Abstrakt

In dem Computerspiel Factorio werden Züge auf Schienen benutzt, um Ressourcen zu transportieren und damit Produkte herzustellen. Bei Schienen-Kreuzungen können Deadlocks vorkommen, die den Transport behindern. Automatisierte Tests können diese Deadlocks aufdecken, bevor es zu Problemen kommt.

Thema Studienarbeit

Automatische Deadlockerkennung im
Schienennetz des Spiels Factorio

im Studiengang

an der Duale Hochschule Baden-Württemberg Mannheim

von

Name, Vorname: Schnüll, Leo und Stella, Sander

Abgabedatum: 16.04.2024

Bearbeitungszeitraum: 17.10.2023 – 16.04.2024

Matrikelnummer, Kurs: 7746976, 1325157, TINF21AI1

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

In dem Spiel Factorio baut der Spieler Schienensysteme, um Ressourcen mit Zügen zu transportieren und damit eine Fabrik aufzubauen und zu beliefern. In diesem Schienensystem kann es zu Problemen kommen, bei denen die Züge nicht mehr weiterfahren können, da sie sich gegenseitig blockieren. Dies könnte man in jedem Einzelfall manuell beseitigen, das macht aber viel Arbeit und keinen Spaß. Komfortabler wäre es, das Schienensystem automatisch zu analysieren, um mögliche Blockaden zu finden, bevor sie auftreten. Je komplizierter die Schienensysteme sind, desto fehleranfälliger sind sie logischerweise. Ab gewisser Komplexität wird es für einen Menschen schwierig, sicherzustellen, dass keine Probleme auftreten werden. Als Lösung soll ein Programm diese Probleme finden oder bestimmt sagen, dass sie nicht existieren.

Das Ziel dieser Arbeit ist es, so eine automatische Blockadeerkennung zu entwickeln.

1.1 Motivation

Das Planen von Schienensystemen kann auch für erfahrende Factorio Spieler schwierig werden. Dazu kommt, dass in einigen Fällen von der Commmunity erstellte Standardbibliotheken für Schienensysteme wegen der Umgebung nicht genutzt werden können, sondern es eigenhändig geplant werden muss. In diesem Fall ist es wichtig zu wissen, ob die Kreuzung, die gerade gebaut wurde, in der Zukunft Probleme machen könnte, so dass z.B. sogenannte Deadlocks entstehen. Aus diesem Grund wird dieses Pilotprojekt entwickelt, um Spielern die Möglichkeit zu geben, die Kreuzung auf solche Probleme hin zu überprüfen.

1.2 Forschungsfrage

Als konkrete Frage für diese Arbeit wird formuliert:

Wie können in dem Spiel Factorio Deadlocks in Schienennetzen automatisch gefunden werden?

Am Ende dieser Arbeit wird diese Frage beantwortet.

1.3 Vorgehensweise

Zuerst wird erläutert, wie genau das Zugsystem von Factorio und dessen Bestandteile funktionieren. Es wird darauf eingegangenen, was ein Deadlock ist und wie dieser vorkommen kann. Als nächstes werden Möglichkeiten der Analyse auf Deadlocks vorgestellt. Es wird eine Entscheidung begründet, welche Methodik in dieser Arbeit weiter ausgeführt wird. Danach wird ein Pilotprojekt vorgestellt, in dem die Methodik umgesetzt wurde. Es wird ausgewertet, wie gut das Pilotprojekt Deadlocks erkennen kann. Als letztes wird die Forschungsfrage im Fazit beantwortet.

2. Theorieteil

2.1 Was ist Factorio?

“Factorio is a game in which you build and maintain factories.” (factorio.com, 01.04.2024)

In Factorio baut der Spieler eine Fabrik auf. Das übergeordnete Ziel ist es, mithilfe dieser Fabrik eine Rakete zu bauen und zu starten. Dazu müssen Ressourcen gesammelt werden und diese zu bestimmten Endprodukten weiterverarbeitet werden. Dabei wächst die Fabrik stetig mit der benötigten Anzahl an Endprodukten, so dass ständig neue Ressourcen gesammelt werden müssen. Dieses Wachstum der Fabrik und die damit in Verbindung stehenden Herausforderungen sind das Kernelement des Spiels. Es müssen Ressourcensammlung, Transport und Verarbeitung geplant und durchgeführt werden. Je größer die Fabrik wird, desto schwieriger wird es, die Ressourcen von der Stelle, wo sie abgebaut werden, zu der Stelle, wo sie weiterverarbeitet werden zu transportieren. Für den Transport der Ressourcen stehen dem Spieler verschiedene Optionen zur Verfügung. Die einfachste und langsamste Form sind Greifarme, die die Ressourcen bewegen. Als

Verbesserung davon gibt es Fließbänder, mit denen die Ressourcen besser transportiert werden können. Als schnellste Option kann der Spieler Züge nutzen, um die Ressourcen über lange Strecken zu transportieren.

In den folgenden Abschnitten wird genauer auf die Bestandteile des Factorio Zug Systems eingegangen, um im Detail darzustellen, was einem Spieler möglich ist und so zu verdeutlichen, welche Probleme dabei entstehen können

2.1.1 Übersicht der Bestandteile des Zugsystems

Es gibt vier verschiedene Bestandteile, mit denen ein Netz gebaut werden kann: Schienen, Signale, Kettensignale und Züge.
Dabei definieren die Schienen die Pfade, auf denen sich Züge bewegen können. Um die Fahrtrichtung der Züge zu bestimmen und zu verhindern, dass Züge eventuell ineinander fahren, gibt es die 2 Signalarten. Diese unterteilen die existierenden Schienen in sogenannte Blöcke.

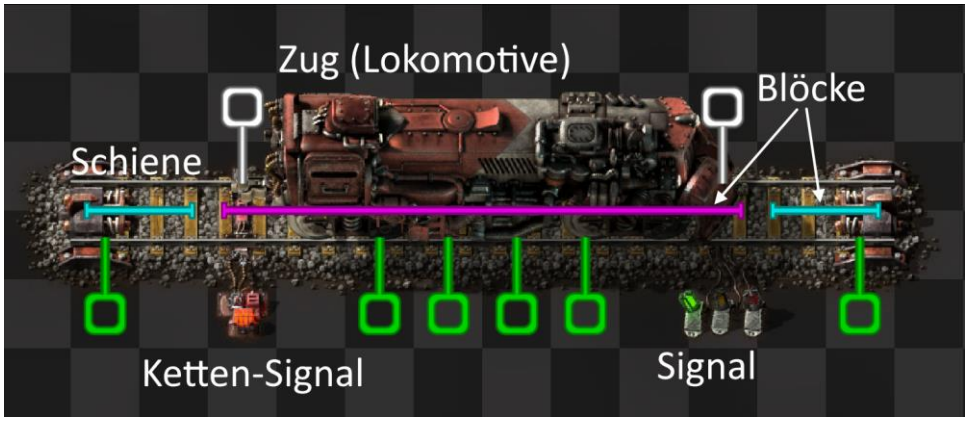

Abbildung 1: Bestandteile des Factorio Zugsystems

Nachfolgendend gilt: Sofern die Art des Signals egal ist, wird dies Zugsignal genannt. Mit Signal bzw. Kettensignal ist dann explizit nur das jeweilige Objekt gemeint, wie dies in der Abbildung 1 gezeigt wird.

Züge können vom Spieler gefahren werden. Alternativ können ihnen Ziele gegeben werden, zu denen sie dann automatisch fahren, geleitet von den Signalen.

Factorio ist in ein Raster aufgeteilt, in dem sich alle Objekte, welche platziert werden, existieren. Dies ist im Bild angedeutet mit den grau schwarzen Kästchen im

Hintergrund der Abbildung 1. Jedes Kästchen hat eine X- und Y- Koordinate, hierbei ist (0,0), wie für Spiele typisch, oben links und wächst nach rechts unten positiv.

2.1.2 Entities (auf Deutsch Entitäten)

Jegliche Spielobjekte, mit denen der Spieler interagieren kann, sind Entities. Diese Entities speichern Informationen über sich selbst. Als erstes eine eindeutige Id, um die Entities vom selben Typ unterscheiden zu können. Danach den Typ von sich selbst und an welcher Position sie stehen und welche Rotation sie besitzen. Die Positionen von den Entities sind dabei, wo ihre Zentren auf dem Raster liegen. Die Rasterlinien sind ganze Zahlen. Wenn das Zentrum nicht auf einer Linie liegt, wie zum Beispiel bei Entitäten, die eine Größe von nur einem Raster haben, wird die Position mit 0,5 angegeben. Es werden pro Typ noch weitere unterschiedliche Informationen gespeichert, diese sind für das Schienensystem aber nicht relevant und werden deshalb hier nicht erläutert.

Es gibt zwei unterschiedliche Typen von Schienen, gerade und gebogene Schienen. Diese beiden werden in den beiden nachfolgenden Abschnitten genauer erläutert.

2.1.3 Gerade Schienen

Gerade Schienen aller Art sind ein 2*2 Raster großes Objekt. Sie können senkrecht, waagerecht oder diagonal verlaufen. Damit die Schienen, egal welcher Art, immer ineinander übergehen, dürfen sie nur an Positionen mit ungeraden Koordinaten gesetzt werden. Dann sind sie auf gleicher Höhe bzw. Breite.

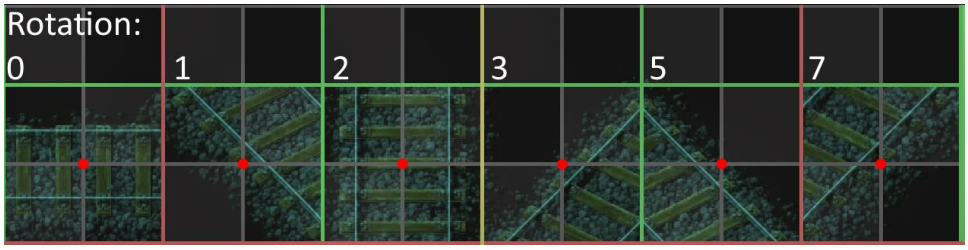

Abbildung 2: Alle geraden schienen nach Rotation sortiert und mit Ankerpunkt markiert

In der Abbildung 2 sieht man die unterschiedlichen Anordnungsmöglichkeiten von geraden Schienen. Die Nummerierung entspricht einer Rotation nach rechts., Es gibt keine Rotation 4 und 6, da sie 0 und 2 entsprechen. Beim Bau der Schienen können abwechseln 3 und 7 zu einer Strecke verbunden werden, ebenso 5 und 1. 0 und 2 können nur mit sich selber oder gebogenen Schienen verbunden werden.

An Schienen können Zugsignale gesetzt werden (die Bedeutung wird in 2.1.9 erläutert). Je nach Rotation sind dafür unterschiedliche Positionen möglich. Schienen mit den Rotationen 2 und 0 haben vier Positionen, an denen Zugsignale gesetzt werden können und die Schienen mit den restlichen Rotationen haben zwei Positionen für Zugsignale.

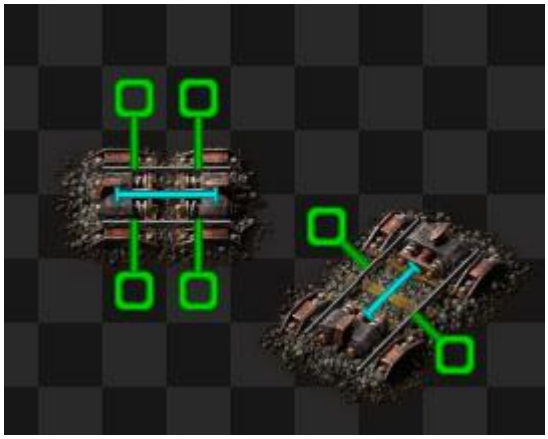

Abbildung 3: Schienen der Rotation 0 und 3 mit deren möglichen Signalpositionen

Die grünen Kästchen im Bild symbolisieren die Positionen, an denen Zugsignale gesetzt werden können. Allerdings können an den Schienen der Rotation 2 und 0 tatsächlich nur 2 Signale gleichzeitig platziert werden, obwohl es 4 mögliche Positionen gibt. Das kommt daher, dass Zugsignale nicht direkt nebeneinander stehen dürfen.

2.1.4 Gebogene Schienen

Gebogene Schienen sind 4*8 Felder große Objekte mit dem Ankerpunkt jeweils im Zentrum des Rechtecks. Gebogene Schienen haben auch Rotationswerte von 0 bis 7.

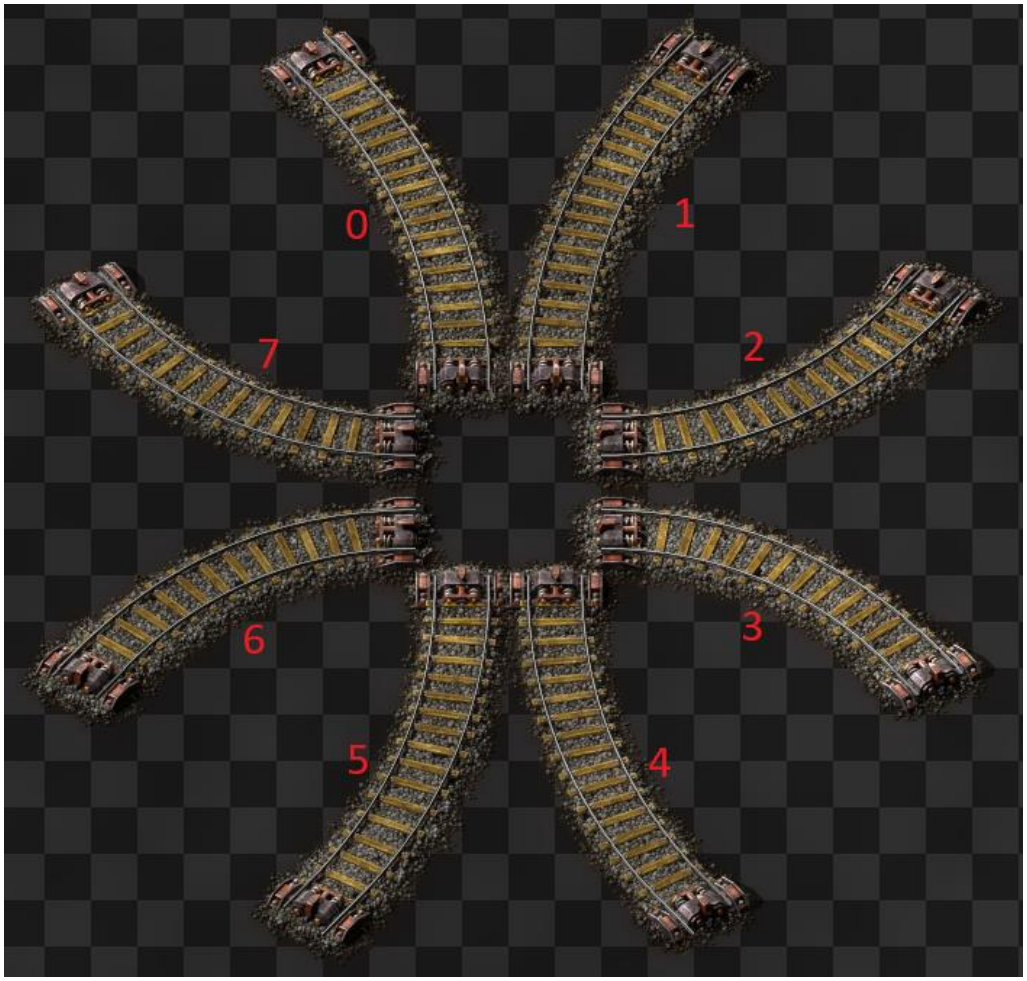

Abbildung 4: Gebogene Schienen mit Rotationswerten

Die Enden der gebogenen Schienen in der Mitte des Bildes können sich mit den anderen gebogenen Schienen direkt verbinden. Bei den Enden am Rande des Bildes muss eine diagonale Schiene als Verbindungsstück platziert werden. Dadurch kann man gebogene Schienen zusammensetzen, um eine beliebige Kurvenrichtung zu erreichen. Eine 90° Kurve kann zum Beispiel erreicht werden, indem gebogene Schienen mit Richtung 0 und 3 zusammengesetzt werden, mit einer Diagonalen Schiene der Richtung 1 dazwischen.

