Zug, der sich in dieser Node befindet, erreichen kann. In diesem sind allerdings nur Nodes enthalten, die durch das Befolgen der vorgegebenen Fahrtrichtungen erreicht werden können. Eine Node hat zwei Enden, in denen ein Zug Einfahren oder Ausfahren kann. Diese zwei Enden werden durch die Aufspaltung der „nextNodes“ in eine Arrayliste von maximal zwei Arraylisten abgebildet. In Abbildung 24 ist beispielhaft dieser Sachverhalt abgebildet. Die Node N1 hat in dieser Grafik insgesamt zwei Arraylisten im „nextNodes“ Attribut. Die eine Liste verweist auf die Node N0, während dagegen die andere auf die Nodes N2 und N3 verweist. Wäre beispielsweise N1 nicht bidirektional, dann würde „nextNodes“ nur eine Arrayliste enthalten. Diese Modellierung ist notwendig, um die Richtung, die ein simulierter Zug fahren kann, zu bestimmen.

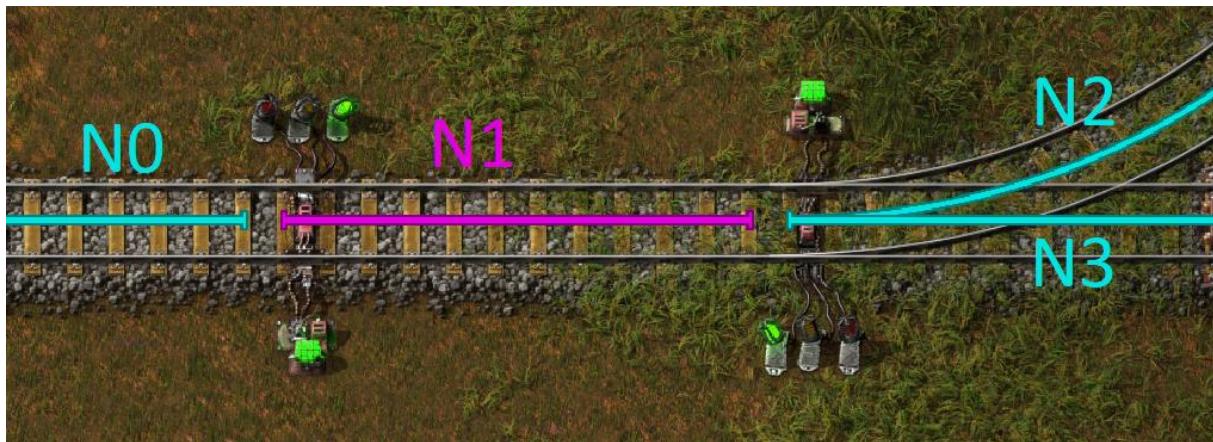


Abbildung 24: Beispiel Aufteilung der „nextNodes“ und „protectedFrom“ Attribute

Das „protectedFrom“ Feld gibt, wie das gleichnamige Track-Attribut, an von welchen Nodes es durch ein Kettenignal beschützt wird. Dieses Attribut ist gleich aufgebaut wie das „nextNodes“ Feld. Hier wird auch eine Unterscheidung der verschiedenen Enden einer Node vorgenommen. Für die Deadlock-Erkennung wird dieses Attribut benötigt.

Um zu erkennen, ob eine Node ein Endstück ist, wird das „isEndNode“ Flag gesetzt. Es wird auf true gestellt, wenn mindestens ein Ende der Node mit keiner weiteren Node verbunden ist. Aus diesem Flag werden die „isInput“ und „isOutput“ Flags abgeleitet. Wenn eine Node ein offenes Ende hat und beispielsweise das „nextNodes“ Feld leer ist, dann ist diese Node ein

Ausgang. Ein Ausgang ist hierbei eine Node, die Züge nehmen können, um die Kreuzung zu verlassen. Ein Ausgang ist in unserer Annahme immer frei. Das bedeutet, dass dort nie ein Zug stehen kann, der diese Ausgangs-Node blockiert. Ein Eingang ist eine Node, die ein Endstück ist und dessen „nextNodes“ Attribut nicht leer ist. Ein Eingang ist im Gegensatz zu einem Ausgang blockierbar. In den Analysen werden Eingänge verwendet, um Züge von dort aus starten zu lassen. Eine Node kann sowohl ein Eingang als auch ein Ausgang sein. In einem späteren Kapitel werden diese genauer beschrieben.

Die Section Klasse enthält alle Nodes, die sich gegenseitig überschneiden und dadurch zusammengehören. Im „nodes“ Attribut sind alle Nodes der Section gespeichert. Das „isFree“ Flag wird für die Deadlock-Erkennung benötigt und gibt an, ob in einer der Nodes ein Zug sich befindet, der aufgrund dessen die gesamte Section sperrt.

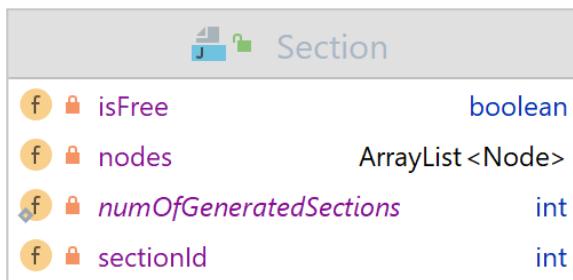


Abbildung 25: UML-Diagramm der Section Klasse

Um die Tracks zu parsen, wird die LookUp Klasse benötigt. Es werden in dieser Klasse verschiedene statische Methoden angeboten, um für ein übergebenes Track-Objekt Positionen zu finden, an denen sich Tracks befinden könnten, die sich mit dem gegebenen Track überschneiden oder sich miteinander verbinden. In Abbildung 26 sind die verschiedenen Methoden, die beim Parsen benötigt werden, abgebildet.

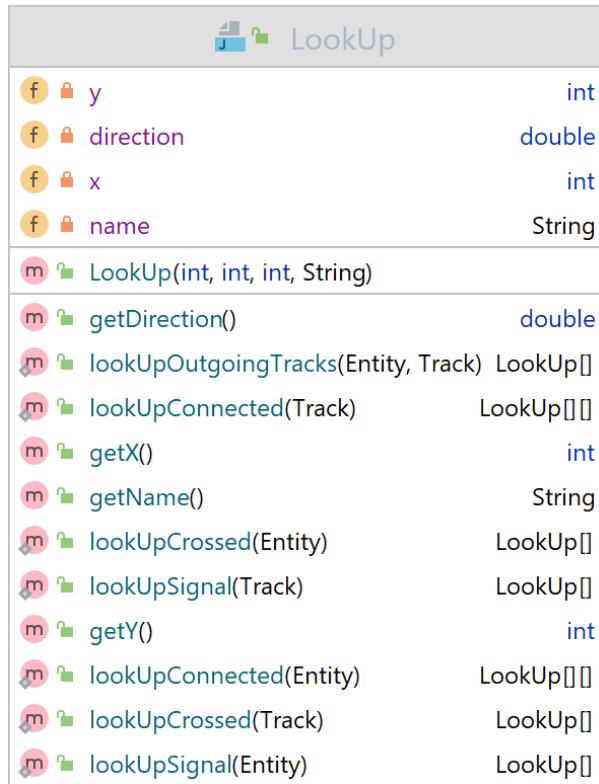


Abbildung 26: UML-Diagramm der LookUp Klasse

Ein Objekt der Klasse LookUp gibt die Position, den Namen und die Ausrichtung eines möglichen Tracks an. Eine Herausforderung bei der Erstellung der Klasse ist es, die verschiedenen Möglichkeiten, wie sich Tracks überschneiden oder miteinander verbinden nachzubilden. Die „lookUpConnected()“ Methode gibt ein zweidimensionales Array an LookUp-Objekten zurück. Hierbei werden die Tracks, die sich mit einem Track verbinden können in zwei Mengen aufgeteilt, da ein Track zwei Enden besitzt. Diese Unterscheidung ist für das Parsen der Tracks notwendig. Abbildung 27 ist ein Auszug aus dem Code der Methode „lookUpConnected()“.

```

public static LookUp[][][] lookUpConnected(Entity entry) {
    LookUp[] entrance = {};
    LookUp[] exit = {};
    int xPos = (int) entry.getPosition().getX();
    int yPos = (int) entry.getPosition().getY();
    switch (entry.getName()) {
        case "straight-rail":
            switch (entry.getDirection()) {
                case 0: // |
                    entrance = new LookUp[] {
                        new LookUp(xPos, yPos + 4, 0, "straight-rail"),
                        new LookUp(xPos - 2, yPos + 10, 5, "curved-rail"),
                        new LookUp(xPos + 2, yPos + 10, 4, "curved-rail")
                    };
                    exit = new LookUp[] {
                        new LookUp(xPos, yPos - 4, 0, "straight-rail"),
                        new LookUp(xPos - 2, yPos - 10, 0, "curved-rail"),
                        new LookUp(xPos + 2, yPos - 10, 1, "curved-rail")
                    };
                    break;
            }
    }
}

```

Abbildung 27: Auszug der „lookUpConnected()“ Methode

In diesem Auszug wird ein zweidimensionales LookUp Array aus den zwei verschiedenen Mengen erstellt. Die Bestimmung der Positionsoffsets wurde händisch aus Factorio ausgelesen und in die verschiedenen Methoden übertragen. Hierbei ist der Debugmodus in Factorio verwendet worden, der die Positionen in Form von x und y Koordinaten, die in Abbildung 28 als grüne Kästchen dargestellt werden, anzeigt. In der Abbildung 28 werden alle möglichen Tracks dargestellt, die sich mit einem horizontalen geraden Track verbinden können. Jede mögliche Richtung, die ein Track haben kann ist ein eigenständiger Fall in der switch-Kontrollstruktur und erfordert eine wiederum eine erneutes Auslesen der entsprechenden Positionsoffsets.



Abbildung 28: Anzeige der Track Positionen im Debug-Modus

Der Aufwand für die manuelle Erstellung der „lookUpConnected()“ Methode ist im Vergleich zur „lookUpCrossed()“ gering gewesen, da für jede möglich Track-Art und Ausrichtung maximal 6 LookUp-Objekte manuell erstellt werden mussten. In Abbildung 29 ist ein Auszug gezeigt, wie ein kurviger Track von einem anderen Track überkreuzt werden kann. Im besten Fall müssen mindestens 60 LookUp-Objekte erzeugt werden. Der Implementierungsaufwand für die LookUp Klasse war aufgrund dessen erheblich, da es insgesamt 14 Fälle in der switch-Kontrollstruktur gibt, die für die „lookUpConnected()“ und „lookUpCrossed()“ implementiert werden mussten.



Abbildung 29: Auszug an Kombinationen von sich überschneidenden Tracks

Die Graph Klasse verwendet alle zuvor beschrieben Klassen, um das Zugstrecken-Modell zu erzeugen. In Abbildung 30 ist das UML-Diagramm der Klasse abgebildet. Das Attribut „sections“ enthält alle erzeugten Sections, der zu analysierende Blaupause. Die Deadlock-Erkennung verwendet dieses Feld als Einstieg. Das „nodes“ Attribut wird für das Parsen der Nodes genutzt, um Ergebnisse zwischenzuspeichern. Außerhalb dieser Klasse wird dieses Feld nicht verwendet. Auch das Attribut „matrix“ wird nur innerhalb der Graph Klasse für verschiedene Zwecke verwendet.

Graph	
f	nodes
f	matrix
f	graphId
f	numOfGeneratedGraphs
f	sections
m	Graph(String)
m	isInsideCurrentSection(Track, Track)
m	filterLookupsToTrack(LookUp[], ArrayList<Track>[][])
m	filterSignals(LookUp[], ArrayList<Entity>[][])
m	getSections()
m	mergeSections(ArrayList<Track>[][])
m	getGraphId()
m	parseTracksToNodes(Track, ArrayList<Track>, ArrayList<Track>[][], ArrayList<Track>) ArrayList<Node>
m	setSections()
m	hasSignalAttached(Track)
m	setIoFlags()
m	mapTrackFieldToNodeField(ArrayList<Track>[][])
m	connectTracks(ArrayList<Track>, Track, ArrayList<Track>[][])
m	setMatrix(String)
m	isNextTrackInSection(Track, ArrayList<Track>)

Abbildung 30: UML-Diagramm der Graph Klasse

Wenn ein neues Graph-Objekt generiert wird, wird im Konstruktor die verschlüsselte Blaupausen Zeichenkette übergeben. Der Konstruktor ruft mit der „setMatrix()“ Methode die Matrix Klasse mit der Zeichenkette auf und initialisiert damit das „matrix“ Attribut. Nachdem das „matrix“ Attribut initialisiert worden ist, wird die Methode „setSections()“ ausgeführt. Diese Methode benutzt dieses Attribut, um die Sections zu generieren.

```

public Graph(String encodedString) {
    nodes = new ArrayList<>();
    sections = new ArrayList<>();
    setMatrix(encodedString);
    setSections();
}

```

Abbildung 31: Konstruktor der Klasse Graph

In Abbildung 32 ist die gesamte „setSections()“ Methode abgebildet. Zuerst überprüft die Methode, ob das „matrix“ Feld initialisiert worden ist. Im Anschluss daran wird eine Arrayliste erstellt, die speichert, ob ein Track bereits besucht worden ist. Auch wird mithilfe des „matrix“ Attributs die Variable „tracks“ initialisiert, da in einem Matrix-Objekt die Elemente innerhalb einer Matrix als Entities gespeichert werden und für das Parsen allerdings Objekte der Klasse Track benötigt werden.