Bei gebogenen Schienen können 4 Zugsignale platziert werden, jeweils 2 an einem Ende.

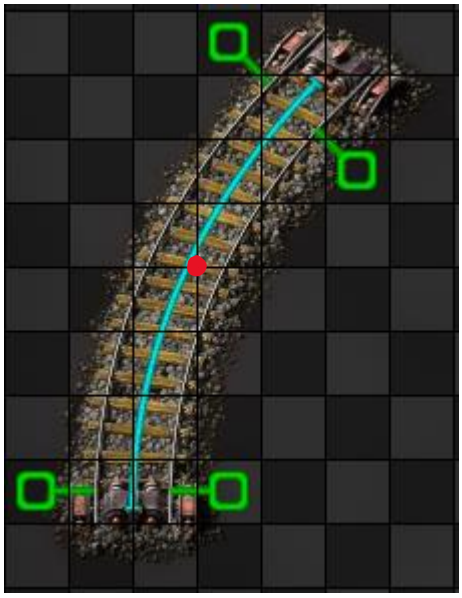

Abbildung 5: Gebogene Schiene mit Ankerpunkt in rot und Zugsignalpositionen in grün

An gebogene Schienen können 4 Zugsignale gleichzeitig platziert werden, weil die Positionen weit genug voneinander entfernt sind.

2.1.5 Kreuzung/Abzweigung/Schienennetz

Dieses Kapitel dient zur Klärung von den Worten Kreuzung und Abzweigung und was der Unterschied zwischen den Beiden ist.

Es ist möglich, mehrere gerade Schienen auf dieselbe Stelle zu platzieren. Dadurch bilden sich Kreuzungen. Hier fahren Züge aus unterschiedlichen Richtungen über dieselbe Stelle. Dies kann in unterschiedlichen Winkeln erfolgen, hier ein Beispiel einer orthogonalen Kreuzung:

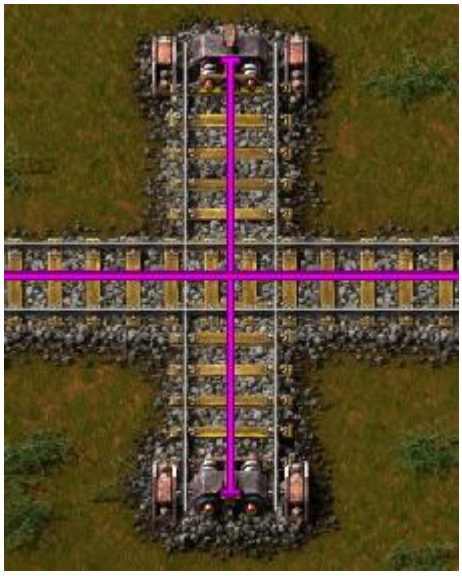

Abbildung 6: Kreuzung aus Schienen der Rotation 0 und 2

Eine Variante der Kreuzung ist die Abzweigung. In dem Fall kommen die Schienen aus derselben Richtung, gehen aber in zwei unterschiedliche Richtungen weiter. Das ist der Fall, wenn gebogene Schienen sich mit anderen Schienen überschneiden.

Abbildung 7: Abzweigung aus gebogener und gerader Schiene

Hier im Beispiel überschneidet eine gebogene Schiene eine gerade Schiene.

Ein Schienennetz ist die gesamte Ansammlung an zusammenhängenden Schienen. In einem Schienennetz kommen meistens Kreuzungen und Abzweigungen vor. Ein Schienennetz besitzt Eingänge und Ausgänge. Ein Streckenstart ohne ein in Fahrtrichtung vorheriges Streckenende ist ein Eingang in das Schienennetz. Ein Streckenende ohne einen in Fahrtrichtung nachfolgenden Streckenstart ist ein Ausgang aus dem Schienennetz. Ein Schienennetz kann ein Ausschnitt aus einem größeren Schienennetz sein. Dementsprechend gelten die beiden Beispiele in diesem Kapitel schon als Schienennetz.

2.1.6 Strecke

Das Verhalten von den Zugsignalen wird in dem Spiel dazu genutzt, die Schienen in Abschnitte einzuteilen. Diese Abschnitte werden Strecken genannt. Eine Strecke startet und endet mit einem Zugsignal, dazwischen liegen zusammenhängende Schienen. Eine Strecke ist mindestens eine Schiene lang. Falls es kein endendes Zugsignal gibt, endet die Strecke am Ende der Schienen. Gibt es kein startendes Zugsignal, so beginnt die Strecke am Anfang der Schienen. Bei Abzweigungen wird von zwei unterschiedlichen Strecken gesprochen, da sie zwei unterschiedliche End- bzw. Startzugsignale besitzen.

Eine Aneinanderreihung von Strecken wird Gesamtstrecke genannt. Eine Gesamtstrecke führt durch das gesamte Schienennetz, vom Eingang bis zum Ausgang.

2.1.7 Block

Ein Block ist eine Ansammlung an Strecken. Dabei gehören Strecken zum selben Block, wenn sie ineinander übergehen oder sich überschneiden. Für die Definition eines Blocks sind keine Zugsignale notwendig. Wenn es kein Signal an irgendeiner Schiene gibt, zählen alle zusammenhängenden Schienen zu einem Block.

Bei Kreuzungen gehören die sich kreuzenden Schienen zu unterschiedlichen Strecken, aber zu demselben Block. Bei Abzweigungen gilt genauso, dass es zwei unterschiedliche Strecken sind, aber derselbe Block.

2.1.8 Zuordnung von Signalen zu Strecken

Beide Formen der Zugsignale kommen in 8 Varianten jeweils für jede Rotation vor. Im Gegensatz zu Schienen kann hier zwischen den horizontalen bzw. vertikalen Varianten unterschieden werden.

In den vergangenen Abschnitten wurde bereits besprochen, wo es möglich ist, Signale zu platzieren. Dabei ist noch zu beachten, dass Signale nur dann platziert

werden können, sofern nicht ein anderes Signal direkt nebenan liegt. Außerdem darf keine andere Schiene oder ein weiteres Objekt im Weg sein.

Jedes Zugsignal unterteilt wie im Kapitel 2.1.6 besprochen die Schienen in Strecken. Dabei ist die Position des Zugsignals an der Schiene wichtig. Sie bestimmt, wo die Strecke endet bzw. startet.

Abbildung 8: Unterschiedliche Streckenlängen

In der Abbildung sind zwei fast gleiche Schienennetze zu sehen. Sie unterscheiden sich nur anhand der Position des Zugsignals. Im oberen Beispiel ist das Signal auf der linken möglichen Position platziert und im unteren Beispiel auf der rechten möglichen Position. Dies ändert, zu welcher Strecke die Schiene gehört. Im oberen Beispiel endet die linke Strecke vor der rot markierten Schiene, diese ist somit Teil der rechten Strecke. Im unteren Beispiel endet die linke Strecke nach der rot markieren Schiene, sie ist somit Teil der linken Strecke. Es ist wichtig, diese Verhaltensweise korrekt nachzubilden, damit eine Analyse korrekte Ergebnisse liefern kann.

2.1.9 Verhalten von Zugsignalen

Um nun die Züge zu leiten, beobachten die Zugsignale immer den nächsten Block und sofern sich ein Zug in diesem befindet, schalten sie selbst von grün auf rot, welches verhindert, dass ein weiterer Zug in den nächsten Block einfährt. Ein Kettensignal schaltet zusätzlich auch auf rot, wenn das in Fahrtrichtung nächste Zugsignal rot ist. Dies kann genutzt werden, um einen Zug an einem früheren Ort warten zu lassen, sodass dieser nicht einen anderen Zug blockiert.

Kettensignale beobachten an Abzweigungen, also dort wo sich eine Schiene in zwei oder mehr Schienen trennt, alle nachfolgenden Zugsignale. Das Kettensignal ist grün bzw. rot genau dann, wenn alle nachfolgenden Zugsignale grün bzw. rot sind. Wenn eine Mischung vorliegt, wird es blau. Genauso können Züge weiterfahren, deren Strecke ein grünes Zugsignal hat. In diesem Fall lässt das Kettensignal Züge durchfahren, die die Strecke mit dem grünen Zugsignal fahren und stoppt die Züge, die die Strecke mit dem roten Zugsignal fahren.

Im Gegensatz dazu stoppen Signale alle Züge bei Abzweigungen, auch wenn nur eine Schiene belegt ist und die Strecke für den durchfahrenden Zug frei wäre. Das ist der Fall, da es den Status blau in diesem Fall nicht gibt, so dass das Signal immer ein eindeutiges rot oder grün anzeigt.

Eine Schiene hat mehrere Stellen, wo ein Zugsignal platziert werden kann. Dabei bestimmt die Seite, heißt entweder links oder rechts neben der Schiene, auf welcher das Zugsignal platziert wird, in welche Richtung der Zug fahren kann. Wird zum Beispiel bei einer senkrechten Schiene das Zugsignal rechts von der Schiene platziert, können Züge nur nach oben fahren. Andersherum wenn es links platziert wird, können sie nur nach unten fahren. Solange nur ein Zugsignal neben der Schiene platziert ist, können die Züge nur in die vom Zugsignal bestimmte Richtung fahren. Ein Zug kann nicht losfahren, wenn die Fahrtrichtung falsch ist. Um es nun wieder möglich zu machen, dass Züge in beide Richtungen fahren können, müssen auf beiden Seiten der Schiene Zugsignale platziert werden. Dabei müssen zwei Zugsignale immer genau gegenüber voneinander sein, eine andere Platzierung verbietet Factorio.

Trotzdem kann es zu Fällen kommen, dass keinem Zug mehr erlaubt ist, die Strecke zu fahren. Dies ist der Fall, wenn zwei Zugsignale auf unterschiedlichen Seiten der Strecke sind und nicht direkt gegenüber voneinander. Das kann durch verbinden von

Schienen, die nur auf einer Seite ein Zugsignal haben, passieren. Dadurch wird in beide Richtungen das Weiterfahren von Zügen blockiert und die Strecke kann nicht mehr befahren werden. Dies ist beim Bauen von Schienennetzen zu vermeiden.
Es gibt eine Ausnahme zu dem oben genannten Verhalten. Züge halten nicht vor roten Zugsignalen an, die sie selbst ausgelöst haben. Das heißt ein Zug darf in einen Block hineinfahren, in dem er selbst schon vorhanden ist, selbst wenn das durch ein Zugsignal verhindert werden sollte.

2.1.10 Verhalten von Zügen

Die Züge in Factorio fahren auf den Schienen. Zum Fahren benötigen die Züge Brennstoff, dabei gibt es unterschiedliche Arten Brennstoff. Sie können automatisch oder vom Spieler gefahren werden. Dabei können sie vorwärts und rückwärts fahren, wenn der Spieler sie manövriert. Wenn sie automatisch fahren können sie nur vorwärtsfahren und werden von den Zugsignalen auf ihr Ziel hin geleitet. Um einen automatisch fahrenden Zug die Möglichkeit zu geben, in beide Richtungen zu fahren, müssen zwei Lokomotiven in unterschiedliche Richtungen an den Zug gebaut werden.

Um Ressourcen zu transportieren können an die Lokomotiven Güterwagen gehängt werden. Es ist eine beliebige Kombination aus Lokomotiven und Güterwagen möglich.

Züge haben eine maximale Geschwindigkeit, die von vielen Faktoren abhängt. Einige Beispiele sind die verwendete Brennstoffart, die Anzahl an Lokomotiven und die Anzahl an gezogenen Güterwagen. Die Beschleunigung hängt genauso von diesen Faktoren ab. Die Bremskraft der Züge hängt von einem anderen Faktor ab. Sie kann von dem Spieler verbessert werden bis zu einem gewissen Wert. Dabei gilt, je stärker die Bremskraft, desto kürzer der Bremsweg. Durch diese ganzen Faktoren ist es schwierig, die Geschwindigkeit von Zügen akkurat zu simulieren. Besonders wenn sie durch Schienennetze mit Kreuzungen fahren, wo sie immer wieder abbremsen und beschleunigen müssen.

Beim Fahren reservieren Züge kommende Blöcke für sich. Es können schon reservierte Blöcke nicht noch einmal reserviert werden. Belegte Blöcke können auch nicht reserviert werden. Wie weit in voraus reserviert wird, hängt von der Geschwindigkeit der Züge ab. Es wird so viel Patzt reserviert, dass der Zug zu einem Stillstand kommen kann, ohne in einen neuen Block hineinfahren zu müssen. Wenn der Zug beim kompletten Bremsen in den Block fährt, wird dieser reserviert. Leicht anders ist es bei Kettensignalen. Eine Reihe an Kettensignalen wird im Ganzen reserviert. Das heißt es werden alle Blöcke reserviert, bis ein Signal kommt.

2.2 Blueprints

Nachdem der Spieler ein großes Schienennetz geplant und gebaut hat, möchte er es in Zukunft einfacher bauen können. Hilfreich dafür wäre eine Copy-und-Paste Funktion. Diese Rolle übernehmen die Blueprints. Der Spieler kann einen Bereich auswählen, in dem sich Entities befinden und diese werden vom Spiel gespeichert. Danach kann der Spieler das blueprint an eine andere Stelle platzieren und die Entities werden automatisch aufgebaut. Blueprints können auch benutzt werden, um Bauwerke mit anderen Spielern zu teilen. So ist es möglich, komplexere Strukturen zu transferieren und dann automatisch in einer Welt bauen zu lassen. Die Informationen eines Blueprints werden für das Teilen von Factorio in eine Zeichenkette verschlüsselt. Wenn man die Zeichenkette entschlüsselt, kann man folgende Struktur erkennen.