Nach der Initialisierung wird durch das „tracks“ Array durchiteriert und es wird nach Einträgen gesucht.

```

private void setSections() {
    if (matrix == null)
        return;

    ArrayList<Track> knownTracks = new ArrayList<>();
    ArrayList<Track>[][] tracks = matrix.convertToTracks();

    for (int x = 0; x < tracks.length; x++) {
        for (int y = 0; y < tracks[x].length; y++) {
            if (tracks[x][y] == null)
                continue;
            for (int i = 0; i < tracks[x][y].size(); i++) {
                Track track = tracks[x][y].get(i);
                if (knownTracks.contains(track))
                    continue;
                parseTracksToNodes(track, null, tracks, knownTracks);
                Section section = new Section(nodes);
                sections.add(section);
                nodes = new ArrayList<>();
            }
        }
    }
    mergeSections(tracks);
    mapTrackFieldToNodeField(tracks);
    setIoFlags();
}

```

Abbildung 32: Die setSections() Methode

Wenn ein Track gefunden wird, überprüft zuerst das Programm, ob dieser bereits besucht worden ist. Falls dies nicht zutrifft, wird die „parseTracksToNodes()“ Methode aufgerufen. Diese Methode erhält als Parameter den gefundenen Track, den bereits gegangenen Pfad, die „tracks“ Variable und die „knownTracks“ Variable.

Das Ziel dieser Methode ist es, dass anhand des gegebenen Tracks die verschiedenen Nodes erzeugt werden. Hierbei werden nur Nodes erzeugt, die sich in derselben Section befinden und miteinander verbunden sind. Überkreuzen sich zwei Nodes und sind aufgrund dessen in einer Section, werden diese in diesem Schritt ignoriert. Nach einem erfolgreichen Aufruf der „parseTracksToNodes()“ Methode wird eine neue Section, die die neu erzeugten Nodes enthält, zum „sections“ Attribut hinzugefügt. Das Zusammenfügen mehrerer Sections, die eigentlich zusammengehören, wird mit der „mergeSections()“ Methode, welche in einem späteren Abschnitt erläutert wird, durchgeführt. An der Abbildung 33 wird der Algorithmus zum Parsen der Tracks beispielhaft erklärt.

Bei einem Aufruf der parseTracksToNodes() Methode, werden alle pink markierten Nodes generiert und in eine Section gespeichert. Alle Tracks, die dabei traversiert worden sind, werden in der „knownTracks“ Arrayliste abgespeichert, damit diese nicht nochmals aufgerufen werden. Der Algorithmus stoppt hierbei, wenn ein Track ein offenes Ende hat oder wenn ein Zugsignal die Section beendet.



Abbildung 33: Zusammengehörige Nodes nach einem Parsedurchlauf

Die Grundidee bei diesem Parsing Algorithmus ist, dass die beiden Enden eines übergebenen Tracks in eine Callback Arrayliste und Frontier Arrayliste aufgeteilt werden. Diese beiden Mengen werden mit dem Aufruf der „lookUpConnected()“ Methode erzeugt. Welche der beiden Mengen als Callback oder Frontier klassifiziert wird, hängt vom bereits gegangenen Pfad des Algorithmus ab. Die Methode ruft sich selbst rekursiv auf und übergibt einen Pfad von Tracks mit dem „visitedTracks“ Parameter, die bereits besucht worden sind. Die Menge, die den vorherigen Track enthält, wird als Callback-Menge definiert. Die andere Menge ist dementsprechend die Frontier-Menge. Wird die Methode initial aufgerufen, ist der übergebene Pfad null. In diesem Fall werden die Mengen willkürlich eingeteilt. In Abbildung 34 wird bei einem initialen Aufruf der Methode auf den Track T1 die Menge {T0} als Frontier und die Menge {T2, T4} als Callback klassifiziert. Die Idee ist hierbei alle Nodes, die durch einen rekursiven Aufruf auf die Frontier und die Callback Tracks erzeugt werden, einzusammeln und anschließend miteinander zu verschmelzen.

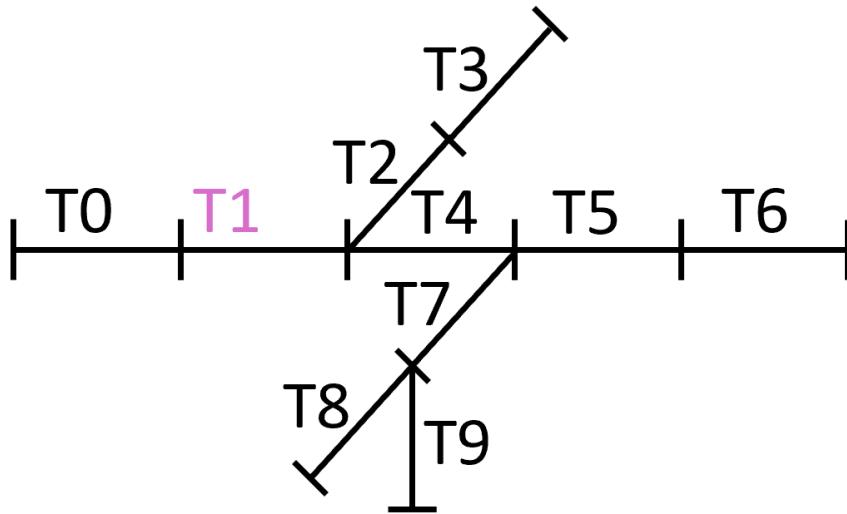


Abbildung 34: Vereinfachte Darstellung verbundener Tracks

Die Methode wird sich aufgrund dessen selbst rekursiv auf dem Track T0 aufrufen. Zuerst wird hier die Frontier- und Callback-Menge bestimmt. Da T1 der vorhergehende Track gewesen ist, ist die Menge {T1} die Callback-Menge. Die Frontier-Menge ist hier leer, da T0 mit keinem Track an diesem Ende verbunden ist.

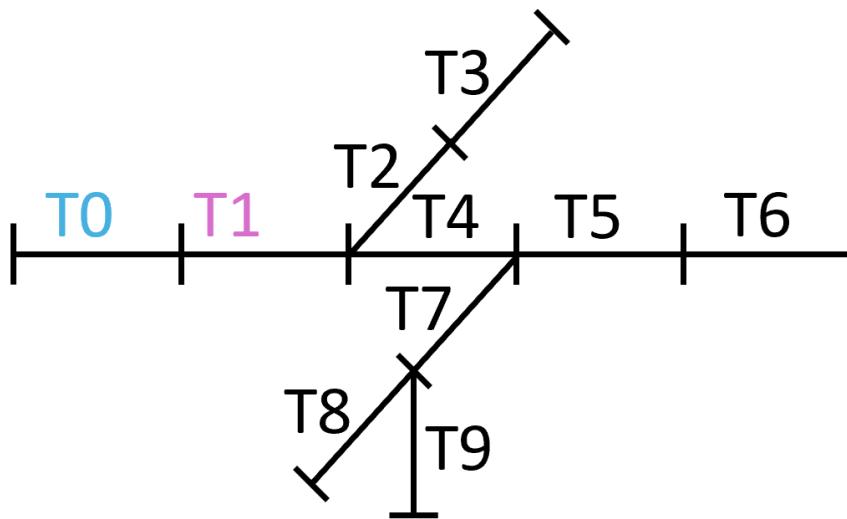


Abbildung 35: Rekursiver Aufruf der Frontier-Tracks

Ist die Callback-Menge oder die Frontier-Menge leer, dann wird das Flag „isEndTrack“ des Tracks auf wahr gesetzt. Wenn wie bei T0 die Frontier-Menge leer ist, dann wird, statt rekursiv die Frontier-Menge aufzurufen, der Basisfall ausgeführt. Im Basisfall wird eine neue Node generiert, der T0 zugeordnet wird. Diese Node wird als Rückgabewert zurückgegeben. Somit erhält T1 eine Node aus der Frontier Menge. Nachdem die Frontier Menge rekursiv aufgerufen worden ist, werden die Tracks der Callback-Menge rekursiv aufgerufen. Daher wird T2

aufgerufen. T2 wird wiederum T3 aufrufen. Da T3 ein Signal besitzt, wird die Frontier-Menge, welche in unserem Beispiel nicht leer ist, entleert und dadurch der Basisfall erzwungen.

```

if (!isNextTrackInSection(currentTrack, frontier)) {
    connectTracks(frontier, currentTrack, tracks);
    frontier = new ArrayList<>();
}
if (!isNextTrackInSection(currentTrack, callBack)) {
    connectTracks(callBack, currentTrack, tracks);
    callBack = new ArrayList<>();
}
}

```

Abbildung 36: Entleeren der callback und frontier Arraylisten

Wie in Abbildung 36 zu erkennen ist, werden auf den Tracks, die durch das Signal getrennt sind, Methoden aufgerufen, um zu erkennen, ob diese getrennt sind, und um diese miteinander zu verbinden. Die „isNextTrackInSection()“ Methode überprüft zuerst, ob der „currentTrack“ ein Signal besitzt. Hierbei wird ein Getter aufgerufen, welcher wiederrum die entsprechende Methode der LookUp Klasse aufruft. Wenn der „currentTrack“ ein Signal besitzt, iteriert die Methode für die Tracks innerhalb der Frontier-Menge und ruft wiederum die Methode „isInsideCurrentSection()“ auf und übergibt den „currentTrack“ und den jeweiligen Track aus der Frontier. Beim Aufruf dieser Methode werden die Parameter vertauscht. Dies hat das Ziel zu erkennen, ob von einem Frontier-Track aus gesehen der „currentTrack“ zur selben Section gehört. Sobald „isInsideCurrentSection()“ false zurückgibt, wird die Schleife abgebrochen und ebenfalls ein false zurückgegeben. Diese Methode enthält die Logik, um zu erkennen, ob zwei Tracks bei einem Signal noch zusammengehören. In Abbildung 37 ist dies an einem grafischen Beispiel abgebildet.

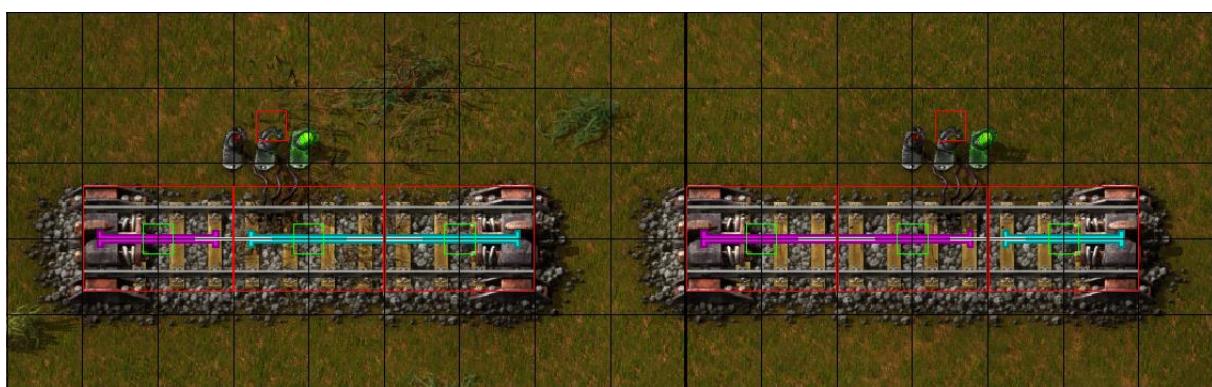


Abbildung 37: Sections-Zugehörigkeit durch verschiedene Platzierungen auf demselben Track

In Abbildung 37 ist unser „currentTrack“ der mittlere Track. Die Frontier wird entleert, wenn der linke Fall vorliegt, da der pinke markierte Track nicht mehr zu den blau markierten gehört. Der rechte Fall ist, wenn der „currentTrack“ selbst nicht mehr zu der blauen Section gehört.

Dieser Fall wird allerdings an anderer Stelle im Code abgefangen. Dies wird bei der Erläuterung gewisser Bedingungen, die gelten müssen, um einen rekursiven Aufruf auszuführen.

Die „`isInsideCurrentSection()`“ Methode erhält zwei Tracks als Parameter und gibt einen Wahrheitswert aus, wenn der „`currentTrack`“ Parameter und der „`nextTrack`“ Parameter in derselben Section befinden. Der „`nextTrack`“ Parameter ist der Track, der das Signal besitzt. Um zu ermitteln, ob der „`currentTrack`“ und der „`nextTrack`“ zur selben Node zusammengehören, werden verschiedene Berechnungen durchgeführt.

In Abbildung 38 sind die Vektoren, die Berechnung, ob zwei gerade horizontal oder vertikale Tracks zur selben Node gehören, dargestellt.



Abbildung 38: Abstandsvektoren ausgehend vom „`currentTrack`“

In der Methode wird zuerst überprüft, ob die beiden Tracks gerade und horizontal oder vertikal sind. Wenn dies zutrifft, wird die Länge des Vektors von der Position des „`currentTracks`“ zur Position des Signals berechnet. Wenn dieser Vektor kleiner als 4.3 ist, dann gehören die Tracks nicht zu derselben Node. In der Abbildung 38 wäre dies der linke Fall.

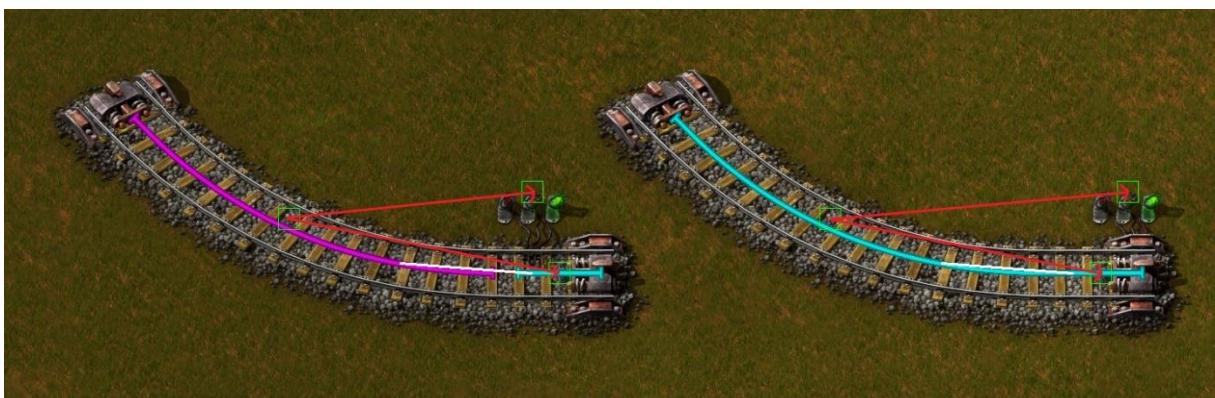


Abbildung 39: Abstandsvektoren ausgehend von einem kurvigen „`currentTrack`“

Bei der Berechnung, ob ein kurviger Track und ein gerader Track zusammengehören, werden auch wieder die Längen der Vektoren zum Signal und zum „`nextTrack`“ berechnet. Wenn das

Signal näher als der „nextTrack“ ist, dann gehört „nextTrack“ zu einer anderen Node und die Methode gibt false zurück. Dies ist die Grundidee aller vektorbasierten Berechnungen.