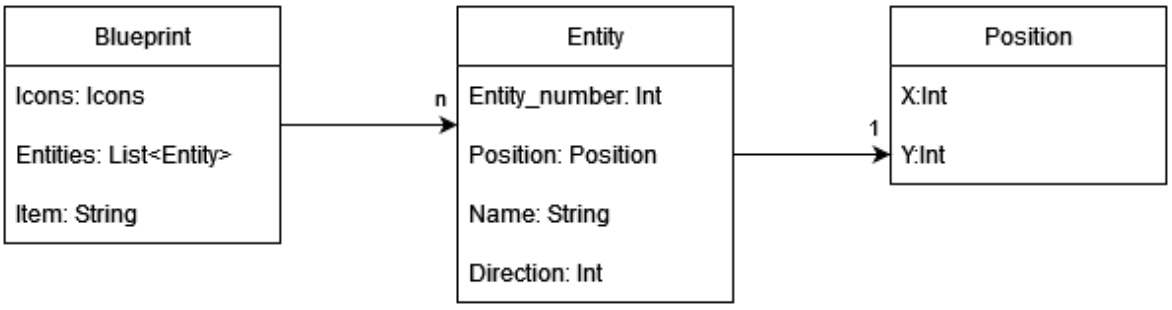

Abbildung 9: Klassendiagramm interner Aufbau von Blueprints

Hierbei ist zu beachten, dass dieses Diagramm vereinfacht wurde, um nur die für diese Arbeit relevanten Teile zu zeigen. Hier relevant ist die Liste an Entitys, welche verrät, wo welches Objekt sich befindet. Über dessen Positionsparameter ist es nun möglich, ein gesamtes komplexes System wieder zu rekonstruieren.

Ein Blueprint-Book ist eine Ansammlung an Blueprints. Blueprint-Books können rekursiv sein, das heißt ein Blueprint-Book kann in einem Blueprint-Book gespeichert werden. Ein Blueprint-Book kann unendlich viele Blueprints[1] halten.

2.3 Der Begriff des Deadlocks

Im Allgemeinen wird von einem Deadlock gesprochen, wenn zwei Prozesse sich gegenseitig blockieren. Das ist der Fall, wenn sie auf eine Ressource zugreifen wollen, die von dem anderen Prozess benutzt und somit blockiert wird.

In dieser Arbeit ist ein Deadlock eine Konfiguration von mindestens zwei Zügen, die in einen Block einfahren können, aber diese ohne einen externen Eingriff nicht mehr verlassen können. Dies ist genau dann der Fall, wenn die Züge sich gegenseitig blockieren, indem sie darauf warten, dass der jeweilige andere weiterfährt. Dadurch wird eine Warteschleife erzeugt, die nur vom Spieler behoben werden kann.

Folgende Voraussetzungen gibt es für einen Deadlock:

Ein zu belegender Gleisabschnitt (Block) kann nur von einem Zug belegt werden. Nach der Belegung durch den ersten Zug kann kein zweiter Zug in den Block einfahren. [2]

Ein Zug muss in seinem Block warten, bis der Block, in den er einfahren möchte, nicht mehr von einem anderen Zug belegt ist und somit frei ist.

Ein Block gilt als frei, wenn der belegende Zug diesen vollständig verlassen hat. Durch diesen Umstand kann ein Zug mit erhöhter Länge mehr als einen Block belegen.

Die Bildung geschlossener Warteketten ist möglich [2], d.h. zwei oder mehr Züge warten gegenseitig auf das frei werden eines belegten Blockes, wobei kein Block frei werden kann, wegen anderen belegten Blöcken.

Die ersten drei Voraussetzungen ergeben sich aus dem Zugsystem von Factorio, die letzte aus der Definition eines Deadlocks und den ersten drei Voraussetzungen.

Damit diese Definition für die Szenarien im Spiel anwendbar ist, muss folgende Randbedingung gelten:

Die Züge können aus dem betrachteten Schienennetz herausfahren. Das heißt, dass Blöcke, die einem Ausgang entsprechen immer als frei angenommen werden. Wenn Züge das Schienennetz nicht verlassen können, weil sich Züge außerhalb des betrachteten Schienennetzes befinden, ist nicht das Schienennetz selbst das Problem, sondern das Problem liegt außerhalb des Schienennetzes.

Ein Beispiel für einen Deadlock ist dieses Schienennetz mit zwei Kreuzungen.

Abbildung 10: Schienennetz mit Deadlock

Ein Zug in dem gelben Block mit Nummer 2 muss warten, bis der pinke Block mit Nummer 1 frei von Zügen ist. Genauso gilt für ein Zug im pinken Block, dass er warten muss, bis der gelbe Block frei von Zügen ist. Wenn nun also zwei Züge gleichzeitig in das Schienennetz einfahren, blockieren sie sich gegenseitig. Der Zug im pinken Block kann nicht weiterfahren wegen des Zuges im gelben Block und andersherum. Als Lösung muss einer der beiden Züge rückwärts wieder aus dem Schienennetz herausfahren, sodass einer der beiden Blöcke nicht mehr belegt ist. Dann kann der Zug, der auf das Freiwerden von dem Block gewartet hat, weiterfahren und danach der Zug, der Platz gemacht hat.

Die blauen Blöcke sind Eingangs bzw. Ausgangsblöcke. Sie tragen nicht zum Deadlock bei, sind aber Teil des Schienennetzes. Die Blöcke mit Nummern 4 und 6 sind dabei die Ausgangsblöcke. Für diese Blöcke gilt die Randbedingung, dass dort keine Züge drin stehenbleiben können.

2.4 Gerichtete Graphen

Um die Analyse des Schienennetzes zu erleichtern, kann das Netzt von Factorio durch gerichtete Graphen abstrahiert werden. Die Daten können in ein Datenformat übertragen werden, dass programmatisch besser ausgewertet werden kann. Dazu eignen sich gerichtete Graphen [3]. Ein gerichteter Graph [4] besteht aus einer Liste an Knoten. Jeder Knoten dieser Liste ist über eine Id eindeutig identifizierbar. Dazu gibt es eine Liste an Verbindungen. In den Verbindungen ist gespeichert, welche beiden Knoten sie verbinden. Die Verbindungen sind gerichtet, das heißt es gibt einen Start und einen Zielknoten. Es können beliebig viele Verbindungen von einem Knoten starten und genauso beliebig viele Verbindungen an einem Knoten Enden. Um den Graphen analysieren zu können, müssen endlich viele Knoten und Verbindungen existieren.

In diesem Fall können die Blöcke von Knoten repräsentiert werden. Die Verbindungen zeigen dann die Beziehungen zwischen den Blöcken an. Ein Zug muss auf einen anderen Zug genau dann warten, wenn eine Verbindung zwischen den Knoten, die ihre Blöcke repräsentieren, besteht. Die Richtung der Verbindung gibt an, welcher Zug warten muss. Der Zug im Block repräsentiert vom Startknoten muss auf den Zug im Block repräsentiert vom Zielknoten warten.

Das Beispiel in Kapitel 2.3 kann in Graphen Form dann so aussehen:

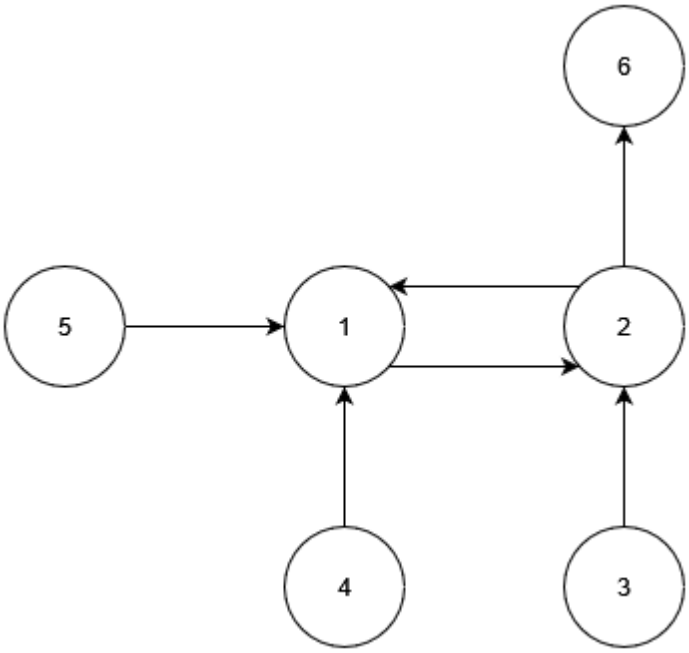

Abbildung 11: Graph mit zirkularer Abhängigkeit

Es ist zu erkennen, dass Züge im Block mit der Nummer 1 auf Züge im Block mit der Nummer 2 warten müssen. Sie bilden eine zirkulare Abhängigkeit. Die restlichen Blöcke sind nicht für den Deadlock verantwortlich und sind lediglich Ein- und Ausgänge des Schienennetzes. Bei komplizierteren Schienennetzen ist es schwieriger, diese zirkularen Abhängigkeiten zu finden. Mithilfe von diesen Graphen kann nun der Schritt der Deadlock Analyse vereinfacht werden.

2.5 Tiernan Algorithmus

Es gibt schon bestehende Algorithmen, um zirkularen Abhängigkeiten zu finden. Einer dieser Algorithmen ist der von Tiernan [5]. Dieser wird im Pilotprojekt dieser Arbeit benutzt und deswegen hier erläutert.

Der Tiernan Algorithmus bildet einen Pfad, um zirkulare Abhängigkeiten zu finden. Ein Pfad ist eine Liste an Knoten. Über die Verbindungen durchläuft der Algorithmus den Graphen und speichert jeden Knoten, an dem er vorbeikommt in dem Pfad. Das Durchlaufen passiert über folgende Regeln:

Der Zielknoten der Verbindung darf nicht im Pfad schon einmal vorhanden sein.

Die Id des Zielknotens der Verbindung muss kleiner sein als die Id des Startknotens des Pfades

Der Knoten, zu dem gelaufen werden soll, darf vom derzeitigen Knoten noch nicht gelaufen worden sein. Das heißt von einem Knoten soll nicht zweimal zu einem bestimmten anderen Knoten gelaufen werden.

Wenn eine der Bedingungen zutrifft, wird geprüft, ob eine zirkulare Abhängigkeit vorliegt und danach mit den anderen Verbindungen des derzeitigen Knotens fortgefahren. Eine zirkulare Abhängigkeit liegt dann vor, wenn der betrachtete Zielknoten der Verbindung derselbe Knoten wie der Startknoten des Pfades ist. Dann wissen wir, dass der Pfad eine zirkulare Abhängigkeit bildet, und speichern diesen.

Danach wird nach weiteren zirkularen Abhängigkeiten gesucht, indem das Durchlaufen mit einer anderen Verbindung des derzeitigen Knotens fortgeführt wird. Wenn es keine weiteren Verbindungen zu überprüfen gibt, wird ein Schritt zurück gemacht. Das heißt der letzte Knoten des Pfades wird entfernt und der vorherige Knoten wird zu derzeitigen Knoten. Es wird gespeichert, dass die Verbindung nicht nochmal betrachtet werden muss. Das Durchlaufen wird wieder mit den anderen Verbindungen fortgeführt.

Der Algorithmus ist mit dem ersten Durchlauf fertig, wenn nur noch der Startknoten im Pfad ist und alle Verbindungen von diesem betrachtet wurden. Dann wird der gesamte Vorgang wiederholt mit einem anderen Startknoten. Diese Wiederholungen werden so lange durchgeführt, bis jeder Knoten einmal Startknoten war.

Als Output hat der Algorithmus eine Liste an Pfaden, die eine zirkulare Abhängigkeit darstellen.

3. Methodik / Analyse

In diesem Kapitel wird die Methodik der Analyse vorgestellt und begründet, warum eine gewisse Methodik gewählt wurde. Es werden außerdem die verwendeten Werkzeuge für die Implementierung erklärt.

3.1 Lösungsmöglichkeiten

Es gibt unterschiedliche Möglichkeiten, um Deadlocks zu erkennen. Im nachfolgenden werden drei Möglichkeiten vorgestellt und miteinander verglichen. Am Ende wird eine begründete Entscheidung getroffen.

3.1.1 Empirische Ansätze

Als erste und naive Methode gibt es die Nachbildung des Zug-Systems. Der Graph wird analysiert, indem das Zugsystem von Factorio simuliert wird. Dabei wird versucht, möglichst genau das Fahren der Züge durch die Kreuzung nachzubilden. Der Aspekt der Geschwindigkeit und Beschleunigung der Züge ist dabei zu kompliziert, um korrekt repliziert zu werden, so dass dieser Aspekt nicht berücksichtigt wird. Alle Züge fahren in der Simulation gleichschnell und können unmittelbar beschleunigen und abbremsen, so dass vereinfacht dafür keine Zeit benötigt wird. Mit diesen Rahmenbedingungen kann nun simuliert werden, wie die Züge durch die Kreuzung fahren und wo sie sich gegenseitig eventuell blockieren.

Die Simulation passiert schrittweise, das heißt zu einem Zeitpunkt ist ein Zug in dem Block X und im nächsten Zeitpunkt ist er in den Block Y weitergefahren. Wenn ein Zug schon in dem Block Y ist, kann der Zug für diesen Schritt nicht weiterfahren und wartet für den Schritt in dem Block X.

Um mithilfe dieser Simulation nun Deadlocks zu erkennen, gibt es zwei unterschiedliche Möglichkeiten, die Züge in die Kreuzung einfahren zu lassen. Einmal mit einer systematischen oder in zufälliger Art und Weise.

Eine Systematik würde der Breitensuche [6] ähneln. Bei jedem Schritt werden alle möglichen Zugpositionen, bzw. deren Möglichkeiten weiterzufahren, betrachtet und gespeichert. Hierfür gibt es eine Next-States Funktion, die berechnet, welche Zustände, heißt Zugpositionen, in der nächsten Zeiteinheit möglich sind. Der Start dieser Suche ist eine leere Kreuzung und das Ziel ist eine Kreuzung, in der sich Züge befinden und es keine Möglichen eines nächsten Zustandes gibt. In der Next-States Funktion ist definiert, in welcher Reihenfolge die Züge in die Kreuzung einfahren. Dabei wird sichergestellt, dass jede mögliche Kombination an einfahrenden Zügen abgedeckt wird. Wenn beispielsweise zwei Einfahrten in die

Kreuzung vorhanden sind, werden folgende Kombinationen ausprobiert: wenn die Züge gleichzeitig einfahren, wenn ein Zug zuerst einfährt, wenn der andere Zug zuerst einfährt, wenn der zweite Zug zwei Zeiteinheiten später einfährt usw.

Bei dieser Lösungsmöglichkeit wird der Speicher sehr schnell mit Zuständen gefüllt, sodass der Algorithmus für große Kreuzungen nicht praktikabel ist. Dafür werden beim Durchlaufen **alle** Zustände betrachtet, wodurch ein Deadlock sicher gefunden werden kann.

Die zweite Möglichkeit ist, die Züge zufällig in die Kreuzung einfahren zu lassen. Dabei wird dann nur ein Zustand gespeichert, der jetzige. Die Next-States Funktion gibt es in dieser Möglichkeit weiterhin. Jedoch werden bei Entscheidungen nicht mehr beide Varianten der Entscheidung ausprobiert, sondern eine zufällig ausgewählt. Dadurch gibt es kein Speicherproblem wie bei der Breitensuche. Allerdings gibt es auch keine Sicherheit, alle Zustände betrachtet zu haben. Die Beendigung des Algorithmus ist dann nicht mehr gewährleistet, wenn es keinen Deadlock gibt. Es kann nach einer bestimmten Anzahl an Schritten eine Aussage getroffen werden, dass die Kreuzung wahrscheinlich Deadlock-frei ist, bewiesen wird es von dem Algorithmus aber nicht.

3.1.2 In-game Möglichkeiten

Das Problem der Deadlock Erkennung ist natürlich in der Community von Factorio bekannt und somit gibt es schon Lösungen dafür. Zum Beispiel gibt es einen Mod, heißt von der Community erstellter optionaler Spielinhalt, mit dessen Hilfe eine Kreuzung getestet werden kann. Der Mod [7] lässt Züge an den Eingängen der Kreuzung erscheinen und an den Enden der Kreuzung verschwinden. Dadurch kann sehr erfolgreich der Durchsatz der Kreuzung getestet werden. Das Testen auf Deadlocks ist auch dadurch möglich, indem zufällig Züge in die Kreuzung einfahren und somit möglicherweise ein Deadlock eintritt, wodurch bewiesen ist, dass die Kreuzung nicht Deadlock-frei ist. Leider kann durch dieses Vorgehen nicht bewiesen werden, dass die Kreuzung Deadlock-frei ist, aber durch so einen ständigen Stresstest kann durchaus gezeigt werden, dass eine geringe Deadlock Wahrscheinlichkeit besteht.

Dieser Ansatz hat den Vorteil, dass es direkt im Spiel passiert und somit keine Abstraktionsebene für Ungenauigkeit sorgen kann. Wenn zum Beispiel ein Signal von dem externen Programm nicht richtig erkannt wird, kann das Ergebnis von der Analyse nicht korrekt sein. So kann durch das Beschleunigen und Abbremsen der Züge im Schienennetz ein Zug in eine Position fahren, die vom externen Programm nicht beachtet wird. So könnte das Programm davon ausgehen, dass ein Zug an bestimmten Stellen nicht fahren kann, wenn in der Realität doch an diese Stelle gefahren werden kann.

Außerdem könnte eine Lösung im Spiel für den Workflow des Spielers besser sein, da er dann nicht extra ein externes Programm starten muss und das Spiel verlässt.

Eine Möglichkeit wäre es nun, eine neue Mod zu schreiben, die dann die Analyse durchführt. Factorio unterstützt Mods über die Scriptsprache Lua. Diese liefert eine API [8], mit der Mods interagieren können. Mit Hilfe dieser API kann die Mod herausfinden, wie das Schienensystem aufgebaut ist und dann dieses analysieren. Dabei könnte zwischen zwei Ansätze unterschieden werden:

(1) Die Züge durch das Schienennetz fahren lassen, wie es in der oben genannten Mod der Fall ist.

(2) Alternativ könnte ein Graph mithilfe der API gebaut und analysiert werden.

Zur ersten Möglichkeit:

Die Mod kann hinsichtlich der Deadlock Analyse verbessert werden, indem eine Systematik hinzugefügt wird. Es soll jede mögliche Kombination durchlaufen werden, wie Züge durch das Schienennetz fahren können. Eine Kombination ist dabei, eine bestimmte Anzahl an Zügen, die zu bestimmten Zeiten in das Schienennetz einfahren. Außerdem gehört zur Kombination, welche Eingangs- und Ausgangs-Blocks die Züge haben. So ist ein Beispiel für eine Kombination bei einer einfachen Kreuzung mit Richtung A und B, dass ein Zug zum Zeitpunkt T in Richtung A einfährt. Eine zweite Kombination könnte sein, dass zwei Züge einfahren, einer in Richtung A zum Zeitpunkt T und einer in Richtung B zum Zeitpunkt T+1. Dabei muss beim Implementieren genauer definiert werden, was nun der Abstand zwischen T und T+1 ist, so dass das Zeitintervall genau spezifiziert wird.

Durch diese Systematik können Deadlocks besser gefunden werden, weil jede Möglichkeit ausprobiert wird. Dadurch kann mit höherer Sicherheit gesagt werden, dass es keinen Deadlock gibt, wenn keiner gefunden wurde.

Der Nachteil ist derselbe wie im Kapitel 3.1.1. Das Ausprobieren von allen Möglichkeiten dauert lange. In diesem Fall sogar möglicherweise noch länger, weil eben die Geschwindigkeit der Züge beachtet wird. Da die Züge wirklich durch das Schienennetz fahren, benötigt jedes Ausprobieren so lange wie die Züge brauchen, um durch das Schienennetz zu fahren. Bei sehr großen Schienennetzen dauert es also zum einen länger durch das Fahren der Züge und zum anderen länger, weil es viele unterschiedliche Kombinationen gibt, die ausprobiert werden müssen.

Zur zweiten Möglichkeit:

Es kann ein Mod gebaut werden, die mithilfe gerichteter Graphen das Schienennetz analysiert.

Die API von Factorio hat eine Reihe an Funktionen für das Interagieren mit Schienen [9]. So kann man zum Beispiel abfragen, zu welcher Strecke eine Schiene gehört. Danach kann abgefragt werden, wo die Zugsignale der Schiene sind und welche Zugsignale diese beobachten. Mithilfe dieser und weiteren Funktionen könnte die Mod das Schienennetz auslesen und zu einem gerichteten Graphen transformieren. Dieser Graph kann dann auf zirkulare Abhängigkeiten analysiert werden mithilfe des Tiernan Algorithmus. Der Vorteil daran ist, dass das Analysieren von einem gerichteten Graphen schneller ist als das Ausprobieren von vielen Möglichkeiten.

Das Problem an diesem Ansatz ist, dass Mods in Factorio nicht parallel zum Spielgeschehen [10] laufen können. Die Analyse des Graphen würde für kleine Schienennetze schnell durchlaufen und somit das Spielgeschehen nur minimal beeinträchtigen. Eine Analyse eines großen Schienensystems würde aber durchaus mehrere Sekunden bis Minuten dauern, in der Zeit das Spiel pausiert. Die Anforderungen von Spielern sind üblicherweise, dass die Analyse das Spielgeschehen nicht unterbricht, sondern währenddessen stattfinden kann.

3.1.3 Analytischer Ansatz

Der Ansatz, zirkulare Abhängigkeiten in einem gerichteten Graph zu finden, kann auch in einem externen Programm verwendet werden. Der Ansatz besteht grob aus drei Schritten. Zuerst wird von dem Schienennetz ein Blueprint in Factorio erzeugt, das analysiert werden soll. Dieses Blueprint wird in das externe Programm gegeben. Das Programm erzeugt einen gerichteten Graphen aus dem Blueprint, indem es zuerst das Schienennetz des Blueprints nachbaut und dann dieses durchläuft und transformiert. Als zweiten Schritt wird der gerichtete Graph auf zirkulare Abhängigkeiten untersucht. Dies passiert mithilfe des Tiernan Algorithmus. Diese zirkularen Abhängigkeiten müssen danach genauer daraufhin untersucht werden, ob der Deadlock verhindert wird. Das Auftreten von Deadlocks kann durch Kettensignale verhindert werden, auch wenn es eine zirkulare Abhängigkeit im Graphen gibt. Dies kommt dadurch, dass Züge durch Kettensignale in Fahrtrichtung früher warten. So blockieren sie die kommenden Blöcke nicht und damit die Züge, die dort durchfahren auch nicht. Wenn die Züge sich nicht blockieren, kommt es nicht zu einem Deadlock.

Um nun festzustellen, ob ein Deadlock auftritt oder von Kettensignalen verhindert wird, müssen die zirkularen Abhängigkeiten untersucht werden. Es muss zuerst herausgefunden werden, welche Gesamtstrecken zu der zirkularen Abhängigkeit gehören. In jeder dieser Gesamtstrecken muss mindestens ein Signal vorkommen. Bei den Gesamtstrecken mit ausschließlich Kettensignalen warten die Züge immer am Eingang des Schienennetzes und blockieren somit keinen Zug, was den Deadlock verhindert.

Nach der Analyse der zirkularen Abhängigkeiten kann das Programm ausgeben, ob es einen Deadlock gibt oder nicht.

Dieser Ansatz besitzt die höchste Abstraktionsebene, einmal durch das Nachbilden des Schienennetzes und durch den gerichteten Graphen. Dies könnte zu einer höheren Anzahl an Ungenauigkeiten oder sogar Fehlern führen.

Der Ansatz hat den Vorteil, dass der Hauptteil der Analyse, das Finden von zirkularen Abhängigkeiten von dem Tiernan Algorithmus übernommen wird. Die Skalierung der Performance anhand der Größe der gerichteten Graphen vom Tiernan Algorithmus ist bekannt. Das macht es möglich, eine Voraussage zu treffen, wie gut der gesamte Ansatz mit der Größe der Kreuzung skaliert. Der Tiernan

skaliert in O-Notation O(n* e(c + 1)). Wobei n die Anzahl an Knoten, e die Anzahl an Verbindungen und c die Anzahl an Kreisen ist [11].
Die Performance des gesamten Ansatzes setzt sich zusammen aus der Skalierung der Vor- und Nachbereitung und dem Tiernan-Algorithmus. Da die Vor- und Nachbereitung keine schlechtere Skalierung als der Tiernan haben werden, ist der Tiernan das auschlaggebende Element für die O-Notation.

3.1.4 Entscheidung

In diesem Absatz werden die Vor- und Nachteile von dem oben erklärten Ansätzen gegenübergestellt und begründet entschieden, welcher Ansatz implementiert wird.

Es gibt mehrere Kriterien, nach denen die Entscheidung gefällt wird. Da wären Nutzerfreundlichkeit, Performance und Fehleranfälligkeit.

Am nutzerfreundlichsten sind die Ansätze, die Analyse als Mod zu implementieren. Dadurch kann der Nutzer direkt während des Spielens sein Schienennetz auf Deadlocks überprüfen. Für die anderen Ansätze muss der Spieler ein externes Programm öffnen und das Schienennetz mithilfe von Blueprints transferieren. Den einzigen Schritt, den der Nutzer für das Nutzen einer Mod machen muss, ist diese zu installieren, was bei den anderen Ansätzen auch der Fall ist. Dafür könnte die Analyse einer Mod das Spielgeschehen stören oder im Schlimmsten Fall Factorio zum Absturz bringen. Es ist für Spieler nicht so problematisch, wenn ein extern laufendes Programm abstürzt, da dadurch kein Spielfortschritt verloren geht.

Die Performance von dem analytischen Ansatz ist voraussichtlich die Beste. Er skaliert am besten mit der Größe des Schienennetzes. Am schlechtesten skaliert der empirische Ansatz, in dem alle Kombinationen ausprobiert werden, mit der Größe des Schienennetzes. Nur ein zusätzlicher Eingangsblock verdoppelt die Kombinationen, die ausprobiert werden müssen. Dasselbe gilt für das zufällige Ausprobieren, es muss bei einem größeren Schienennetz um einiges länger ausprobiert werden, um einen Deadlock zu finden.

Die Mod Lösungen sind die, die am wenigsten Abstraktionsebenen besitzen. Dadurch gibt es bei ihnen weniger Fehlerquellen. Die Lösung mit der größten Fehlerquelle ist das zufällige Ausprobieren, bei ihr ist der Zufall die größte

Fehlerquelle. Es kann sein, dass ein Deadlock sehr schnell gefunden wird oder überhaupt nicht. Dies macht den Ansatz unzuverlässig, kann also keine gute Lösung sein. Die analytische Methode hat viele Abstraktionsebenen, wodurch Fehler passieren können. Dort wird der häufigste Fehler ein Falsch-Positiv Fehler sein, dass eine zirkulare Abhängigkeit als Deadlock erkannt wird, auch wenn ein Kettensignal dies verhindert.

Insgesamt hat die analytische Methode den größten Vorteil in der Performance. Aus diesem Grund wurde sie gewählt. Als Implementierungsmöglichkeiten für diese Methode gibt es die Möglichkeit eine Mod zu programmieren oder ein externes Programm. In Absprache mit dem Betreuungsleiter wurde entschieden, dies nicht als Mod zu implementieren, sondern als externes Programm.

3.2 Verwendete Werkzeuge

Dieser Abschnitt beschreibt, welche Werkzeuge bei der Implementierung genutzt wurden.

Den internen Aufbau von Factorio blueprints haben wir durch die Webseite https://burnysc2.github.io/Factorio/Tools/DecodeBlueprint/ herausgefunden. Diese übersetzt den Blueprint aus einem nicht menschlich lesbaren Format in ein JSON-Format, das von Menschen lesbar ist. Dadurch konnten wir genau herausfinden, welche Daten Factorio beim Exportieren bietet. Dieses Wissen wurde für das Pilotprojekt benötigt. In diesem wurde der Vorgang der Webseite, die Daten in ein JSON zu transformieren, übernommen. Daraufhin wurde ein zweites Tool genutzt Google Gson [12], um aus dem JSON-Format, Kotlin Objekte zu generieren. Dieser Vorgang importiert die Blueprint Daten in das Programm, damit danach mit ihnen weitergearbeitet werden kann.

Zur Visualisierung des Outputs wurde das Programm Graphviz[13] zu Hilfe genommen. Das Programm zeichnet aus einem Input einen Graphen. Dies wurde bei der Implementierung besonders zum Debuggen genutzt. Außerdem wird es genutzt, um dem Benutzer das Ergebnis der Analyse zu visualisieren.

4. Pilotprojekt (Ergebnis)

In diesem Kapitel werden wir unser entwickeltes Pilotprojekt vorstellen sowie auf die entstandenen Probleme eingehen und wie diese angegangen wurden. Hierbei wird zunächst eine Top Level Übersicht über das Projekt gegeben, um daraufhin die verwendeten Datenstrukturen genauer zu erläutern. Im weiteren Verlauf wird auf den Datenfluss des Programmes eingegangen und hierbei iterativ in die inneren Details des Projekts erläutert. Bei der Erklärung dieser Funktionen werden nicht immer Standard UML Diagramme verwendet und teilweise in Spiel Bilder verwendet, um einzelne Punkte zu verdeutlichen. Ziel des Programms ist zum einen festzustellen, ob ein Deadlock überhaupt möglich ist und sofern dies der Fall, was die maximale, sichere Zuglänge für das zu untersuchendes System ist.