Bei geraden diagonalen Tracks wird in der Methode eine andere Berechnung durchgeführt, um zu ermitteln, ob diese zur selben Node gehören.



Abbildung 40: Diagonale Tracks mit Signalen

Ob zwei diagonale Tracks zusammengehören, hängt von der Position und der Richtung der entsprechenden Tracks ab. Wenn die linke Kombination an Tracks dieselbe x-Koordinate besitzen, dann gehören die beiden Tracks unabhängig von der Signalposition zur selben Node. Ist allerdings die x-Koordinate verschieden, dann sind diese nie in der gleiche Node. Die rechten beiden Kombinationen verhalten sich analog zu dem zuvor Beschriebenen. Das bedeutet, dass wenn die y-Koordinate identisch ist, diese Tracks immer zur gleichen Node gehören. Die Methode gibt false bei unterschiedlichen x- beziehungsweise y-Koordinaten und true bei gleichen x- beziehungsweise y-Koordinaten zurück.

Ein Sonderfall tritt bei der Erkennung auf, wenn ein kurviger Track auf einen diagonalen Track trifft. Eine Berechnung mithilfe der Vektoren ist nicht möglich, da in manchem Richtungskombinationen diese zu einer Node gehören und in manchen nicht. Um diesen Sonderfall abzufangen, mussten diese Kombination in der Methode fest codiert werden. Der Sonderfall ist in Abbildung 41 einsehbar.

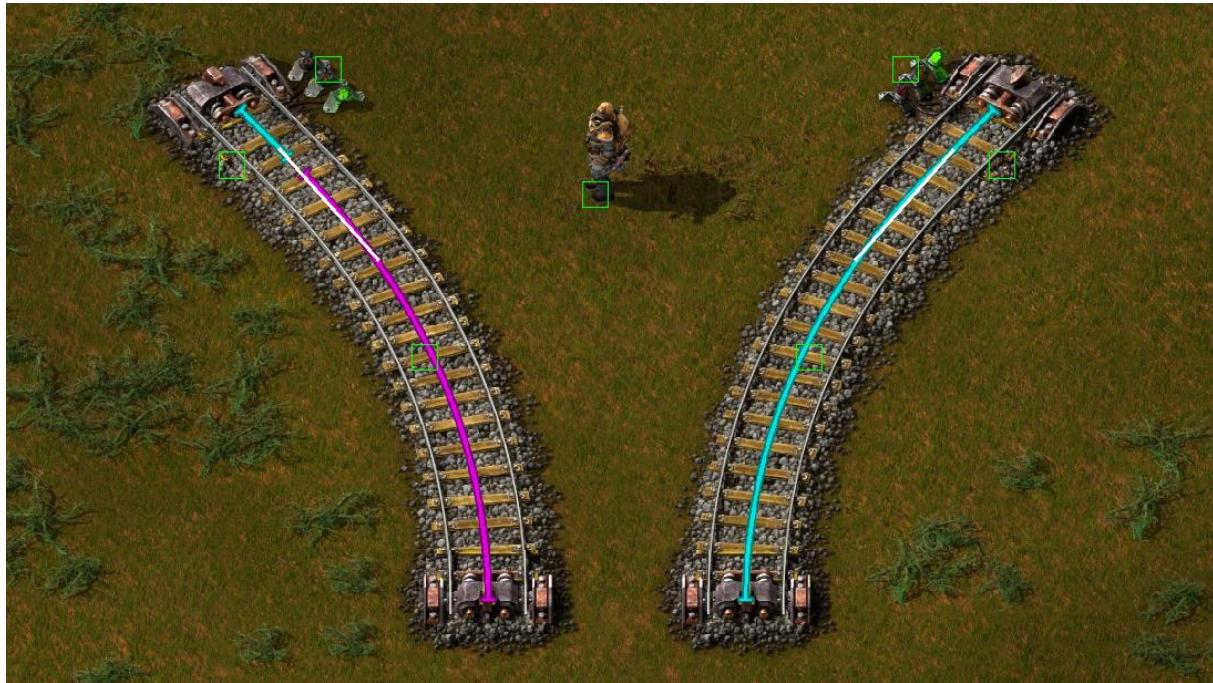


Abbildung 41: Kombination von kurvigen Tracks mit diagonalen Tracks

Nachdem T3 seinen Basisfall aufgrund deines Zugsignals erreicht hat und eine Node erzeugt und zurückgegeben hat, wurde T4 und schließlich T5 rekursiv aufgerufen. T5 hat auch T6 als seine Frontier zuerst rekursiv aufgerufen und eine Node zurückerhalten.

Da bei T5 die Menge der Callback Tracks aus mehreren Tracks besteht, müssen alle außer der vorhergehende Track T4 rekursiv aufgerufen werden.

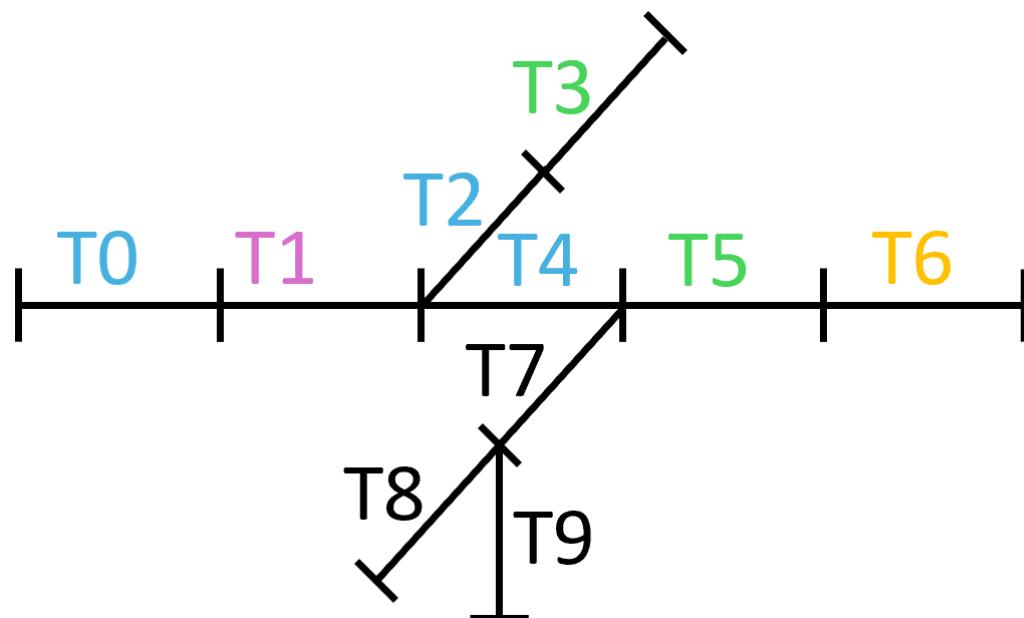


Abbildung 42: Rekursive Ausführung bis T5

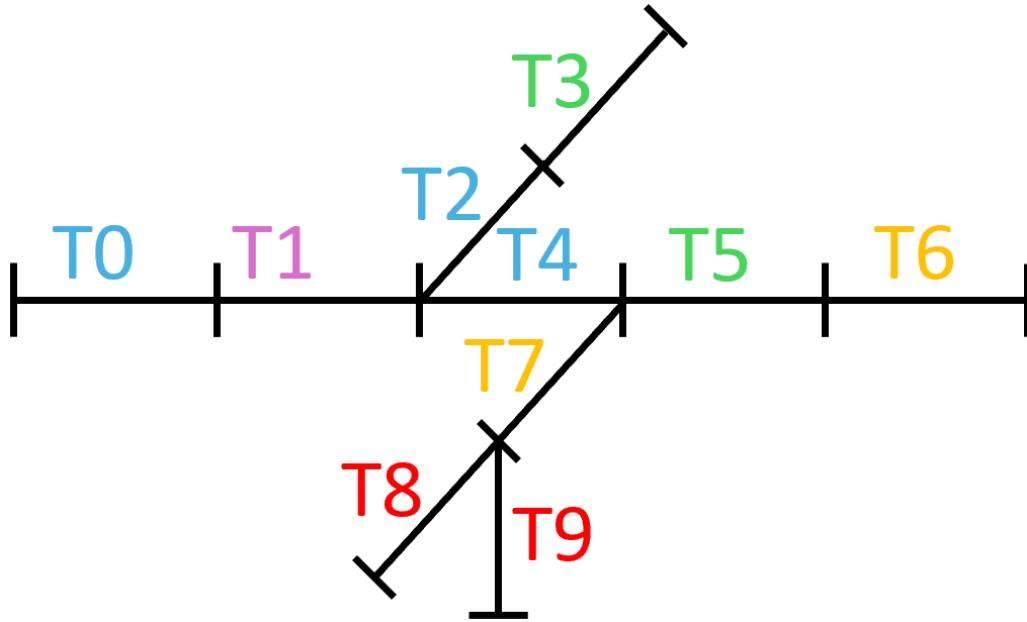


Abbildung 43: Rekursiver Aufruf der Callback Tracks von T5

Daher wird T7, wie in Abbildung 43 dargestellt, rekursiv aufgerufen. T7 wird wiederum T8 und T9 als Frontier-Tracks aufrufen, welche jeweils eine Node zurückgeben. Aufgrund dessen erhält T5 von seinen Callback-Tracks insgesamt zwei Nodes und von seinen Frontier-Tracks eine Node. Bevor T5 die Node, die er aus dem Aufruf des Frontier-Tracks erhalten hat, zurück an T4 gibt, muss T5 die Nodes aus dem Aufruf des Callback-Tracks mit diesen ausmultiplizieren. Das Ausmultiplizieren ist notwendig, um Sub-Pfade korrekt parsen zu können. Beispielsweise erhält T5 eine Menge an Nodes {N1, N2} von dem Callback-Track und eine Menge an Frontier Nodes {N3}. Das Ziel beim Ausmultiplizieren ist es, eine neue Node zu erzeugen, die aus einem Tupel von jeweils einer Callback-Node und Frontier-Node besteht. Im Beispiel würden insgesamt 2 neue Nodes N4 und N5 entstehen, die sich aus <N1, N3> und <N2, N3> zusammensetzen. N4 und N5 werden in das „nodes“ Attribut des Graphen geschrieben, da diese nach dem Ausmultiplizieren vollständig sind. Nach dem Multiplizieren wird die Menge {N3} der Frontier-Nodes von T5 nach T4 zurückgegeben.

Insgesamt erhält T1 von seiner Frontier eine Node und von seinem Callback zwei Nodes. Da T1 der Einstieg gewesen ist und keinen vorhergehenden Track besitzt, wird T1 auch seine Frontier-Nodes uns Callback-Nodes miteinander multiplizieren und in das „nodes“ Feld schreiben. Dadurch ist die Methode erfolgreich ausgeführt worden und es sind alle verbundenen Nodes zum „nodes“ Feld hinzugefügt worden.

Nachdem alle Nodes und ihre Sections mithilfe der Schleife in Abbildung 32 erzeugt worden sind, müssen die Sections, welche sich gegenseitig überschneiden, miteinander verbunden werden. In Abbildung 44 ist ein möglicher Zustand des „sections“ Attributs gezeigt. In diesem Beispiel befinden sich im „sections“ Attribut die Sections S1 und S2. In Factorio gehören diese Sections zusammen und werden als eine einzige Section zusammengefasst. Um solche Sections miteinander zu kombinieren, erfolgt der Aufruf der „mergeSections()“ Methode.

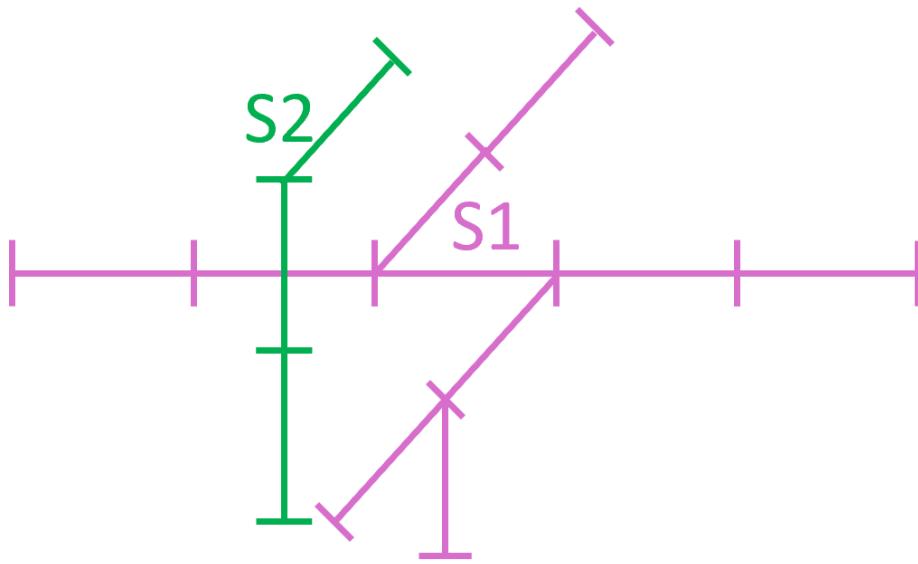


Abbildung 44: Inhalt des „sections“ Attribut nach dem Durchlaufen der Schleife

Die Methode durchläuft mit einer Schleife das „sections“ Feld und fügt dessen Nodes in eine Queue hinzu. Für jede Node in dieser Queue werden die einzelnen Tracks durchlaufen und überprüft, ob diese von anderen Tracks geschnitten werden. Dies erfolgt mit dem Aufruf der „getCrossed()“ Methode, die intern die „lookUpCrossed()“ Methode der LookUp Klasse auruft. Wenn eine Node aus der Queue aufgerufen wird, wird diese aus der Queue entfernt. Wenn ein Track geschnitten wird, dann werden die Nodes des überschneidenden Tracks der Queue und einer Arrayliste „checkNodes“, die alle besuchten Nodes auflistet, hinzugefügt. Um eine Endlosschleife zu vermeiden, wird für die jeweilige Node das Flag „hasBeenMerged“ auf true gesetzt. Nachdem die Queue komplett geleert worden ist, werden alle Nodes, welche der Arrayliste hinzugefügt worden sind, zu einer Section zusammengefasst. Diese Section wird der Arrayliste „mergedSections“ hinzugefügt. Danach wird „checkNodes“ für den nächsten Durchlauf der Schleife, die durch „sections“ iteriert, entleert.