4.1 Übersicht

Die Konsolen Anwendung wird zunächst aufgerufen mit einem Blueprint-String welcher auch als Datei angenommen werden kann. Weitere Parameter und Flags werden in dem Kapitel 4.8 besprochen.

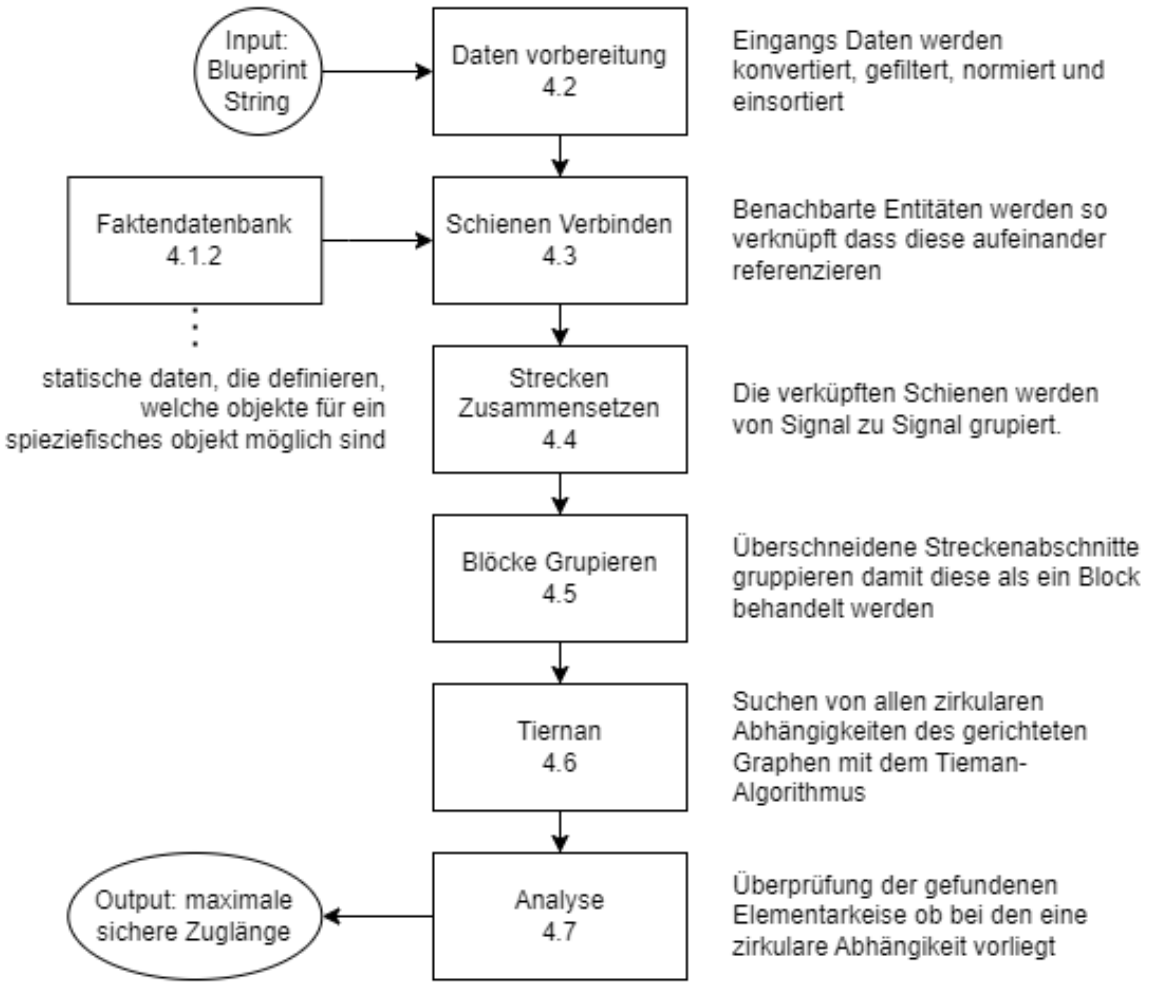

Abbildung 12: Übersicht zur Datenverarbeitung

Die Abbildung 12 zeigt das gesamte Schaubild und den Datenfluss der Arbeit. Dieser ist im Allgemeinen linear welches sich daraus ergibt, dass die Antwort, die in den Eingangsdaten vorkommt, nur schwierig ist zu lesen. Daher ist auch der erste Schritt die eingehenden Daten dekomprimiert, um aus dem Blueprint String ein lesbares JSON zu formen. Dieses wird dann via GSON zu einem Objektbaum geparst. Darauf folgt das die irrelevanten Daten rausgefiltert werden sowie die gefundenen Entitys nun im ersten Schritt normiert und entsprechend einsortiert werden.

Darauf agiert das erste komplexere System und verbindet die Schienen, welches für direkte Nachbarn eine Referenz erstellst. Dieses erschaffene Schienennetzt wird von der Stecken-Erstellung genutzt, um herauszuarbeiten, welche Strecken die Züge befahren können. Für die Strecken werden Kollisionsboxen berechnet, um bestimmen zu können, welche Strecken zu demselben Block gehören. Zudem wird

gespeichert welche strecken als nächstes befahrbar sind von einer gegebenen strecke aus.

Die erschaffenen Blöcke bilden einen Abhängigkeitsgraphen, auf welchen die Analyse angewendet werden kann, um herauszufinden welche zyklischen Abhängigkeiten in dem System existieren. Um diese zu finden, wird eine Algorithmus verwendet, welcher von James C. Tiernan zunächst beschrieben wurde. Sofern eine oder mehrere zyklische Abhängigkeiten gefunden wurden, müssen diese genauer betrachtet, um festzustellen was die maximale sichere Zuglänge für dieses System ist. Das Minimum dieser Werte stellt dann die sichere Länge für das Gesamtsystem da.

Im weiteren Verlauf wird zunächst auf die Datenstruktur erläutert, die für dieses Projekt verwendet wird. Darauf wird auf die Faktendatenbank genauer eingegangen, welche auf der linken Seite in Abbildung XY zu sehen ist. Auf diesem wird dem Programm Ablauf von oben nach unten gefolgt, wobei die einzelnen Boxen teilweise tiefer aufgeschlüsselt werden.

4.1.1 Klassen Diagramm

Die Abbildung 13 zeig eine Übersicht über die verschiedenen Klassen des Programmes. Hierbei handelt es sich um einen Mix zwischen Klassendiagramm und ER-Modell allerdings sind nicht alle Felder fachlich korrekte Klassen, da Kotlin Code

außerhalb von Klassen erlaubt..

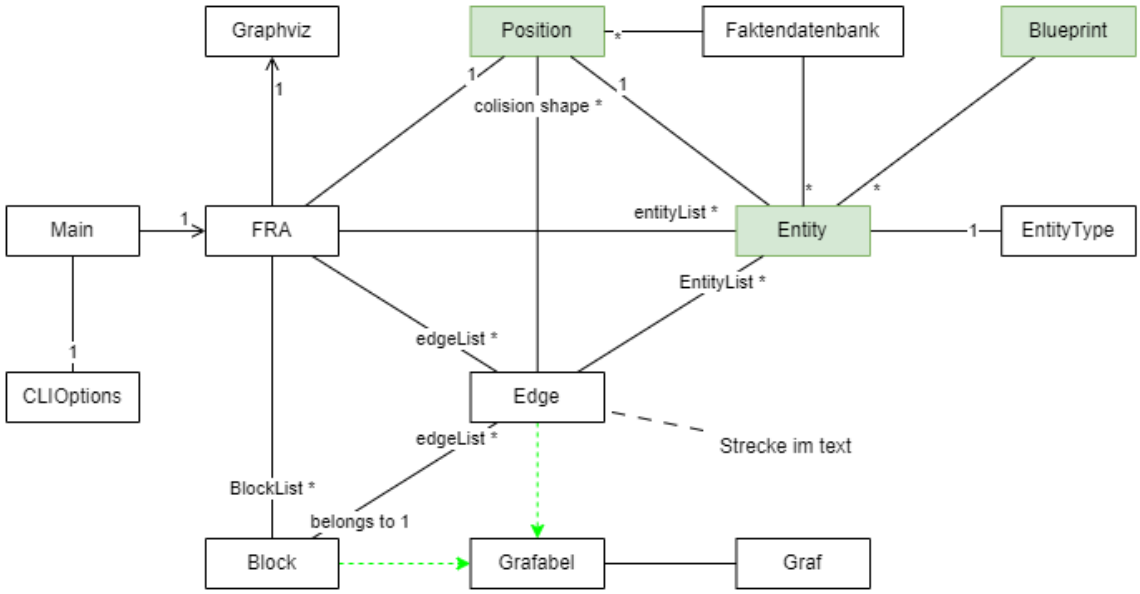

Abbildung 13: Übersicht Datenaufbau

Zunächst ist der Einstiegspunkt der Applikation die Main Funktion auf der linken Seite. Die eingegebenen Parameter werden mit den CLIOptions abgefangen, um verschiedene kleinere Einstellungen zu setzten. Auf diese wird am Ende des Kapitels in 4.8 eingegangen. Die Main-Funktion ruft dann mit dem angegebenen Blueprint-String den Hauptteil der Anwendung auf. FRA steht hier für Factorio Rail Analyzer und enthält alle Variablen welche über die Laufzeit hinweg benötig werden.

Zunächst erkennbar sind oben rechts, grün markiert, die Klassen, die ursprünglich bereits in Abbildung 9 vorgestellt wurden. Diese wurden um Funktionen und Variablen erweitert, um weitere Informationen abzubilden, welche nicht von dem Blueprint mitgeliefert werden. Rechts von Entity findet sich Entity-Type einem Enum, welches alle Möglichkeiten widerspiegelt, die für die Analyse relevant sind. Über Entity findet sich die Faktendatenbank. Dies ist keine Klasse per se, wird hier aber dargestellt, um zu verdeutlichen, dass diese statischen Informationen über Entitys beinhalten. Der Aufbau und Details hierzu werden im nächsten abschnitt besprochen.

Im weiteren Verlauf der Anwendung werden zunächst die Variablen in Entitys erweitert, um darauf Entitys in einer Strecke zu gruppieren. Hierbei hält die Edgeklasse, im Weitern verlauf Strecke genannt, lediglich Referenzen auf eine Reihe an aufeinanderfolgenden Entitys. Blöcke sind wiederum eine weitere Gruppierung von Strecken. Per Definition gehört eine Strecke zu genau einem Block. Für Entitys hingegen ist dies nicht ganz so strickt. Signale gehören in der Regel zu mindestens 2 Strecken und auch bei Schienen kann es vorkommen das diese zu einer oder mehreren Strecken gehören. Daher existiert hier keine Referenz.

Unten ist dann noch das Graflabel Interface zu sehen welches von Edge und Block implementiert wird. Die Klasse Graf ist eine abstrakte Klasse, welche mit jeder Klasse welche Graflabel implementiert, instanziiert werden kann. Grad wird von dem Tiernan Algorithmus (Tiernan, 1970) verwendet, um dessen Laufzeit variabel zu halten

Zuletzt gibt es noch über der FRA-Klasse eine Klasse Graphviz, welche Funktionen zur Darstellung der Daten gruppiert, welche auch auf die Bibliothek Graphviz zugreift.

4.1.2 Faktendatenbank

In dem Abschnitt 2.1 wurde auf den Aufbau der Entitys und Spezifikationen von Schienen und Signale vorgestellt. Daraus resultiert, das für eine gegebene Schiene nur begrenzt viele Möglichkeiten existieren, wo ein weiteres Entity existieren könnte, welches für die gegebene Schiene relevant ist. Dies sind maximal 10 unterschiedliche Möglichkeiten pro Entity Variante.

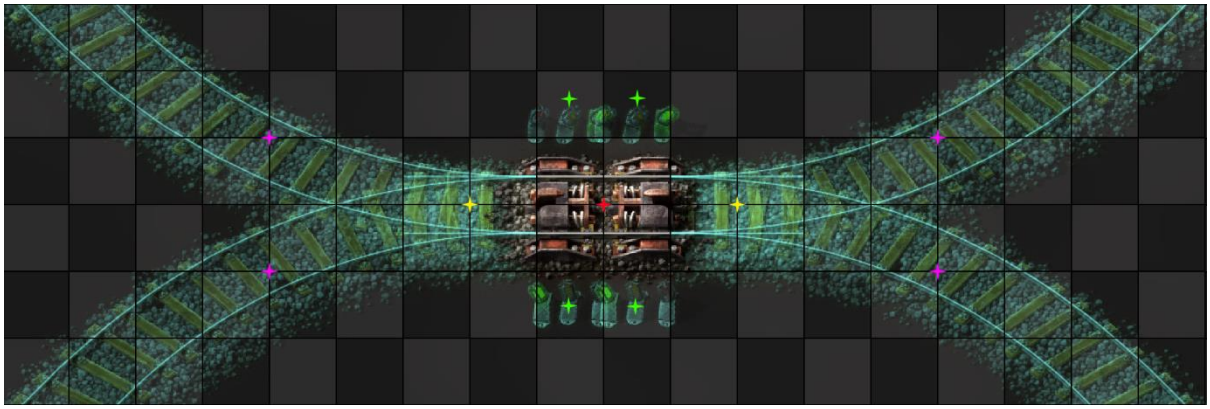

Abbildung 14: gerade schienen mit allen möglichen Entitys dargestellt

Für verschiedene Rotation des gleichen Objektes ergeben sich Variationen der Daten. Diese könnten theoretisch mit einer Funktion abgebildet werden, welche dies entsprechenden werte berechnet. Allerdings hat es sich als einfacher ergeben, diese Daten als einen Faktenwürfel zu definieren. Diese resultierende “Faktendatenbank” ist in keiner Form eine Datenbank und letztlich nur ein statisches 3-dimensionales Array, welches aber für uns gleich funkgiert. In der ersten Dimension des Facktenwürfels unterscheidet sich die Art des Entity-Typen, gerade oder gebogene Schienen. Die zweite Dimension des Würfels unterscheidet zwischen den unterschiedlichen Rotationsrichtungen der Schienen. Die dritte Dimension hat keine spezifische Sortierung und ist somit eine Liste aller möglichen Entitys, Schienen und Signale, dabei abhängig von den ersten 2 Dimensionen. Die aufgelisteten Objekte

geben dann den relativen Abstand zum Objekt an sowie deren Ausrichtung sowie ob diese Objekte sich mit oder gegen die Standart-Fahrtrichtung befinden. Für Signale wird außerdem noch “removeRelatedRail” Gesetz, wodurch markiert wird, ob ein Signal am Anfang oder am Ende einer Schiene sitzt, relativ zu der Fahrtrichtung des Signales.

So z.B.in Abbildung 14 ist zentral eine Schiene, deren Ankerpunkt in Rot maskiert wurde dargestellt. Alle anderen Maskierungen sind hierbei andere Objekte, welche von der roten Schiene erreichbar sind. Für all diese Objekte existiert ein Eintrag in der Faktendatenbank mit all den Informationen, die dieses Objekt ausmachen.

4.1.3 Definition von Richtung

In den vorherigen Abschnitten wurden bereits des häufigeren links und rechts als Richtungen sowie mit und gegen die Fahrtrichtung benannt. Diese Terme sind in der Entwicklung entstanden und spezifizieren Konventionen, welche hier erläutert werden.

Zunächst wird erläutert, warum es notwendig ist, Schienen in eine linke und eine rechte Seite einzuordnen. Wenn Schienen zu Strecken zusammengesetzt werden, muss eine Richtung angegeben werden, welche dieser Schienen als nächstes Element der Strecke infrage kommen. Dies Stellt auch sicher das nicht Schienen betrachtet werden welche bereits Teil der Strecke sind oder eine illegale strecke erschaffen würden. Das Problem besteht nun darin zu definieren, wo weiter und wo zurück liegt.

Die erste Möglichkeit ist hier nun die Schienen in einem Keis anzuordnen und nun einmal im Keis gehen und dabei jeweils die Richtungen als im Uhrzeigersinn und gegen den Uhrzeigersinn zu markieren. Dies funktioniert für die Kurven und die diagonalen relativ gut. Allerdings fällt auf, dass horizontalen und vertikalen Schienen genau gegensätzlich markiert wurden. Dies ist ein Problem, da es so unmöglich ist, diese Objekte zu unterscheiden, da sie sich gleich verhalten müssen. Eine Lösung wäre nun jeweils eine der horizontalen und vertikalen Richtungsdefinitionen umzudrehen, sodass dies zunächst konsistent sind. Dies sorgt allerdings für 4 Punkte, an denen die Suchrichtung invertiert werden muss, wenn die Strecke nach der Fahrtrichtung durchgegangen wird. Dieser Ansatz ist in Abbildung 33 dargestellt.

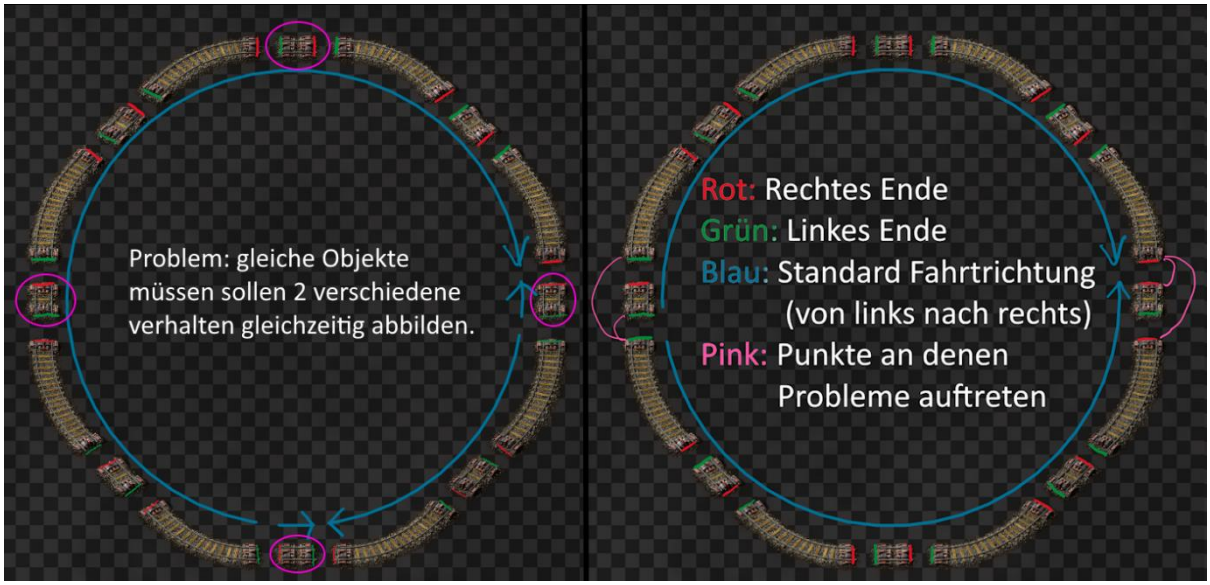

Abbildung 15: Links: Erste Idee Rechts: Lösung

Bei dieser ersten Idee existiert das Problem das sich gleiche Objekte anders verhalt abhängig der angrenzenden Objekte. Dies kann verbessert werden, indem wir auch die gesamte untere Hälfte den bauen Richtungspfeil invertieren.

In Abbildung 15 ist das System, dass sich letztendlich ergeben hat, abgebildet. Durch dieses System hat sich auch der Standard “in Fahrtrichtung” gebildet welches nun immer von links nach rechts fährt. In der Abbildung ist auch zu erkennen, wie alle gerade Schienenpaare nun die gleiche Ausrichtung haben und somit gleichbehandelt werden können.

In dem Bild ist auch erkennbar, dass es lediglich 2 Punkte gibt, an denen das Muster (grün, rot, grün rot...) nicht passt. Für diese in pink markierten Fällen muss immer noch ein Edge-Case abgefangen werden. Dieser Edge-Case wird in der Strecken Generierung abgefangen

4.2 Vorbereitung der Daten

In den Kapitel 4.2 bis 4.7 wird dem in 4.1 vorgestellten Diagramm und geht dabei auf den entstandenen Problemen und wie zu der aktuellen Lösung gekommen wurde ein

4.2.1 Eingangsdaten

Der Factorio Rail Analyser nimmt als Input den String eins einzelner Blueprints. Eine andere Datenform, wie ein Blueprint-Book wird zum aktuellen Zeitpunkt nicht unterstützt. Unter der Haube ist ein Blueprint ein JSON, welches Base64 komprimiert wurde. Daher wir der Blueprint zunächst dekomprimiert. Dies geschieht durch eine Funktion, welche gefunden wurde und zunächst von @marcouberti auf Stack Overflow bereitgestellt wurde [14]. Der resultierende JSON-String wird mit Hilfe von GSON in Objekte geparste. Bei dem Prozess desparsen werden zunächst alle Items in dem Blueprint in ein Entity verwandelt. Jedoch kann nur bei den relevanten Einträgen kann ein Entitytyp zugeordnet werden. Die restlichen sollen den Entitytyp Error bekommen. Dies hängt mit der Funktionsweise von GSON zusammen. Mit diesem Schritt oder Vorgehen bricht GSON das Nullsaftey Konzept von Kotlin. Dieses Problem lässt sich lösen, indem ein eigener Type Adapter für Kotlin definiert wird, was allerdings zum Zeitpunkt der Arbeit noch nicht geschehen ist. Der aktuelle Workaround ist nach dem Parsen alle Objekte, welche an dieser Stelle null sind, herauszufiltern und zu löschen. Alle anderen Entitys wandern in eine Liste, auf die später zurückgegriffen wird, um die Analyse voranzutreiben. Zudem wird aus der Liste aller Entity eine Liste aller Signale vorbereitend erstellt.

4.2.2 Normierung

In der nun erschaffenen Liste existieren alle Schienen und Signale. Im Weitern wird auf diese Liste als Entity Liste referenziert. Jedes Entity wird mit einer X und Y Koordinate übergeben, welche bei Erstellung des Blueprint auf die absoluten Koordinaten innerhalb der Factorio Spielwelt gesetzt werden. Dies bedeutet das X und Y jede Natürliche Zahl des 4 Byte Integer Zahlenraumes annehmen kann. Daher müssen die Werte normiert werden. Dies geschieht zunächst durch Bestimmung der maximallen X und Y, die jemals Vorkommen, um dann jede Koordinate mit dem

Maximalen Wert minus zu nehmen. Dies sorgt effektiv dafür, dass die obere linke Ecke des Blueprint bei (0,0) liegt, was bereits erwähnt wurde.

4.3 Schienen verbinden

In diesem Abschnitt soll für alle Schiene jedes anliegende Entity verbunden werden. Hierfür müssen aus der Entity Liste das entsprechende Entity herausgesucht werden. Dieses Problem unterteilt sich in zwei Teile. Zunächst wird eine unterstützende Matrix erstellt, auf die im zweiten Teil zurückgegriffen wird. Die Matrix ist intern ein 3-dimensionales Array. Für Darstellungszwecke wird diese als eine Tabelle repräsentiert, welche Listen enthält.

4.3.1 Matrix Vorbereitung

Das Ziel der Matrix ist es, Objekte, die nah beieinander liegen, näherungsweise nebeneinander anzuordnen. Efflektiv passiert eine Indexierung über die Koordinaten. In den beiden untenstehenden Bildern ist dies beispielhaft dargestellt. Die farbigen Lienen im linken Bild repräsentieren eine Zelle in der rechten Tabelle.

Abbildung 17: Kreuzungs Ausschnitt

matix	0	1	2	3
0	schiene(r= 5)	NULL	NULL	NULL
1	schiene(r= 1) signal(r=2)	schiene(r= 5)	NULL	signal(r=2)
2	schiene(r= 2)	schiene(r= 1) schiene(r= 2)	schiene(r= 5) schiene(r= 2)	schiene(r= 2)
3	signal(r=6)	NULL	schiene(r= 1)	schiene(r= 5) signal(r=6)

Abbildung 16: Ergebnis Matrix

Im rechten Bild ist ein Ausschnitt aus Factorio zu sehen. Dabei sind diverse in den vorhergehenden Kapiteln beschriebenen Entitys

dargestellt. Außerdem sind die einzelnen Kacheln markiert, sowie Gruppierungen von 2*2 Kacheln. Für Factorio Entitys ist es durchaus möglich, dass der Ankerpunkt an den exakt gleichen Koordinaten liegt. Daher ist notwendig, für eine Koordinate eine Liste an Entitys anzulegen.

Wenn nun die Factorio Koordinaten 1 zu 1 in ein Array übersetzt werden, entsteht das Problem, das viele Zeilen und Spalten in dem Array gänzlich leer wären, da die Schienen selbst nur ¼ des Array Platzes benutzen. Dies ist eine Folge des für das Schienensystem verwendeten 2 mal 2 Rasters. Daher wird der Matrix Koordinatenraum, um Faktor 2 in beiden Achsen reduzieren. Um zwischen Factorio Koordinaten und Matrixraum zu konvertieren wird immer durch 2 geteilt und abgerundet. Auch alle weiteren Zugriffe auf die Matrix erfolgen auf der Basis dieser Formel.

4.3.1 Schienen Referenzen erstellen

Der Rail-Linker verfolgt das Ziel, dass jedes Entity auf seine respektiven Nachbaren zeigt. Hierbei soll, um die nächsten Schritte zu vereinfachen, auch festgestellt werden, ob der Nachbar links oder rechts das ursprüngliche Entity liegt. Um dieses Ziel zu erreichen, wird auf die in Kapitel 4.1.2 definierten statischen Daten zugegriffen. Abbildung XYZ zeigt ein Datenfluss Diagramm, welches den groben Ablauf der Funktion darstellt, auf welche im Folgenden genauer eingegangen wird.

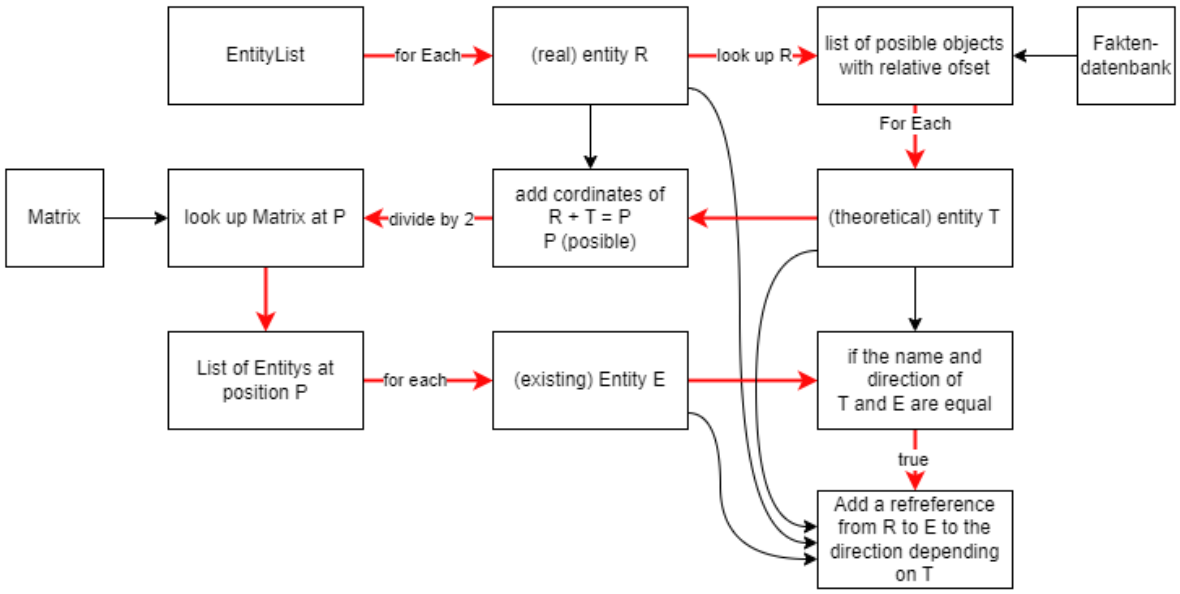

Abbildung 18: Rail linker Programmablauf

Das Rail-Linker System bekommt 3 Eingangsinformationen: (1) die Liste aller Entitys, (2) die statischen Daten der Faktendatenbank und (3) die eben erschaffene Matrix.

Im ersten Schritt wird über jedes Objekt in der Entity-Liste iteriert und für dieses Objekt (R) wird auf das Dictionary in der Faktendatenbank zugegriffen. Diese listet nun alle für R möglichen Entitys (T) auf. Die Entitys T geben einen relativen Abstand an, mit welcher die Position errechnet wird, an welcher sich ein benachbartes Entity befinden könnte. Hierfür werden T und R addiert, um P zu bekommen. P ergibt eine Koordinate (x, y) welche in der Matrix nachgeschlagen wird. Dies ergibt eine Liste an Objekten, welche sich an dieser Koordinate befinden. Diese Liste wird nun durchsucht nach dem Objekt (T), welches von der Faktendatenbank als möglich angegeben wurde. Sofern ein exakt passendes Objekt gefunden wird, so kann eine Referenz von dem originellen Objekt R zu dem nun gefundenen Objekt E erschaffen werden. Die Richtung, in welche das Objekt E eingetragen wird, ist hierbei abhängig von dem Objekt T. In der Faktendatenbank ist gespeichert, welche Richtung das Objekt T in Relation zum Objekt R hat. Für Signale wird außerdem noch eine Flag gesetzt, welche im Strecken ErstellungsTeil benötigt wird. Auf diese wird im Abschnitt 4.4.3 genauer eingegangen.