Nachdem erfolgreich über das „sections“ Attribut iteriert worden ist, wird dieses mit dem „mergedSections“ überschrieben. Nach erfolgreicher Ausführung der Methode, bilden alle Sectionen den Verhalt in Factorio getreu nach.

Nachdem die Sectionen miteinander vereint worden sind, wird im nächsten Schritt durch die Ausführung der „mapTrackFieldToNodeField()“ einige Attribute der Tracks auf deren Parent-Nodes übertragen. Die Methode geht alle Tracks durch und überträgt die „goesTo“ und die „protectedFrom“ Felder auf das entsprechende Pendant der Parent-Nodes. Beispielsweise werden bei einem Track, bei dem das Attribut „goesTo“ einen anderen Track referenziert, die Parent-Nodes ermittelt und in deren „nextNodes“ Feld die Parent-Nodes des referenzierten Tracks eingefügt. Dadurch werden die Abhängigkeiten, die die Nodes aufgrund der Signale untereinander besitzen, im Graphen nachgebildet. Auch wird das Flag „isEndNode“ der Parent-Nodes eines Tracks, der ein offenes Ende hat, auf wahr gesetzt. Dieses Flag wird im nächsten Verarbeitungsschritt benötigt.

Zuletzt wird die „setIoFlags()“ Methode aufgerufen. Diese Methode iteriert über alle Sections und deren Nodes und sucht Nodes, die das „isEndNode“ Flag auf true gesetzt haben. Wird eine Node gefunden, bei dem dieses Flag gesetzt ist, dann wird ermittelt, ob die Node ein Eingang oder ein Ausgang der Kreuzung ist. In Abbildung 45 ist ein Auszug aus der Methode abgebildet.

```

if (node.getIsEndNode()) {
    if (node.getNextNodes().isEmpty()) {
        node.setIsOutput(true);
    } else {
        if(node.getNextNodes().get(0).get(0).getNextNodesMerged().contains(node))
            node.setIsOutput(true);
        node.setIsInput(true);
    }
}

```

Abbildung 45: Logik zur Erkennung einer eingehenden bzw. ausgehenden Node

Wenn bei einer Node das „nextNodes“ Attribut leer ist, dann ist diese Node ein Ausgang der Kreuzung. Ist das „nextNodes“ Feld nicht leer, dann ist diese Node ein Eingang. In diesem Fall muss auch überprüft werden, ob eine andere Node die gegebene Node in „nextNode“ referenziert. Falls dies der Fall ist, dann ist diese Node sowohl ein Eingang als auch ein Ausgang der Kreuzung.

Nachdem die „setIoFlags()“ Methode ausgeführt worden ist, ist auch die „setSections()“ Methode beendet. Die Ausführung der „setSections()“ befüllt das „sections“ Attribut mit verschiedenen Sections, die Nodes enthalten, welche sich gegenseitig referenzieren. Dadurch wird ein Netz, das eine Kreuzung in Factorio korrekt repräsentiert, aufgespannt. Dieses Netz wird bei der Deadlock-Erkennung und bei der Performanceanalyse benötigt. Beide Themen werden in den folgenden Kapiteln weiterverfolgt.

5.4. Deadlock-Erkennung

Der generierte Graph kann auf Deadlocks untersucht werden. Das grundsätzliche Vorgehen dabei ist, dass in jede Node, von einer Input-Node beginnend, ein Zug simuliert wird, der nicht in die nächste Node einfahren kann. Der Grund, wieso der Zug nicht in die nächste Node einfahren kann, muss sein, weil dort ein weiterer Zug steht. Auf diese Art und Weise werden alle Nodes mit einem Zug belegt, bis ein Zyklus entsteht oder bis alle möglichen Wege mit einer Output-Node enden. Dafür wird ein Rekursions-Algorithmus entwickelt, der den Graphen Knoten für Knoten durchgeht und einen Zyklus sucht. Dieser startet bei jeder Node, die als Input markiert wurde. Anschließend wird die Section dieser Node als belegt markiert und es wird in die nächste Node gesprungen. Das genaue Vorgehen wird im Folgenden noch genauer erläutert. Ein Deadlock wird erkannt, wenn die nächste Node in die gesprungen wird, in einer Section ist, die als belegt markiert ist.



Abbildung 46: Beispiel Deadlock in Factorio

In der Abbildung 46 wird dies nochmal besser ersichtlicher. In jede Node wird ein Zug gesetzt, wodurch diese belegt ist, es darf also kein weiterer Zug hineinfahren. Die nächste Node nach

der lila farbenen, wäre eine Node in dem gelben Sektor. Diese Section wurde allerdings vorher bereits als belegt markiert, weshalb der Zug aus der lila Section dort nicht hinfahren kann. Es wurde ein Zyklus und daraus folgend ein Deadlock gefunden. Diese Art der Überprüfung wiederholt sich für jede Input-Node in der Blaupause. Eine wichtige Voraussetzung hierfür ist die Annahme, dass in jede Node mindestens eine Lok passt. Dies kann in sehr seltenen Fällen nicht der Fall sein, wodurch es Abweichungen geben kann. Diese Einschränkung dient zur Vereinfachung der Deadlock-Analyse und die eventuell resultierende Fehlerquote ist vernachlässigbar. Dies wird dem Nutzer aber auch vorher erläutert.

Andere Varianten für die Deadlock-Erkennung wären möglich, aber weniger effizient. Beispielsweise das Abwarten eines Timers, ob eine Output-Node in einer freien Section innerhalb einer gewissen Zeit erreicht wird. Da die Größe der Blaupause variieren kann, müsste der Timer oft länger warten als nötig, weshalb die andere Variante implementiert wird.

5.4.1. Rekursions-Algorithmus

Für die Deadlock-Erkennung wird eine eigene Klasse definiert. Diese benötigt den komplett geparsten Graphen, um auf alle Sections und Nodes als Objekte zugreifen zu können. Als Ergebnis wird ein Objekt der Klasse Result zurückgegeben. Diese enthält eine Liste, um Endlosschleifen zu vermeiden. Da Nodes, die durch ein Zug-Kettensignal geschützt sind, nicht belegt werden können, sondern erst die nächste Node überprüft werden muss, um zu verifizieren, dass ein Zug einfahren kann, bleiben geschützte Nodes immer frei. In einem Schienennetz, dass ausschließlich aus geschützten Nodes besteht, droht die Gefahr, dass sich ein Zyklus bildet, der nicht registriert werden kann, da die Nodes immer frei bleiben. Deswegen werden die Sprünge von geschützten Nodes in ihre nächsten Nodes in dieser Liste gespeichert und kein zweites Mal ausgeführt. Eine weitere Liste ist der Deadlock-Pfad. Damit der Nutzer nach Ausführung des Programmes mehr Informationen hat als die Erkenntnis, dass irgendwo in seiner Blaupause ein Deadlock vorkommen kann, wird der Pfad der Nodes, der diesen Deadlock verursachen kann in der richtigen Reihenfolge gespeichert. Diese Liste kann im Anschluss in beispielsweise einem simplen Userinterface leichter visualisiert werden, da die Node-Objekte dieselben sind, die ebenfalls im Komplettgraphen vorkommen. Das Userinterface wird in einem späteren Kapitel genauer erläutert. Zum Schluss gibt es noch ein „isDeadlock“ Attribut, das zu Beginn den Wert false hat. Dieser Wert wird nur auf true geändert, wenn der Algorithmus einen Deadlock findet. Dieser Wert ist vor allem aus Debug

Gründen da, um schneller verschiedene Blueprints zu testen, aber auch um das Ergebnis zu verifizieren. Der genaue Code der Rekursion wird im Folgenden erläutert.

Zunächst wird die „deadlockDetection()“ Methode aufgerufen. Hier wird eine Liste aus Result-Objekten erzeugt, da für jede Input-Node ein Result erzeugt wird und alle am Ende zurückgegeben werden sollen. Danach wird in allen Sections und allen darin enthaltenen Nodes nach den existierenden Input-Nodes gesucht, um von dort aus die Rekursion zu starten. Sollte eine Input-Node gefunden werden, wird ein neues Result-Objekt erzeugt und der Rekursionsmethode mit der entsprechenden Input-Node übergeben. Nach Abschluss der Rekursion wird das Result-Objekt der Liste an Results hinzugefügt und alle existierenden Sections werden wieder freigegebenen, damit der Rekursionsprozess für eine weitere Input-Node wieder von vorne anfangen kann.

```
private ArrayList<Result> deadlockDetection(ArrayList<Section> sections) {
    ArrayList<Result> result = new ArrayList<Result>();
    for (Section section : sections) {
        for (Node node : section.getNodes()) {
            if (node.getIsInput()) {
                Result resultOnePath = new Result();
                Result deadlockPath = recursion(node, resultOnePath, null);
                result.add(deadlockPath);
                for (Section sectionFree : sections) {
                    sectionFree.setIsFree(true);
                }
            }
        }
    }
    return result;
}
```

Abbildung 47: Die „deadlockDetection()“ Methode

Für die eigentliche Logik der Deadlock-Erkennung wird die Rekursionsmethode genutzt. Diese benötigt als Parameter die Node, von der aus agiert werden soll, das bisherige Resultat, dessen Deadlock-Pfad erweitert wird, und die jeweilige Vorgänger-Node, um nachvollziehen zu können, aus welcher Richtung simuliert wird. Zu Beginn ist die erste Node eine Input-Node mit dem Vorgänger Null. Nun wird sich mit einem Getter-Aufruf die Parent-Section dieser Node geholt. Sollte diese als belegt markiert sein, bedeutet dass, das hier bereits ein Zug steht und ein Deadlock vorhanden ist. Am Anfang kann dies nicht der Fall sein, weil noch keine Node als belegt markiert wurde. Die darauffolgende Logik wird in einer for-Schleife für alle Nodes in der Section aufgerufen, sodass am Ende durch die Rekursion jeder mögliche mit der Input-Node

verbundene Weg abgegangen wird. Zunächst wird überprüft, ob die Node ein Output ist. Eine Output-Node gilt als immer frei, da hier der Weg zu Ende ist und es für das Programm unbekannt ist, wie das Schienennetz darüber hinaus aufgebaut ist. Deswegen wird hier die for-Schleife mit einem „continue“ unterbrochen und dasselbe wird für die nächste Node der Section aufgerufen. Es gilt allerdings zu unterscheiden, ob eine Node nur eine Output-Node ist oder im Fall einer bidirektionalen Schiene zusätzlich auch eine Input-Node ist. Da wir auch hier nicht wissen, wie die Schiene weiter aufgebaut ist, es allerdings eine Input-Node ist, geht das Programm davon aus, dass hier ein Zug stehen kann. Deswegen gilt der Abbruch der for-Schleife nur für reine Output-Nodes. Handelt es sich also um keine Output-Node oder eine Output-Node, die zusätzlich eine Input-Node ist, wird die Section vorläufig schonmal auf belegt gesetzt und die Node wird dem Deadlock-Pfad hinzugefügt.

```

if (nodeInSection.getIsOutput() && !nodeInSection.getIsInput())
{
    continue;
}

if(!nodeInSection.getIsOutput() || nodeInSection.getIsOutput() &&
nodeInSection.getIsInput())
{
    nodeSection.setIsFree(false);
}

```

Abbildung 48: Überprüfung auf besondere Fälle

Im Anschluss wird überprüft, ob die Node eine geschützte Node ist. Dafür besitzt die Klasse Node eine Liste an Nodes, von denen sie beschützt wird. Handelt es sich um eine ungeschützte Node wird die Rekursionsmethode für jede nächste Node aufgerufen. Als Parameter wird die nächste Node als zu betrachtende Node, das Result mit dem bisherigen Pfad und die aktuelle Node als Vorgänger mitgegeben. Hier gelten keine Sonderregelung, da diese Node nur von normalen Zugsignalen umgeben ist. Sollte die Node doch von einem Zug-Kettensignal geschützt werden, wird das daran erkannt, dass die Liste der Beschützer-Nodes eine größere Länge als Null hat.