Das Ergebnis sieht dann als Beispiel so aus:

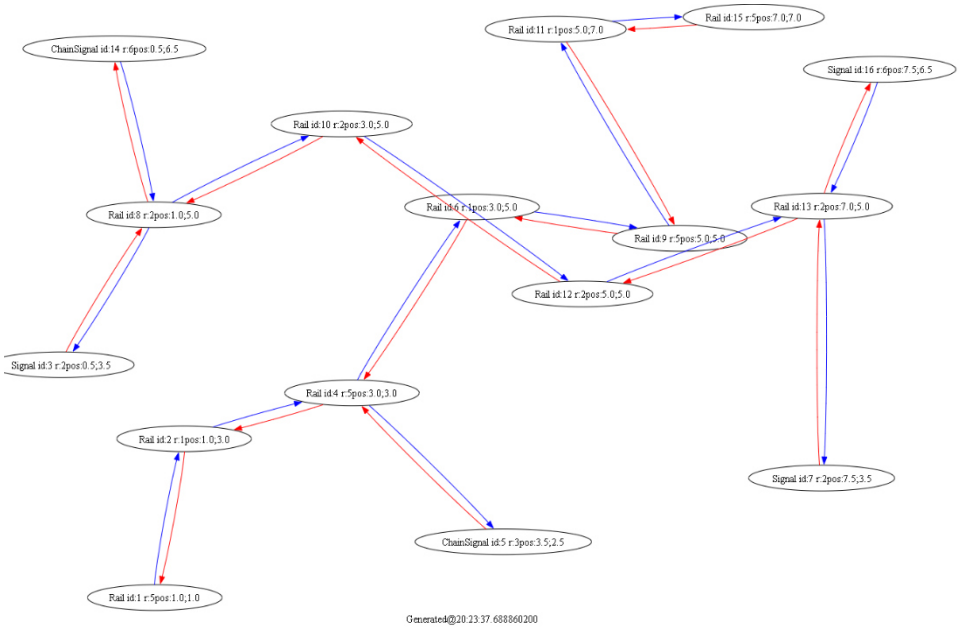

Abbildung 19: Grafviz Ergebnis nach Rail-Linker

Die Abbildung 19 zeigt das Teilergebnis des Programms, das durch Grafviz generiert wird. Als Eingabe für das Programm wurde die Kreuzung aus Abbildung 4.3.1.1 verwendet. In der Abbildung sind zum einen alle Entitys als Kreise repräsentiert, so wie die nun durch den Rail-Linker erschaffenen Verbindungen. Aus diesem Ergebnis ist nicht ersichtlich, wie dies mit dem Originalbild zusammenhängt. Dies ist der Ursache geschuldet, dass Grafviz nicht dafür ausgelegt ist, diese Daten anzuzeigen. Allerdings ist zu erkennen, dass dies 2 Hauptstränge sind: (1) einmal von links nach rechts und (2) einmal von unten links nach oben rechts geht, was wiederum die beiden Schienenstrecken aus dem ersten Bild repräsentieren. Hierbei ist anzumerken, dass dieses Ergebnis an der Y-Achse gespiegelt ist. Diese Ansicht, auch wenn sie für den Endnutzer nicht relevant ist, bot dennoch gute Möglichkeiten um dieses Projekt zu debuggen.

4.4 Strecken Zusammensetzen

Das System der Edge-Creation hat zum Ziel alle Strecken in dem Gesamtsystem zu finden und zu definieren. Dies bedeutet alle Stecken, welche von einem Zugsignal zu einem anderen Signal gehen sowie die Strecken, die vor einem ersten Signal und nach den letzten Signale liegen. Das nachstehende Flussdiagramm bietet eine Übersicht über die Funktionsweise der Streckenerstellung. Zunächst wird das Gesamtsystem erklärt, darauf folgen Details zu einzelnen Teilen des Systems.

4.4.1 Gesamtsystem

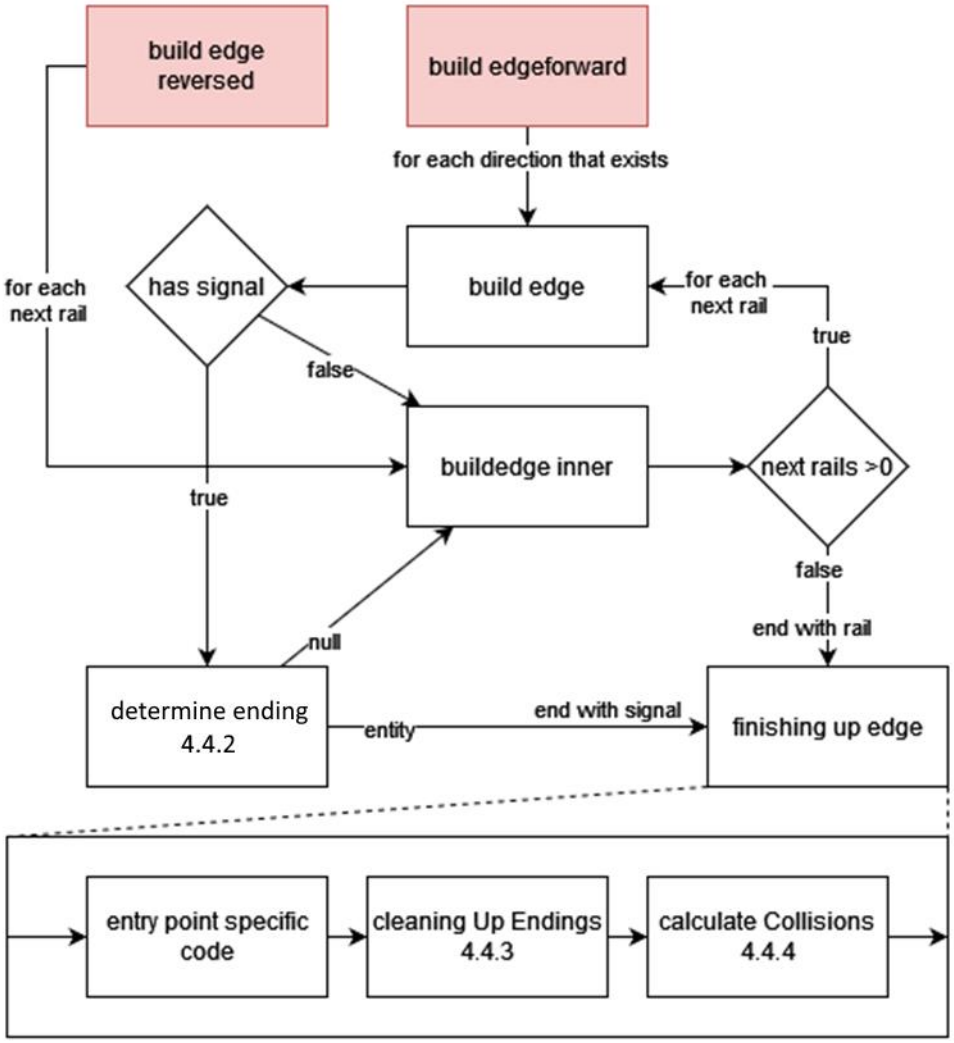

Abbildung 20: Programablauf vom Strecken Zusammensetzen

Zuerst oben in Abbildung 20 rot markiert sind 2 verschiedene Einstiegspunkte. Das Erstellen von Stecken rückwärts geschieht nach der Vorwärtssuche und wird benutzt, um alle Strecken vor einem ersten Signal zu finden. Alle anderen Strecken werden über den “normalen” Einstiegspunkt aufgerufen. Diese Funktion wird so für jeden Eintrag in der Liste an Signalen aufgerufen.

Im ersten Schritt wird geprüft, ob ein Signal an dem letzten Teil der Strecke steht. Sofern dies der Fall ist, wird keine detailliertere Prüfung gestartet. Diese wird in 4.4.2 besprochen. Andernfalls wird die innere, rekursive Funktion aufgerufen. Diese prüft zunächst auf den Special Case, welche zunächst in 4.1.3 besprochen wurde, welches unter Umständen die Suchrichtung invertiert. Weiterhin wird dann für alle gefundenen nächsten Schienen in der Richtung die ursprüngliche Funktion

aufgerufen. Sofern hier keine nächsten Schienen existieren, sind ist dies das Ende der Schienen und es wird mit einem virtuellen Signal beendet.

Sofern die Edge rückwärts gesucht werden muss, wird hier am Ende die Reihenfolge der gefundenen Schienen invertiert werden, bevor weiter gemacht werden kann, mit der gleichen Beendigung schritten. Dies ist zunächst das Aufräumen der ersten und letzten Schienen, welche unter Umständen entfernt werden, muss. Sowie darauf das berechnen der Kollision Linien. Diese beiden werden in ihren entsprechenden Abschnitten 4.4.3 und 4.4.4 besprochen.

4.4.2 Bestimmung des Endes

Sobald an einer Schiene ein Signal anhängt, wird die Funktion determining ending ausgelöst. Diese hat das Ziel, zwischen 3 Fällen zu unterscheiden. Zunächst prüft die Funktion, ob eines der betrachteten Signale relevant ist, sodass die Strecke an diesem Punkt beendet werden muss. Und sofern dies der Fall ist, ob sich das relevante Signal auf derselben Seite wie das Startsignal befindet, sodass diese Strecke valide ist.

Das Signale irrelevant sind kommt immer dann vor, wenn die erste Schiene einer Strecke betrachtet wird. Da diese Schiene basierend darauf hinzugefügt wurde, dass sie an einem Signal anliegt, wird dementsprechend auch diese Funktion ausgelöst. Dieser Fall muss trotzdem überprüft werden, da es möglich ist, mehrere Signale an eine Schiene zu stellen. Dadurch kann es ein relevantes, beendendes Signal gibt und die Strecke lediglich eine Schiene lang ist.

Insgesamt ergeben sich 31 verschiedene Fälle, wie Signale allein an den gebogenen Schienen positioniert sein können. Glücklicherweise ergibt sich, dass die Möglichkeiten für eine gerade Schiene ein Sub-Set der Möglichketen für gebogene Schienen sind. Es fallen Möglichkeiten weg, da Factorio bestimmte Kombinationen von Signalpositionen verhindert. So reicht es die Funktion für gebogene Schienen zu bauen.

Die 31 Edge Cases ergeben sich aus den möglichen Konstellationen. So können 1- bis 4 Signale an 4 möglichen Positionen sein, sowie 1 Signal an 2 möglichen Positionen das Start Signal sein. So ergeben sich 2^5-1 Möglichkeiten.

Dies kann als 5 Bits dargestellt werden, wobei das erste Bit angibt, ob das Start Signal in der Menge der Signale ist und die weiteren 4 Bits repräsentieren jeweils, ob an der dem Bit zugeordneten Stelle ein Signal ist. Diese 5 könnten nun in einem KV-Diagramm aufgelöst werden. Die Information das ein Start Signal existiert ist leider nicht aussagekräftig genug, das Ergebnis ist auch abhängig von der Position des Start Signals. Daher wird hierfür das Problem in 2 fälle gespalten, einmal ist das Startsignal anliegend zu der Schiene welche die determining ending

Funktion ausgelöst hat. Und im zweiten Fall liegt das Startsignal an einem beliebigen Punkt vor der auslösenden Schiene. Zunächst wird der einfachere zweite fall erklärt. Um beide Fälle zu erklären, müssen zunächst die 4 möglichen Signal Positionen definiert sein. Diese werden für die weiteren Diskussion wie folgt benannt:

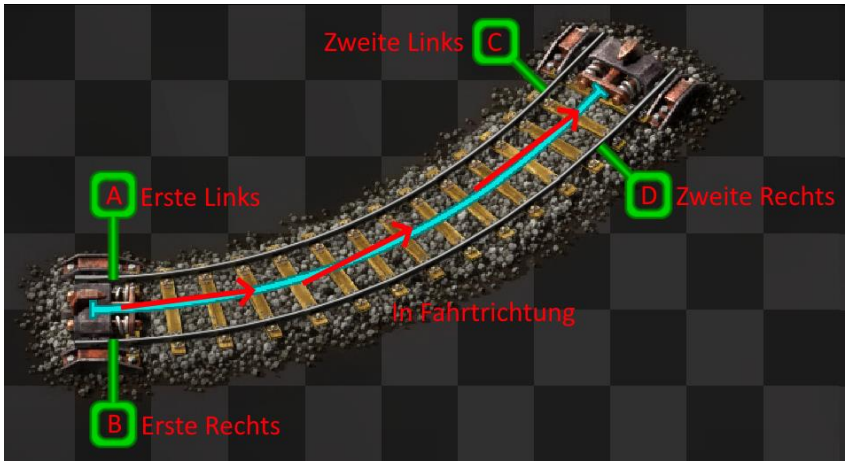

Abbildung 21: Kurvenschiene mit annotierten Signal Positionen

In der Abbildung XX zu sehen sind die 4 möglichen Signalpositionen an einer gebogenen Schiene. Die Idee ist nun, dass eine Position wie ein Boolean fungiert. Entweder ist dort ein Signal und somit ist die entsprechende Variable gesetzt oder nicht, dann ist diese null. Hiermit lässt sich nun ein KV-Diagramm aufstellen, welches angibt, in welcher Konstellation an Signalen welches beendende Signal zurückgegeben werden muss. Baut man dies auf, so erhält man die folgende Zuweisung:

		A		!A	
		B	!B	B	!B
	!D	B	A	B	NA
!C	D	B	A	B	D
	!D	B	A	B	C
C	D	B	A	B	D

Abbildung 22: KV-Diagramm 1: Signal vor der kurve

Hieraus ergeben sich dann 4 logische Formeln:

$$B \rightarrow B$$
$$A \wedge \neg B \rightarrow A$$
$$\neg A \wedge \neg B \wedge D \rightarrow D$$
$$\neg A \wedge \neg B \wedge C \wedge \neg D \rightarrow C$$

der Fall, dass alle false bzw. Null sind, ist hier ausgeschlossen da ansonsten die gesamte Funktion nicht ausgelöst werden sollte. Für die eigentliche Implementierung ist es nicht von Nöten die Bedingungen abzufangen, welche bereits vorher raussortiert wurden. Somit ergibt sich eine schlichte Priorität für die Reihenfolge B, A, D, C und die Funktion muss das erste Element zurückgeben welches nicht null ist.

		Y		!Y	
		C	!C	C	!C
	!A	C	!C	N	N
!X	A	C	!C	N	N
	!A	D	D	N	N
X	A	D	D	N	N

Abbildung 23: KV-Diagramm 2: Signal in der kurve

Für den Fall, dass das Start Signal in der Menge der Signale vorkommt, so ist sicher, dass das Startsignal per Definition in Fahrtrichtung ist.
Somit ist auch sicher das mindestens ein Signal in Fahrtrichtung existiert. Daher definieren wir 2 neue Variablen

$$X = C \wedge D$$
$$Y = C == StartSignal$$

Dabei bestimmt X ob es 1 oder 2 Signale auf der Fahrtseite gibt und Y definiert, ob das Startsignal an erster oder zweiter Stelle sitzt.