Wenn eine Node durch eine andere Node geschützt wird, gelten bestimmte Sonderfälle. Zunächst wird die Section der Node wieder als nicht belegt bezeichnet, da in dieser Node nicht direkt angenommen werden kann, dass hier ein Zug steht. Durch das Zug-Kettensignal muss erst die nächste Node überprüft werden, da sonst nicht in die geschützte Node hineingefahren werden darf. Eine Ausnahme besteht, wenn die geschützte Node zeitgleich eine Input-Node

ist. Da hier der weitere Streckenabschnitt unbekannt ist, darf diese Node als belegt gesetzt werden. Anschließend wird die Rekursionsmethode wieder für die nächsten Nodes der geschützten Node aufgerufen. Hier gilt es verschiedene Bedingungen zu beachten.

```
nextNode != predecessor && !findPair(resultYet.getChainSignalsVisited(),  
                                nextNode, nodeInSection)
```

Abbildung 49: Erster Sonderfall einer geschützten Node im Rekursionsaufruf

Die erste gibt lediglich vor, dass in jede nächste Node gesprungen werden darf, wenn diese nicht der Vorgänger ist und dieser Sprung nicht bereits durchgeführt wurde. Für die Vorgänger-Nodes gibt es andere Bedingungen, die im Weiteren noch erläutert werden. Da die Section einer geschützten Node nicht als belegt gesetzt wird, werden diese Sprünge in einer Liste abgespeichert, die vor jeder Rekursion mit der „findPair()“ Methode durchsucht wird, ob dieser Sprung bereits durchgeführt wurde, sodass keine endlosen Zyklen entstehen, die das Programm abstürzen lassen. Die Methode gibt entweder true oder false zurück, wenn sich die Kombination aus der Node und der nächsten Node als Dictionary in der Liste befinden.

```
nextNode.equals(predecessor) && nodeInSection.getIsInput() &&  
!findPair(resultYet.getChainSignalsVisited(), nextNode, nodeInSection)
```

Abbildung 50: Zweiter Sonderfall einer geschützten Node im Rekursionsaufruf

Der zweite Sonderfall besteht, wenn die nächste Node der Vorgänger ist. Das kann der Fall sein, wenn es sich um eine bidirektionale Schiene handelt, Züge also in beide Richtungen fahren können. Bei normalen Zugsignalen entsteht hier sofort ein Deadlock. Bei Zug-Kettensignalen gelten besondere Regeln. Ein Fall ist, wenn die geschützte Node eine Input-Node ist. Hier gibt es nur eine nächste Node, die zwangsläufig die Vorgänger-Node ist, weswegen von dort die Rekursion weitergehen kann. Auch hier gilt die Überprüfung, ob dieser Sprung bereits vorher durchgeführt wurde.

```
!inList(nodeInSection.getProtectedFrom(), predecessor) &&  
nextNode.equals(predecessor)) &&  
!findPair(resultYet.getChainSignalsVisited(), nextNode, nodeInSection)
```

Abbildung 51: Dritter Sonderfall einer geschützten Node im Rekursionsaufruf

Da es bereits klar ist, dass die aktuelle Node geschützt ist, wird hier überprüft von wem genau. Sollte es sich dabei nicht um den Vorgänger, darf dorthin gesprungen werden, um diese Node zu überprüfen, von der es abhängt ob in die aktuelle Node ein Zug einfahren könnte.

Sollten diese Bedingungen erfüllt werden, darf in die nächste Node gesprungen werden und der Sprung selbst wird als Dictionary der Liste des Result-Objektes hinzugefügt, um zukünftige Endlosschleifen zu vermeiden. Ein Durchlauf der Rekursion wird im Folgenden Kapitel genau betrachtet.

5.4.2. Rekursionsdurchlauf anhand eines Beispiels

Die Rekursion wird Schritt für Schritt anhand Folgendem Beispiel erklärt.

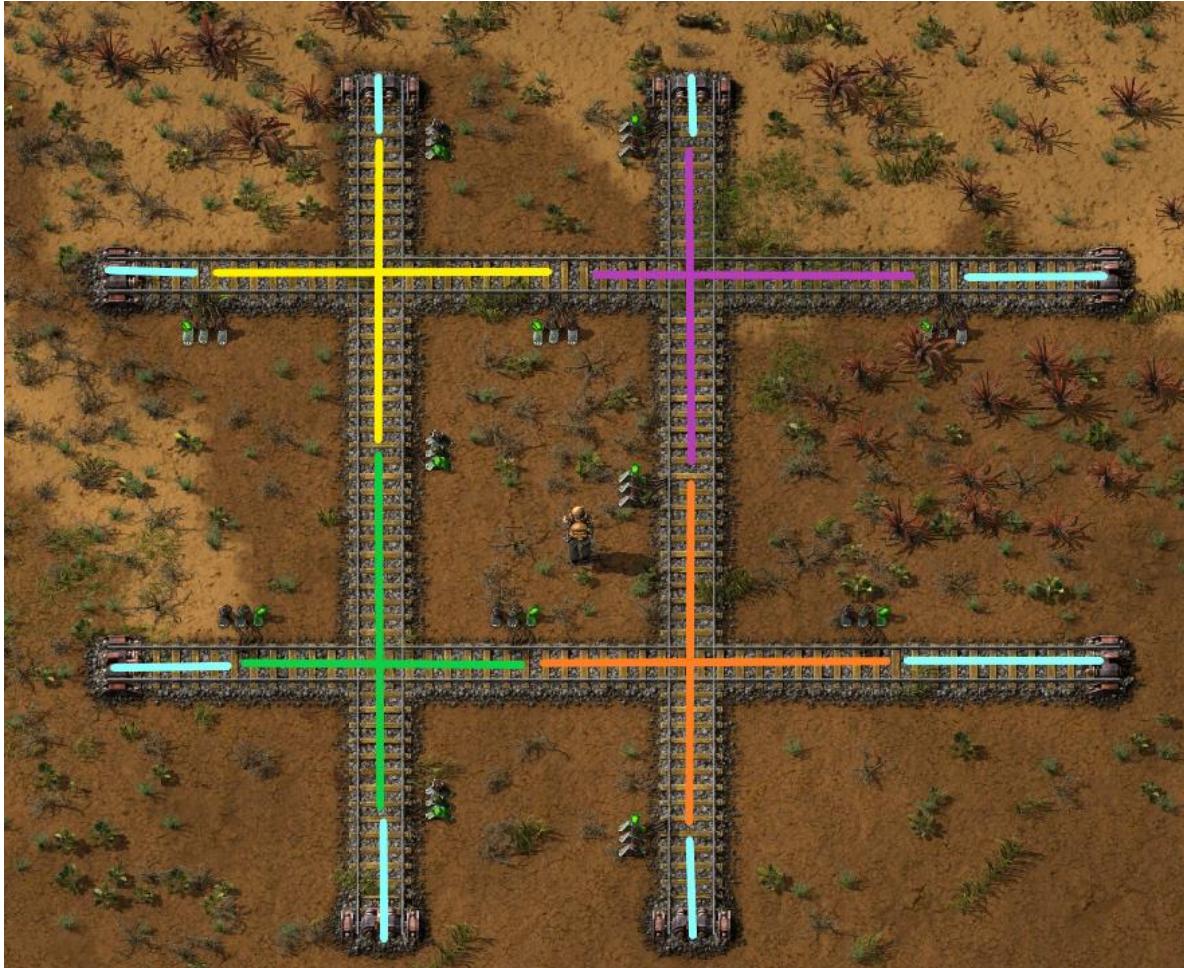


Abbildung 52: Beispiel-Kreuzung für den Rekursions-Algorithmus mit markierten Sektionen

Auf dem Bild sind farbig die existierenden Sections hervorgehoben. Die blaufarbigen zeigen alle Output- oder Input-Nodes an. In dieser Kreuzung gibt es keine Zug-Kettensignale oder bidirektionale Gleise. Die Richtung wird durch die Zugsignale bestimmt, die in einem anderen Kapitel erklärt worden sind. Die Rekursion wird für jede Input-Node durchgeführt und der von dort resultierende Weg auf Deadlocks geprüft. In diesem Beispiel wird lediglich ein einziger dieser Wege durchgeführt, beginnend bei der Input-Node oben links im Bild. Die Node in der sich die Rekursion befindet wird dunkelblau markiert und die Nodes die bereits besucht und als belegt markiert worden sind, werden rot dargestellt. Da keine Zugkettensignale vorhanden sind, wird nicht bei jedem Schritt erwähnt, dass diese Überprüfung fehlschlägt, sondern es werden nur die relevanten Überprüfungen beschrieben. Außerdem gibt es in diesem Beispiel viele doppelte Schritte, die nur einmal erläutert werden.



Abbildung 53: Beginn des Rekursions-Beispiels

Im ersten Schritt der Rekursion ist die aktuelle Node eine Input-Node. Diese befindet sich in einer eigenen Section, die überprüft wird, ob sie bereits belegt wurde. Es wird außerdem überprüft, ob es sich um eine reine Output-Node handelt, die übersprungen werden kann, dies ist aber nicht der Fall, weswegen die Section im Anschluss auf belegt gesetzt werden kann. Zusätzlich wird die Node dem bisherigen Deadlock-Pfad hinzugefügt und die Rekursion wird für alle nächsten Nodes aufgerufen. Als Parameter wird das Result-Objekt übergeben, um den weiteren Pfad und das Ergebnis nachvollziehen zu können sowie die aktuelle Node als Vorgänger.



Abbildung 54: Schritt 2 des Rekursions-Beispiels

Die vorherigen Schritte wiederholen sich in der nächsten Node. Es wird zunächst die Section überprüft, ob sie bereits auf belegt gesetzt wurde. Dies ist nicht der Fall. Da es sich auch nicht um eine Output-Node handelt, kann die Section nun auf belegt gesetzt werden und die Node wird dem Deadlock-Pfad im Result-Objekt hinzugefügt. Die Rekursion wird für die nächsten Nodes dieser Node aufgerufen mit aktualisierten Parametern. In dieser geschehen genau dieselben Schritte. Der darauffolgende Schritt beschreibt allerdings einen Sonderfall.

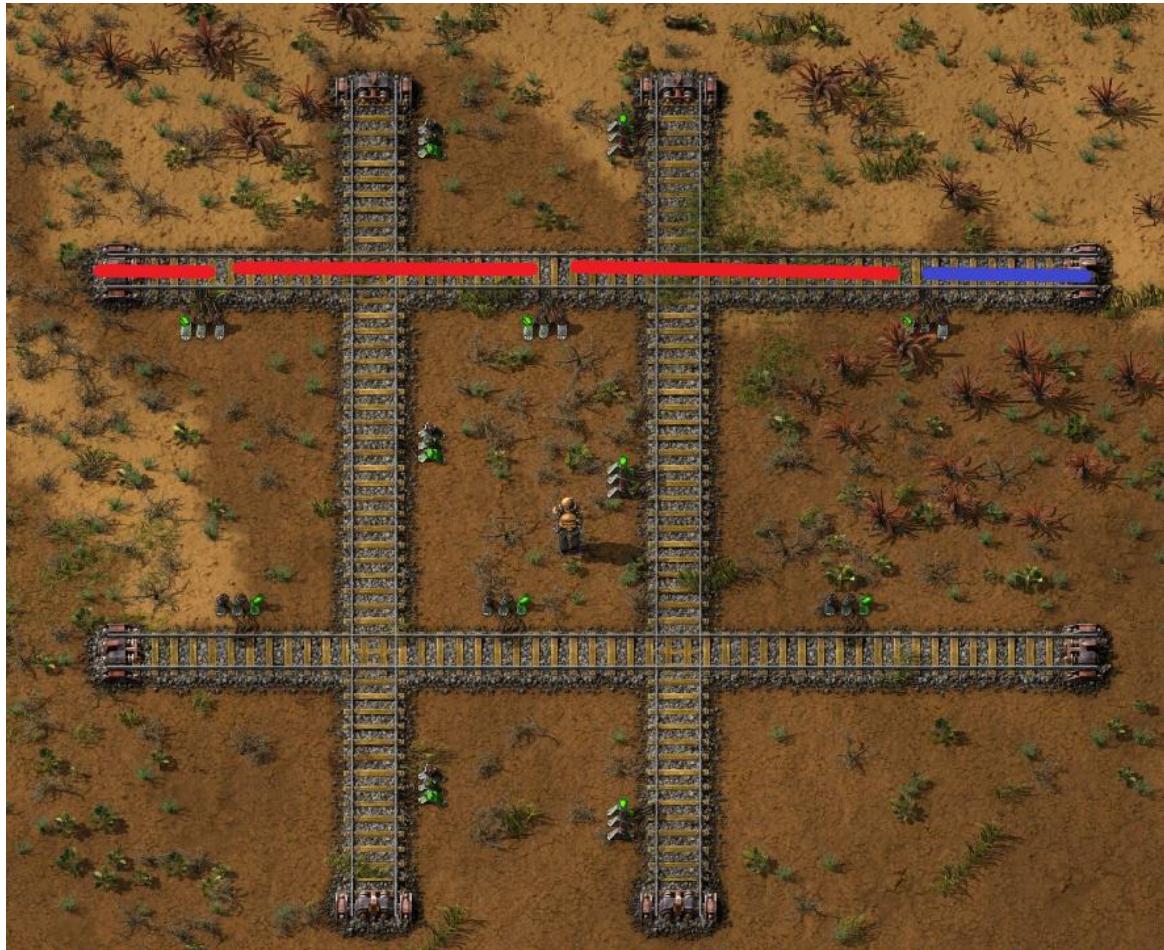


Abbildung 55: Sonderfall durch eine Output-Node im Rekursions-Beispiel

In diesem Fall ist die aktuelle Node eine Output-Node, die als immer frei definiert ist. Das bedeutet dieser Weg ist abgeschlossen und die Rekursion springt wieder eine Node zurück. Wichtig zu erwähnen ist, dass diese Node auch nicht dem Deadlock-Pfad hinzugefügt wird. Das bisher beschriebene Vorgehen wird für jede Node in der Section ausgeführt. Das bedeutet es wird nun der Weg der zweiten Node in der Section der vorher besuchten Node durchgegangen.

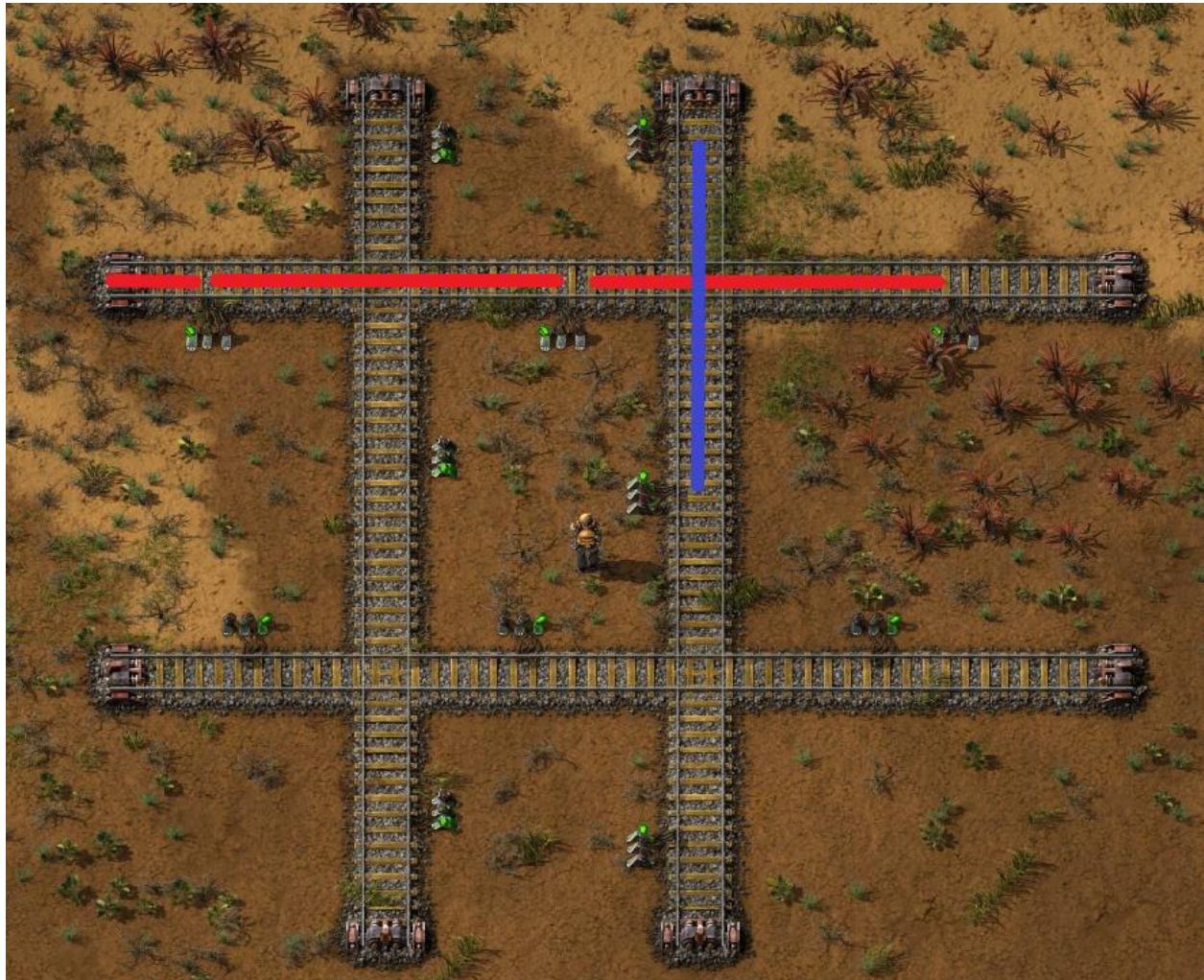


Abbildung 56: Neuer Weg im Rekursions-Beispiel

Die Überprüfungen bleiben dieselben. Da es sich um keine Output-Node handelt, wird die Section nochmal auf belegt gesetzt, wobei dies bereits von vorheriger Node in derselben Section getan worden ist. Im Anschluss wird der Deadlock-Pfad erweitert und die Rekursion wird für die nächste Node mit den benötigten Parametern aufgerufen. Auch dieser Weg wird in diesem Beispiel auf eine Output-Node treffen und der eben beschriebene Ablauf wird sich wiederholen. Das nächste Bild zeigt den nächsten Sonderfall, nachdem die Rekursion ähnlich wie bisher weitergelaufen ist.

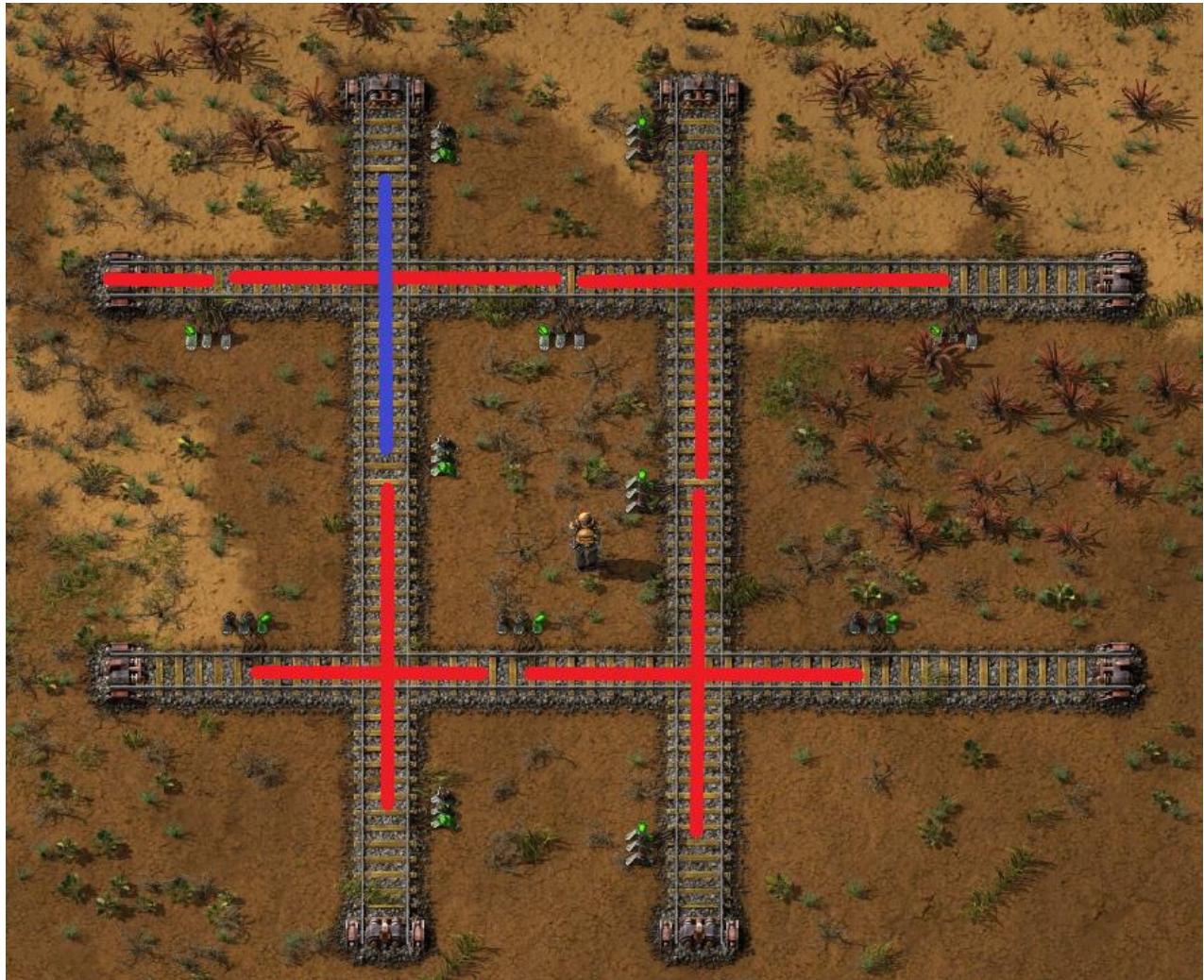


Abbildung 57: Gefundener Deadlock im Rekursions-Beispiel

In diesem Fall ist die aktuelle Node in einer Section die bereits als belegt worden ist. Das bedeutet, die Rekursion wird direkt abgebrochen und es wurde ein Zyklus, also auch ein Deadlock gefunden. Im Result-Objekt wird der Wert, ob ein Deadlock gefunden wurde auf true gesetzt. Danach werden alle Sections wieder freigegeben und die Rekursion beginnt von einer anderen Input-Node erneut.

Ein ähnlicher Ansatz dieser Art der Fortbewegung durch den Graphen wird in der Performanceanalyse genutzt, die Nachfolgend erläutert wird.

5.5. Performanceanalyse

Eine weitere Anforderung an das Projekt ist die Analyse der Performance einer Kreuzung hinsichtlich ihres Durchsatzes. Nachfolgend wird ein Ansatz diskutiert, der allerdings in der Anwendung aufgrund von mangelnder Zeit nicht implementiert worden ist.

Die Performanceanalyse an einer Kreuzung wird nur durchgeführt, wenn die vorhergehende Deadlock-Analyse keinen Deadlocks gefunden hat. Eine Kreuzung besitzt im Graphen

eingehende und ausgehende Nodes. Diese Nodes werden als Einstiegspunkt verwendet, um Züge, die sich durch diese Kreuzung bewegen, zu simulieren. Für die zu simulierenden Züge wird eine entsprechende Klasse definiert, die die Attribute eines Zuges enthält. Hierbei ist die Länge, die Beschleunigung, die Geschwindigkeit, die Treibstoffart, das Level der erforschten Bremskraft, der Bremsweg und die Nodes, in denen sich der Zug befindet, relevant.

Die Simulation der Züge iteriert in einem festen Zeitintervall über alle Züge, die sich in der Kreuzung befinden. In jedem Tick der Simulation wird anhand der Beschleunigung und der Geschwindigkeit des Zuges dessen Position aktualisiert. Die Position ist durch die Nodes, in denen sich der Zug befindet, definiert.

Um die Maximalgeschwindigkeit und die Beschleunigung eines Zuges zu bestimmen, muss bekannt sein, welcher Treibstoff verwendet wird.²² Des Weiteren wird die Beschleunigung und Entschleunigung eines Zuges durch dessen Länge bestimmt. Bei der Entschleunigung kommt zusätzlich die erforschte Stufe der Bremsfähigkeit hinzu. Damit ist gemeint, dass die anfängliche Bremskraft eines Zuges erheblich schwächer ist als die Bremskraft auf der maximal im Spiel erforschbaren Stufe. Die Bremskraft beeinflusst, wie viele Nodes im Bremsweg Attribut aufgeführt werden. Diese Informationen benötigt die Performanceanalyse, um Züge wie in Factorio simulieren zu können.

Wenn ein Zug am Ende einer Nodes angekommen ist und daher eine neue Node betreten wird, dann muss entschieden werden, in welche nachfolgende Node, die sich im „nextNodes“ Attribut der zu verlassenden Node befindet, der Zug einfahren kann. Für diese Entscheidung muss beachtet werden, dass der Zug eine Node, welche in seiner Fahrtrichtung liegt, aus der Menge „nextNode“ auswählt. Beispielsweise ist es unmöglich, dass ein Zug in Factorio bei einer bidirektionalen Node seine Fahrtrichtung ändert und in die zuvor entgegengesetzte Richtung fährt.

In Abbildung 57 ist ein Zug abgebildet, welcher eine Node verlässt. In diesem Fall wird in der Simulation entschieden, in welche Node aus „nextNodes“, der Zug einfahren soll. Da der Zug sich von links nach rechts im Bild bewegt, wird die Simulation entscheiden, dass der Zug in die rechte Node einfährt. Um entscheiden zu können in welche Node gefahren wird, speichert das Zug-Objekt die zuletzt mit der führenden Lok vollständig durchfahrene Node ab. Der Zug

²² Vgl. „Fuel - Factorio Wiki“, zugegriffen 14. März 2024, <https://wiki.factorio.com/Fuel>.

entscheidet sich dann in diesem Fall für eine Menge, in denen die vergangene Node nicht enthalten ist.

Wenn die Menge der möglichen Nodes mehr als ein Element enthält, dann entscheidet das Programm durch Zufall, welche Node ausgewählt wird. Ist allerdings eine der Nodes ein Ausgang, dann wird diese priorisiert.

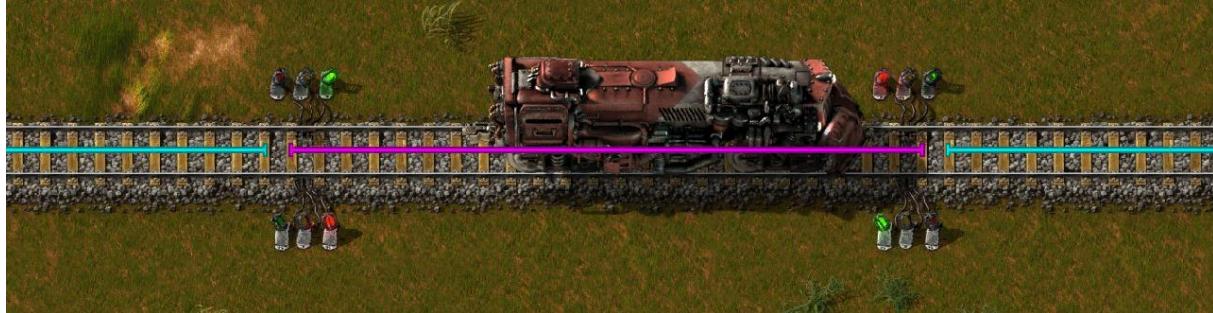


Abbildung 58: Zug beim Verlassen einer bidirektionalen Node

Ein fahrender Zug reserviert Nodes und Sections, die sich innerhalb seines Bremsweges befinden. Die Reservierung wird durchgeführt, um dem Zug ein rechtzeitiges Abbremsen zu ermöglichen. Erreicht ein Bremsweg eine gesperrte Section, dann wird der Zug unverzüglich abbremsen, um rechtzeitig vor diesem Signal stehen zu bleiben. Dieser Bremsweg muss bei der Simulation beachtet werden und wird im Attribut „reservedNodes“ nachgebildet. Alle dort enthaltenen Nodes sperren dementsprechend deren Parent-Section. Der Bremsweg gibt somit den zukünftigen Pfad des Zuges an und die Entscheidungen, welche Node der Zug als nächstes nimmt, müssen dort entschieden werden.

Sobald eine Section mit einer eingehenden Node frei wird, wird ein neuer Zug auf der Eingangs-Node erzeugt. Dies wird getan, um die Kreuzung mit einem kontinuierlichen Strom an Zügen zu versorgen und dadurch die Kreuzung zu stresstesten.

Wenn ein Zug einen Ausgang der Kreuzung erreicht hat und dieser den Ausgang vollständig durchfahren hat. Dann erhöht diese erfolgreiche Durchfahrt der Kreuzung einen Zähler.

Die Simulation wird für 30 Sekunden ausgeführt und der Zähler, welcher die erfolgreichen Zugfahrten zählt, wird auf eine Minute hochgerechnet. Dadurch ergibt sich ein Wert, der verwendet werden kann, um verschiedene Kreuzungen miteinander zu vergleichen.

Wie zuvor erwähnt worden ist, konnte dieser Ansatz in der Software aufgrund von fehlender Zeit nicht implementiert werden.

5.6. User-interface

Damit Nutzer mit der Anwendung interagieren und das Ergebnis einsehen können, wird eine Art von Benutzerschnittstelle benötigt. Im Folgenden werden zwei verschiedene Arten erläutert und erklärt, weshalb schlussendlich eine umgesetzt wurde und die andere nicht.

5.6.1. JavaFX-UI

Eine Möglichkeit die sinnvoll erscheint, ist die Umsetzung einer kompletten grafischen Benutzeroberfläche mit UI-Elementen. Da die Applikation in Java geschrieben ist und deshalb die entsprechenden Objekte bereits erstellt wurden, können diese mit JavaFX visuell dargestellt werden.²³ Dafür wird zunächst das Schienennetz gezeichnet. Dieses wird als Graph in einer zweidimensionalen Matrix gespeichert und kann in dieser Form auf dem Canvas abgebildet werden. Dafür wird durch diese iteriert und nach den Schienen gesucht. Diese werden daran erkannt, dass der Wert an dieser Stelle der Matrix nicht den Wert null hat. Auch Zugsignale werden in diesem Fall ignoriert, da nur der grobe Aufbau des Schienennetzes relevant ist, um darauf den Deadlock anzuzeigen. Nachdem eine Schiene in der Matrix gefunden wird, erfolgt die Platzierung auf dem Canvas durch eine mathematische Rechnung, da mit Koordinaten in dem JavaFX Fenster gerechnet werden. Dabei hat die linke obere Ecke die Koordinaten (0, 0) wobei die erste Koordinate die x-Achse und der zweite Wert die y-Achse repräsentiert.

```
double xPos = 2 + i * scalePos / 60;  
double yPos = 80 + j * scalePos / 80;
```

Abbildung 59: Koordinatenberechnung der Schiene

Der i-Wert bezeichnet die Zeile und der j-Wert die Spalte der Schiene in der Matrix. Die Werte davor, die dazu addiert werden, dienen als feste Abstände zu den Rändern des Fensters. Am Ende wird ein Skalierungswert multipliziert, damit die Anwendung responsive ist und der Nutzer das Fenster vergrößern oder verkleinern kann. Dieser passt die Koordinaten zum Verhältnis zur Größe des Fensters an. Nachdem die Koordinaten im Fenster feststehen, muss noch visuell dargestellt werden, dass hier eine Schiene ist. Der übersichtlichste Ansatz wäre hier ein Bild einer Schiene aus Factorio zu verwenden, dabei treten allerdings einige Schwierigkeiten auf, die später genauer erläutert werden. Zunächst wird als Ersatz der Buchstabe „O“ bei einer einzelnen Schiene und der Buchstabe „X“ für eine Kreuzung von Schienen verwendet. Alle gefundenen Schienen werden in einer neuen Liste gesammelt, um

²³ Vgl. „JavaFX“, JavaFX, zugegriffen 16. März 2024, <https://openjfx.io/localhost:1313/>.

nicht erneut die gesamte Matrix durchzugehen, die viele null Werte besitzt. Für eine bessere Lesbarkeit wird die Hintergrundfarbe des Fensters auf Schwarz und die Schriftfarbe auf Blau gesetzt.

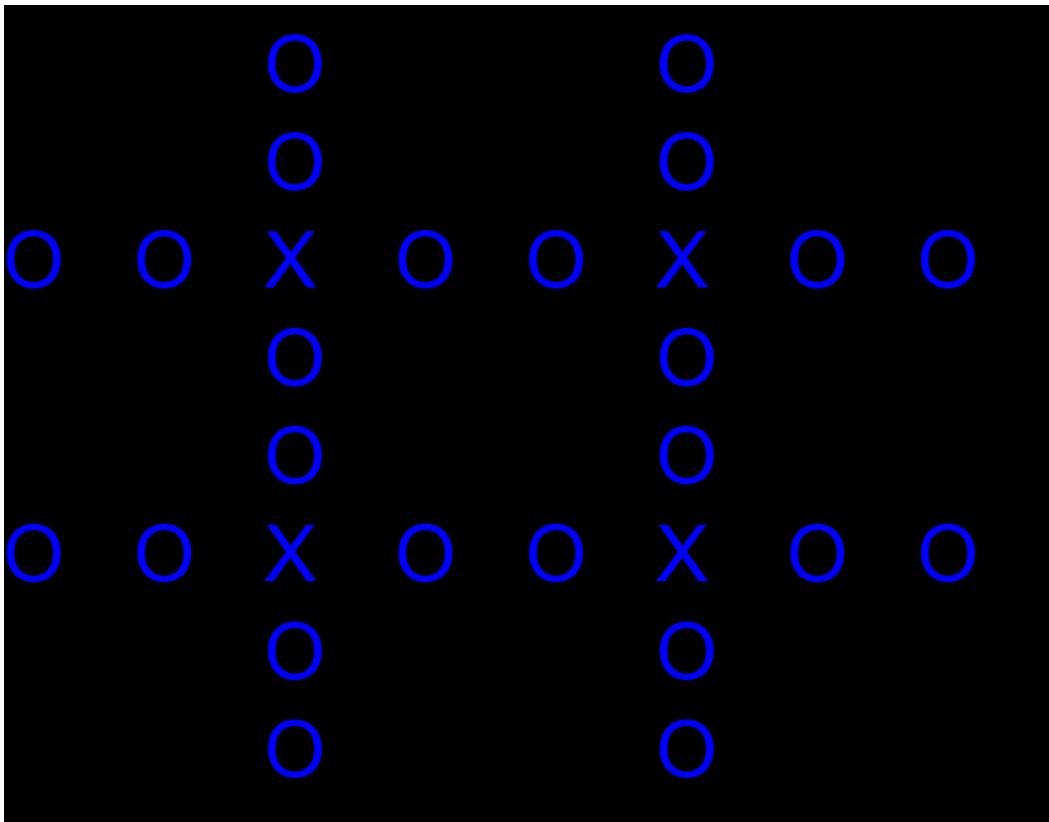


Abbildung 60: JavaFX-Darstellung einer Kreuzung

Nach dem Durchgang der Rekursion wird der Deadlock-Pfad gespeichert. In diesem befinden sich die relevanten Node-Objekte aus dem Graphen. Diese Objekte enthalten mehrere Schienen-Objekte, die in alle wie vorher beschrieben aus der Matrix rausgesucht und visuell dargestellt werden. Um einen Deadlock anzeigen zu können müssen also alle relevanten Schienen aus dem Deadlock-Pfad gefunden und in der UI visuell verändert werden. Damit der Graph aber leichter zu lesen ist, werden nicht die Buchstaben der Schienen visuell verändert, sondern lediglich der Pfad des Deadlocks angezeigt. Dafür werden die relevanten Schienen, die auch im Deadlock-Pfad vorkommen, rausgefiltert und mit einer roten Linie verbunden. So ergibt sich der Pfad, der durchgelaufen wird, um den Deadlock zu finden und der Nutzer kennt den genauen betroffenen Bereich. Da die einzelnen Schienen jeweils in einer Node gesammelt sind, sind nur die Schienen innerhalb derselben Node miteinander verbunden. Die Nodes an sich sind nicht verknüpft.

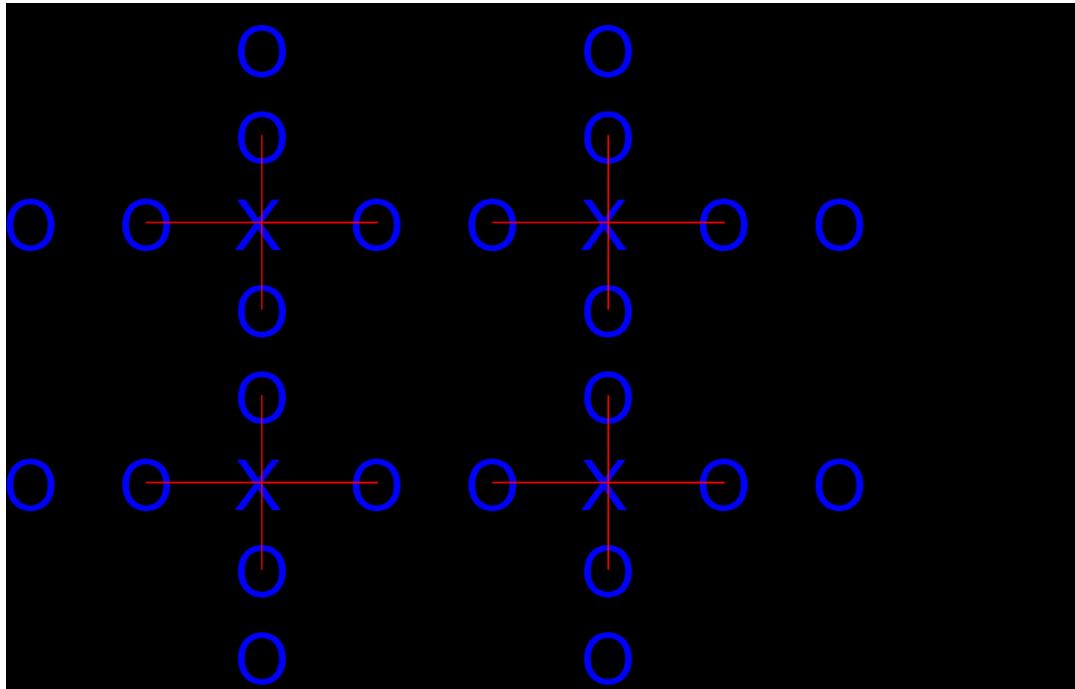


Abbildung 61: Deadlock-Darstellung in JavaFX

Diese Art der Darstellung ist für Debug-Zwecke bereits sehr nützlich, da schnell Blaupausen getestet und nachvollzogen werden können. Für den Endanwender ist es allerdings zu abstrakt, unvollständig dargestellt und nicht intuitiv nutzbar. Die Verwendung von Bildern an den entsprechenden Koordinaten ist also eine Möglichkeit das Schienennetz übersichtlicher und näher am Spiel darzustellen. Für das obige Beispiel mit ausschließlich linearen Nodes ist das auch problemlos umsetzbar. Sobald eine Node allerdings eine Kurve besitzt, funktioniert der simple Ansatz mit dem Einfügen eines Bildes an den Koordinaten allerdings nicht mehr. Eine Kurve in Factorio besteht nicht aus einer einzelnen Schiene, die gebogen ist. Im Verlauf der Kurve gibt es mehrere Schienen-Objekte, die teilweise nicht in Bogenform angeordnet sind. Dennoch könnte dieses Problem behoben werden, indem beispielsweise die möglichen Positionen der Schienen innerhalb der Kurve ermittelt werden und so die Länge und Form bestimmt werden kann. Anschließend kann ein entsprechendes Bild der Kurve in der Mitte der Krümmung platziert werden. Da es eine endliche Anzahl an Möglichkeiten für Kurven in Factorio gibt, wäre dies umsetzbar. Allerdings konnte aus zeitlichen Gründen und aufgrund von anderen Anforderungen dieser Ansatz nicht vollendet werden.

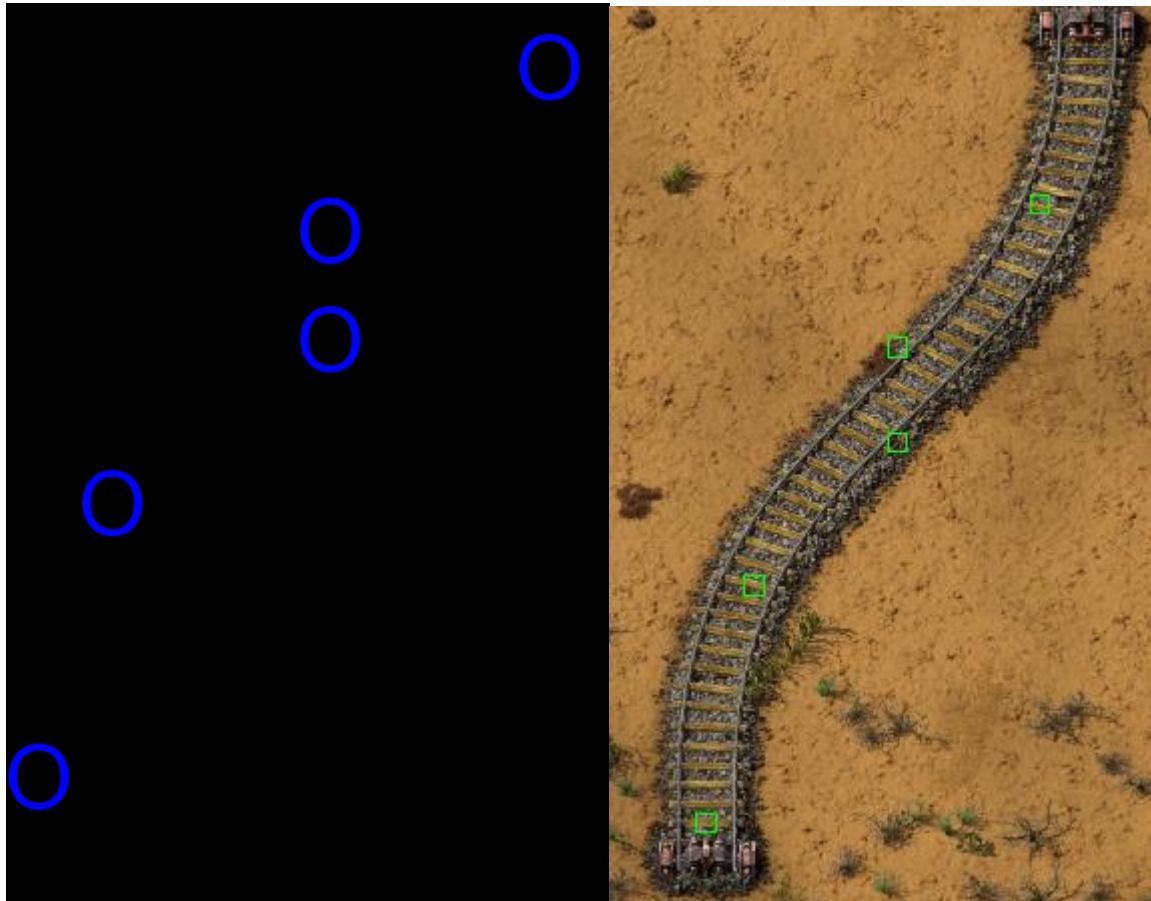


Abbildung 62: JavaFX-Darstellung einer Kurve

5.6.2. Command Line Interface (CLI)

Der zweite Ansatz für eine Benutzerschnittstelle wird umgesetzt. Die Applikation muss mit Gradle gebaut werden, sodass eine ausführbare jar-Datei entsteht. Dieser muss beim Ausführen eine Textdatei mit der Blueprint mitgegeben werden. Dafür wird der Dateipfad verwendet. Anschließend wird eine json-Datei als Ergebnis erstellt. Für das Erstellen des Json-Inhalts wird Open Source Library „Gson“ verwendet.²⁴ Damit ist es möglich, Java-Objekte in Json-Format umzuwandeln. Es wird ein Objekt der Klasse Gson erzeugt, welches die Methode „toJson“ ausführen kann. Dieser Methode wird ein Objekt der Klasse Output mitgegeben, welches automatisch durch einen Java-FileWriter in ein neues Textdokument in Json-Form geschrieben wird. Die Klasse Output wird extra hierfür definiert. Es enthält alle Deadlock-Pfade aus dem Result-Objekt der Analyse sowie den Graphen des gesamten Schienennetzes. Die Gson-Library geht beim Schreiben des Json-Strings durch alle Klassen und ihre Attribute. Beispielsweise wird beim Attribut Graph der Klasse Output als nächstes in die Klasse Graph

²⁴Vgl. „google/gson“, Java (2015; repr., Google, 16. März 2024), <https://github.com/google/gson>.

gewechselt. Hier werden ebenfalls alle Attribute durchgegangen und so geht es immer weiter bis nur noch Werte bleiben. Dadurch können aber auch Loops entstehen. Beispielsweise hat die Klasse Node das Attribut Next-Nodes, in dem alle Nachbar-Nodes enthalten sind. Bei einem bidirektionalen Graphen hat diese Next-Node aber in ihrem Attribut der Next-Nodes wieder einen Verweis auf die Node aus der gerade gewechselt wurde. Dadurch springt der Algorithmus unendlich oft hin und her und das Erstellen der Json endet mit einem StackOverFlow-Error. Deswegen und um unnötige Informationen zu vermeiden, werden einige Attribute mit dem Modifier „transient“ ausgeschlossen. Die Gson-Library ignoriert also bei ihrem Durchlauf des Klassenbaums alle Attribute mit diesem Modifier. Schlussendlich ist das Ziel, dass die Json eventuell für eine detailliertere Darstellung genutzt werden kann, weshalb nur die Schienen mit ihrer Länge, Richtung und Position wichtig sind. Außerdem besitzt jede Schiene eine eigene ID sowie die Information ob es eine Input- oder Output-Schiene ist, um sie voneinander zu unterscheiden. Die Auflistung der Schienen wird jeweils für das gesamte Schienennetz ausgegeben und für den Deadlock-Pfad. Sollte keine Textdatei mit einer Blaupause der jar-Datei mitgegeben werden, wird eine Warnung im Terminal ausgegeben und es wird keine Analyse ausgeführt oder Output-File erstellt. Durch diese Art der Ausgabe kann die Applikation in weiteren Anwendungen eingebaut und die Daten der json-Datei dort weiterverwendet werden. Die Funktion als mögliche Schnittstelle, anstelle von einer eigenständigen Software mit lediglich einer visuellen Darstellung des Deadlocks ist für die Modding-Community des Spiels zugänglicher, da so in der Theorie eine große Mod erstellt werden kann, die verschiedenste Dienste nutzt und diese zusammenbringt, anstatt das jede Mod ihre eigene Benutzeroberfläche nutzt.

Außerdem gibt es die Möglichkeit mithilfe eines Feature Flags die Software im Debug-Modus laufen zu lassen. Dabei wird eine Beispiel-Blaupause, die sich im Code befindet und verändert werden kann, mit einem Deadlock bereitgestellt, die automatisch für den Algorithmus verwendet wird.

Zum Schluss folgt ein Fazit bezüglich der Implementierung der ursprünglichen Aufgabenstellung und möglicher weiterer Schritte.

6. Fazit

In dieser Studienarbeit ist das Factorio Zugmodell bezüglich der Deadlock-Freiheit und Performance analysiert worden. Es sind zuerst theoretische Konzepte vorgestellt worden, die für das Verständnis dieser Arbeit benötigt werden.

Daran anschließend ist das zugrundeliegende Problem beim Design einer Zugkreuzung in Factorio erläutert worden. Ein Spieler muss ein Design finden, dass sowohl performant arbeitet als auch frei von Deadlocks ist. Um diesen Prozess zu automatisieren, sind Anforderungen für ein Tool definiert worden. Auch sind verschiedene Ansätze zur Erfüllung dieser Anforderungen vorgestellt worden.

Das Kernstück dieser Arbeit war die Implementierung von geeigneten Ansätzen, um den Anforderungen an das Tool gerecht zu werden. Anfangs ist ein Algorithmus vorgestellt worden, der aus einer Blaupause eine für die Analysen geeignete Datenstruktur, welche als Graph bezeichnet worden ist, erstellt. Die Beschreibung des Algorithmus zur Deadlock-Erkennung, welcher auf der Suche nach einem zyklischen Wartegraphen basiert, erfolgte im Anschluss daran. Auch ist ausführlich ein Ansatz für die Performanceanalyse erläutert worden. Dieser ist allerdings aufgrund von Zeitmangel nicht umgesetzt worden. Zum Schluss ist dargelegt worden, wie ein Nutzer mit diesem Tool interagiert. Hierbei ist entschieden worden, dass ein CLI, welche eine txt-Datei als Eingabe und eine json-Datei als Ausgabe der Analyseergebnisse nutzt, verwendet wird.

Eine Weiterführung dieser Arbeit würde bedeuten, dass sowohl der Ansatz der Performanceanalyse als auch eine grafische Benutzeroberfläche implementiert wird. Auch ist eine Analyse, inwiefern der vom Tool generierte Graph für andere Anwendungsfälle verwendet werden kann, relevant. Das gesamte Projekt kann unter <https://github.com/alimasigna/factorio-train-analyser> aufgerufen werden.

7. Literaturverzeichnis

Academic dictionaries and encyclopedias. „Vogel-Strauß-Algorithmus“. Zugegriffen 11. Februar 2024. <https://de-academic.com/dic.nsf/dewiki/1471161>.

„betriebssysteme-11-23-05-25-Wdh.pdf“. Zugegriffen 11. Februar 2024.
https://moodle.dhbw-mannheim.de/pluginfile.php/535400/mod_resource/content/1/betriebssysteme-11-23-05-25-Wdh.pdf.

„Configuring Deadlock Detection and Lock Wait Timeouts“. Concept. Zugegriffen 11. Februar 2024. <https://db.apache.org/derby/docs/10.9/devguide/cdevconcepts16400.html>.

Datta, Subham. „Deadlock: What It Is, How to Detect, Handle and Prevent? | Baeldung on Computer Science“, 20. Juni 2021. <https://www.baeldung.com/cs/os-deadlock>.

„Deadlock v/s Starvation - Coding Ninjas“. Zugegriffen 11. Februar 2024.
<https://www.codingninjas.com/studio/library/deadlock-vs-starvation>.

„Eisenbahn - Factorio Wiki“. Zugegriffen 21. Februar 2024.
<https://wiki.factorio.com/Railway/de>.

Factorio. „Factorio“. Zugegriffen 12. Februar 2024. <https://www.factorio.com/>.

„Factorio - Steam Charts“. Zugegriffen 15. Februar 2024.
<https://steamcharts.com/app/427520#7d>.

„Factorio on Steam“. Zugegriffen 12. Februar 2024.
<https://store.steampowered.com/app/427520/Factorio/>.

„Fließband-Transportsystem - Factorio Wiki“. Zugegriffen 21. Februar 2024.
https://wiki.factorio.com/Belt_transport_system/de.

„Fuel - Factorio Wiki“. Zugegriffen 14. März 2024. <https://wiki.factorio.com/Fuel>.

GeeksforGeeks. „Wait For Graph Deadlock Detection in Distributed System“, 25. April 2022.
<https://www.geeksforgeeks.org/wait-for-graph-deadlock-detection-in-distributed-system/>.

„google/gson“. Java. 2015. Reprint, Google, 16. März 2024. <https://github.com/google/gson>.

„Gradle User Manual“. Zugegriffen 28. Februar 2024.
<https://docs.gradle.org/current/userguide/userguide.html>.

JavaFX. „JavaFX“. Zugegriffen 16. März 2024. <https://openjfx.io/localhost:1313/>.

Official Factorio Wiki. „Blaupause - Factorio Wiki“. Zugegriffen 21. Februar 2024.
<https://wiki.factorio.com/Blueprint/de>.

Official Factorio Wiki. „Zug-Kettensignal - Factorio Wiki“. Zugegriffen 21. Februar 2024.
https://wiki.factorio.com/Rail_chain_signal/de.

Official Factorio Wiki. „Zugsignal - Factorio Wiki“. Zugegriffen 21. Februar 2024.
https://wiki.factorio.com/Rail_signal/de.

„Referat ‚Deadlocks‘ im Proseminar ‚Computer Science Unplugged‘“, o. J.

„Resource Allocation Graph | Deadlock Detection | Gate Vidyalay“. Zugegriffen 11. Februar 2024. <https://www.gatevidyalay.com/resource-allocation-graph-deadlock-detection/>.

8. Abbildungsverzeichnis

- Abbildung 1: Ablade-Station in Factorio, in: „Eisenbahn - Factorio Wiki“
- Abbildung 2: Streckenabschnitte in Factorio
- Abbildung 3: Belegter Streckenabschnitt
- Abbildung 4: Ein Zug-Kettensignal bei einem versperrten Ausgang
- Abbildung 5: Miteinander verkettete Kettensignale
- Abbildung 6: Ein unsicherer Kreuzungsabschnitt, in: „Zug-Kettensignal - Factorio Wiki“
- Abbildung 7: Das „Dining-Philosopher“ Problem, in: „Referat ‚Deadlocks‘ im Proseminar, Computer Science Unplugged“
- Abbildung 8: Ein Ressourcenzuweisungsgraph, in: „Wait For Graph Deadlock Detection in Distributed System“
- Abbildung 9: Ein Wartegraph, in: „Wait For Graph Deadlock Detection in Distributed System“
- Abbildung 10: Timeout Ansatz zur Deadlock-Erkennung, in: „Configuring Deadlock Detection and Lock Wait Timeouts“
- Abbildung 11: Deadlock im Factorio Zugsystem
- Abbildung 12: Deadlockfreie aber inperformante Zugkreuzung
- Abbildung 13: Performantere aber nicht deadlockfreie Kreuzung
- Abbildung 14: Zug kollidiert mit sich selbst trotz Signale
- Abbildung 15: Ordnerstruktur der Applikation
- Abbildung 16: Alle Track-Varianten in Factorio
- Abbildung 17: Nodes im Zugstrecken-Modell
- Abbildung 18: UML-Klassendiagramm der „Matrix.java“
- Abbildung 19: Aufbau des „factorio.train.analyser.graph“ Package
- Abbildung 20: UML-Diagramm der Track Klasse
- Abbildung 21: Tracks mit gesetztem „isEndTrack“ Flag
- Abbildung 22: Beispiel für die „goesTo“ und „protectedFrom“ Attribute
- Abbildung 23: UML-Diagramm der Node Klasse
- Abbildung 24: Beispiel Aufteilung der „nextNodes“ und „protectedFrom“ Attribute
- Abbildung 25: UML-Diagramm der Section Klasse
- Abbildung 26: UML-Diagramm der LookUp Klasse

- Abbildung 27: Auszug der „lookUpConnected()“ Methode
- Abbildung 28: Anzeige der Track Positionen im Debug-Modus
- Abbildung 29: Auszug an Kombinationen von sich überschneidenden Tracks
- Abbildung 30: UML-Diagramm der Graph Klasse
- Abbildung 31: Konstruktor der Klasse Graph
- Abbildung 32: Die setSections() Methode
- Abbildung 33: Zusammengehörige Nodes nach einem Parsedurchlauf
- Abbildung 34: Vereinfachte Darstellung verbundener Tracks
- Abbildung 35: Rekursiver Aufruf der Frontier-Tracks
- Abbildung 36: Entleeren der callback und frontier Arraylisten
- Abbildung 37: Sectionszugehörigkeit durch verschiedene Platzierungen auf demselben Track
- Abbildung 38: Abstandsvektoren ausgehend vom „currentTrack“
- Abbildung 39: Abstandsvektoren ausgehend von einem kurvigen „currentTrack“
- Abbildung 40: Diagonale Tracks mit Signalen
- Abbildung 41: Kombination von kurvigen Tracks mit diagonalen Tracks
- Abbildung 42: Rekursive Ausführung bis T5
- Abbildung 43: Rekursiver Aufruf der Callback Tracks von T5
- Abbildung 44: Inhalt des „sections“ Attribut nach dem Durchlaufen der Schleife
- Abbildung 45: Logik zur Erkennung einer eingehenden bzw. ausgehenden Node
- Abbildung 46: Beispiel Deadlock in Factorio
- Abbildung 47: Die „deadlockDetection()“ Methode
- Abbildung 48: Überprüfung auf besondere Fälle
- Abbildung 49: Erster Sonderfall einer geschützten Node im Rekursionsaufruf
- Abbildung 50: Zweiter Sonderfall einer geschützten Node im Rekursionsaufruf
- Abbildung 51: Dritter Sonderfall einer geschützten Node im Rekursionsaufruf
- Abbildung 52: Beispiel-Kreuzung für den Rekursions-Algorithmus mit markierten Sektionen
- Abbildung 53: Beginn des Rekursion-Beispiels
- Abbildung 54: Schritt 2 des Rekursions-Beispiels
- Abbildung 55: Sonderfall durch eine Output-Node im Rekursions-Beispiel
- Abbildung 56: Neuer Weg im Rekursions-Beispiel
- Abbildung 57: Gefundener Deadlock im Rekursions-Beispiel

Abbildung 58: Zug beim Verlassen einer bidirektionalen Node

Abbildung 59: Koordinatenberechnung der Schiene

Abbildung 60: JavaFX-Darstellung einer Kreuzung

Abbildung 61: Deadlock-Darstellung in JavaFX

Abbildung 62: JavaFX-Darstellung einer Kurve