Dier Formeln die sich ergeben:

$$\neg Y \rightarrow null$$
$$Y \wedge X \rightarrow D$$
$$Y \wedge \neg X \wedge C \rightarrow C$$
$$Y \wedge \neg X \wedge \neg C \rightarrow null$$

diese Formeln lassen sich wieder stark vereinfachen. So muss Y nicht getestet werden, sofern es durch die vorherige Bedingung rausfällt. Auch die Letzten beiden können dadurch zusammengefasst werden, unter Beachtung das !C bedeutet das C = null ist und somit immer lediglich C zurückgegeben werden kann. Das Ergebnis sind dann 3 recht einfache Bedingungen.

```
if (secondRight == startSignal)
    null
else if (goodSide.size == 2) //
    secondRight
else
    secondLeft
```

!Y -> null

X -> D

_ -> C

Die Lösung des Problems ist nun recht simple trotz des zunächst komplexen Projekts. Nur ist es schwierig, den Code nachzuvollziehen ohne das Wissen, welches aus den KV-Diagrammen hervorgeht.

Für gerade Schienen funktioniert dies genauso, da hier die 4 Signale enger beisammen sind. Hier ist es für einen Spieler theoretisch möglich, Zustände zu erschaffen, welche für invalide Signale sorgen, allerdings ist dies in einem normalen Spiel nicht möglich und es kann davon ausgegangen werden, dass dem Spieler bewusst ist, was getan wurde.

Bei diagonalen Schienen erscheint es zunächst so, als könnte es Probleme ergeben, da diese nur 2 Signalpositionen haben und somit das Konzept von erster oder zweitem Signal nicht sinnvoll erscheint. Es ergibt sich allerding daraus, wie die statischen Daten gesetzt sind, dass innerhalb der Funktion beide Signale so betrachtet werden, als wären sie beide an erster Stelle oder beide an zweiter Stelle. Daher funktionieren Diagonale schienen mit dem gleichen System ohne extra code

4.4.3 Enden der Strecken aufräumen

Nachdem festgestellt wurde, welches der Signale die Strecke beendet, müssen nun die Enden der Entity Liste betrachtet werden. Unter bestimmten Bedingungen

werden im ersten Schritt mehr Schienen hinzugefügt als eigentlich zu der Strecke gehören. Dies liegt zu einen daran, dass beim Hinzufügen lediglich direkt angrenzenden Nachbarn betrachtet werden. Allerdings wurden bereits in Kapitel 2.1.8 zwei Signale beschrieben, die sich an derselben Schiene befinden und Strecken an zwei verschiedenen Stellen beenden. Dieses Problem ist in dem untenstehenden Bild (siehe Abbildung 24) verdeutlicht.

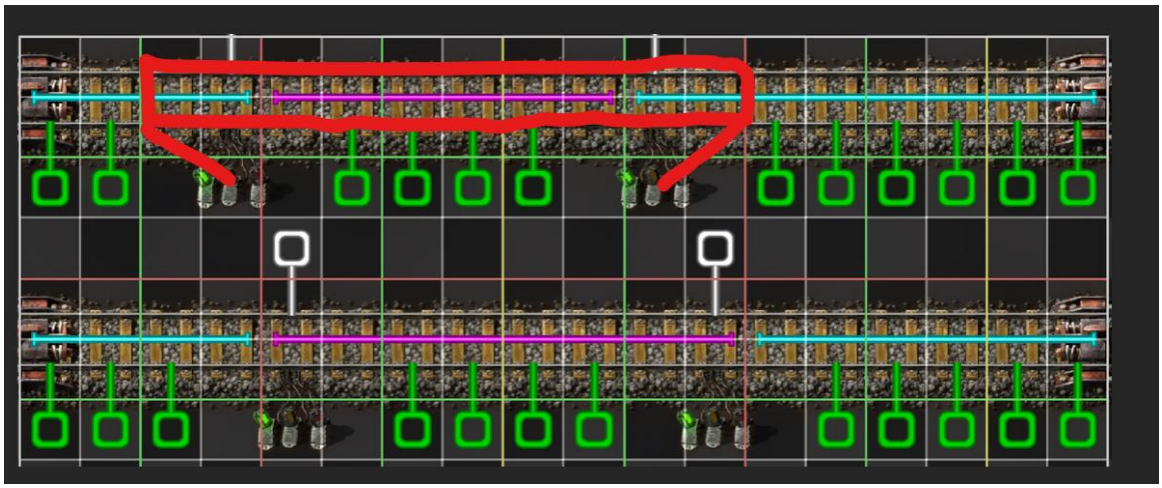

Abbildung 24: verschiedene Stecken längen bei Signalen mit gleichem abstand

In der Abbildung 24 zu sehen sind 2 Paare an Signalen. Dabei haben beide eine unterschiedliche Streckenlänge markiert durch die pinke Linie im Zentrum. In der oberen Reihe sind alle Objekte rot markiert, welche zu der Liste im ersten Schritt hinzugefügt werden. Dabei muss die erste und letzte Schiene entfernt werden, um zu dem korrekten Zielzustand zu kommen.

Ob eine Schiene entfernt werden muss, ist immer abhängig von der Position eines Signals relativ zur Schiene sowie welche Schiene in einer gegebenen Rotation vorhanden ist. Daher ist diese Information als eine Art “Default Fall” in der Faktendatenbank hinterlegt. Im Schritt des Rail-Linkers wird dann für jedes Signal der Wert aus der Faktendatenbank übernommen. Nur in äußerst spezifischen Fällen kann es vorkommen, dass zwei verschiedene Schienen für das gleiche Signal einen unterschiedlichen Fakt gespeichert haben. In der nachfolgenden Abbildung wird dieses Problem verdeutlicht.

Abbildung 25: Faktendatenbank Konflikt Sonderfall

In der Abbildung 25 ist rechts dargestellt, wie dieses Signal eine Diagonale bzw. eine Kurve beendet. Dabei gehört die Rot markierte schiene im diagonalen fall zur Blauen Strecke und im Kurven fall zur Pinken Strecke. Diese Spezifikationen sind nun auch in der Faktendatenbank abgelegt. Das Problem entsteht nun, wenn diese beide Fälle aufeinandertreffen. Dies zusehen ist im linken Teil der Abbildung 25. Dabei ist auch ersichtlich das die maskierte schiene Teil des Pinken Block ist bzw. den beiden pinken Strecken. Also ob eine schiene zu einer Strecke dazugehört, ist abhängig von anderen Schienen in der direkten nähe.

Die Lösung des Problems besteht darin das der Setter der variable zu einer Funktion ausgebaut wird. Wenn nun der Rail-Linker ein Signal als zu verbindendes Entity findet. So wird diese Funktion aufgerufen, wenn (Variablen aus Abbildung 18)
R ist eine Schiene mit einem Signal wie die in Abbildung 15 rot markiert wurde
E ist das echte anliegende Signal und
T ist das Äquivalent aus der Faktendatenbank.

Nun wird für E die variabel “removeRelatedRail” in Abhängigkeit von R, E und T. Die Funktion ist in dem nächsten Struktogramm dargestellt. In diesem Diagramm wird RRR als Abkürzung für “removeRelatedRail” verwendet:

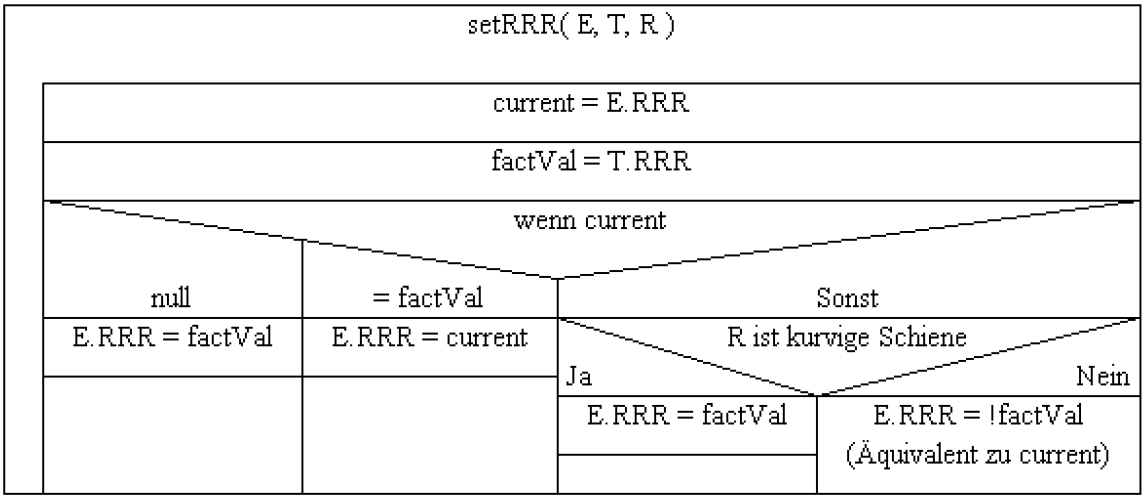

Abbildung 26: Struktogramm zum setzten von remove Related Rail

Das Setzen von RRR funktioniert wie folgt: Sofern noch kein Wert gesetzt wurde, wird der Wert aus der Faktendatenbank übernommen. Sofern der bereits gesetzte wert dem des Faktes übereinstimmt, so gibt es keinen Konflikt. Ansonsten wenn der aktuelle Wert und der Faktwert sich widersprechen, so wird der Wert welcher in Faktendatenbank an der Kurve liegt.

Dies wurde noch nicht erwähnt, aber grundsätzlich treten solche Konflikte nur bei Kurven auf, welche auf Diagonale treffen und es gilt, auch immer das der Faktwert der Kurve überwiegt. Dadurch das der Wert von RRR innerhalb des Signals gesetzt wird und alle Stecken lediglich eine Referenz auf das gleiche Signal halten, verhalten sich die Strecken gleich, wenn es darum geht, die Streckenenden anzupassen.

4.4.4 Berechnungen der Kollisionslinien

Zuletzt werden vorbereitende Berechnungen vorgenommen, dass diese in dem direkt nachfolgenden Schritt benötigt wird. Dies geschieht jedoch immer mit Abschluss einer Edge.

Ziel dieser Funktion ist es, eine Liste an Schienen auf eine minimale Anzahl an Punkte zu reduzieren. Wobei diese Punkte dem Streckenverlauf abstrahiert folgen. Zunächst dargestellt ist ein Input-Output-Paar:

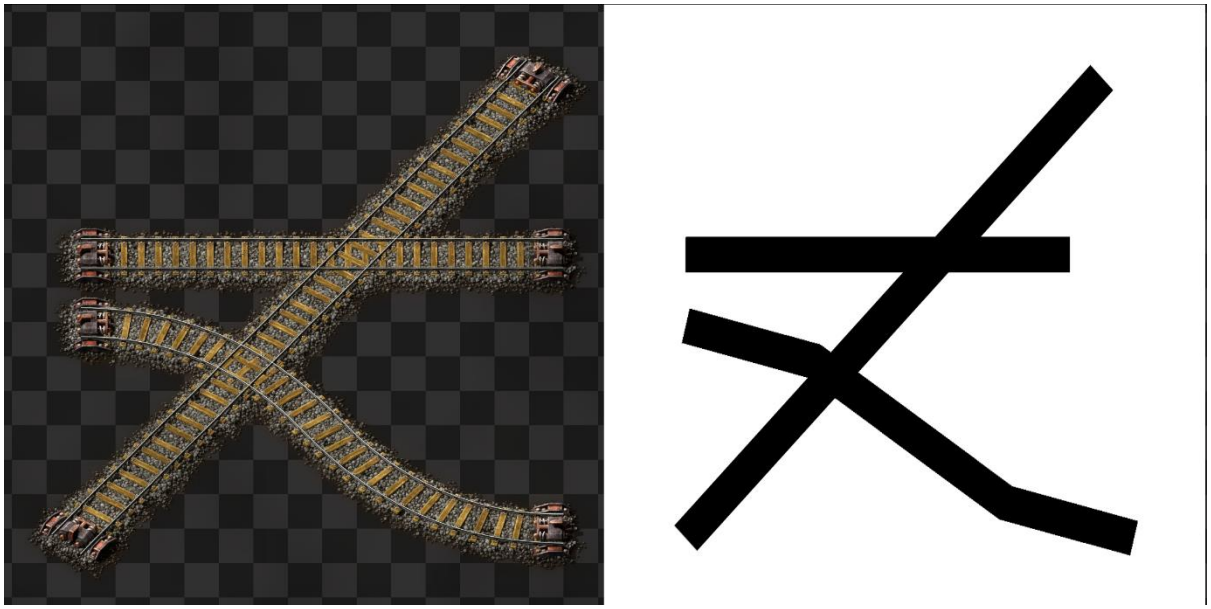

Abbildung 27: Beispiel Input und Ergebnis der Berechnungen für Kollisionslinien

In Abbildung 27 ist rechts ein Ausschnitt aus Factorio zu sehen und links die resultierenden Kollisionslinien. Dabei bestehen die Lienen aus einer Liste an Punkten, welche aus den Koordinaten der ursprünglichen Entitys errechnet werden. Ziel ist es eine, Strecke so minimal wie möglich darzustellen, ohne relevante Details zu verlieren. Daher werden für gerade Strecken lediglich 2 Punkte gespeichert, also Anfang und Ende. Kurven werden mit 3 Punkten repräsentiert; Start, Mitte und Ende der Kurve. Duplikate werden übersprungen. Abschließend werden die Enden der Linien um ein kleines Stück gekürzt, um Kollisionen von aufeinanderfolgenden, durch Signale getrennte Stecken zu verhindern.

Für dies System wird erneut auf eine in der Faktendatenbank gespeicherte Variable zurückgegriffen. Diese gibt für jede Schienenvariante 2 Punkte an, welche die Linie dieser Schiene definieren. Weiterhin wird innerhalb des Systems zum einen auf eine Hilfsfunktion zurückgegriffen, welche für 2 Punkte bestimmt, welcher näher an einem dritten Punkt liegt.

Von diesen Definitionen ausgehend wird in der Generierung das erste und letzte Entity der Strecke betrachtet sowie jede dazwischen liegende gebogene Schiene.

Zur Bestimmung des ersten Punktes wird die erste Schiene und das Startsignal verwendet. Von den 2 Faktenpunkten wird der nähere zu dem Startsignal als erster Punkt der Kollisionslinie verwendet.

Im nächsten Teil werden nun für jede Kurve die gleichen Schritte durchgeführt: Zuerst werden der Start und Endpunkte mit der Faktendatenbank berechnet. Darauf wird der Punkt bestimmt, welcher auf einer horizontalen, vertikalen oder diagonalen Geraden zum letzten Punkt ausgerichtet ist. Dieser wird zunächst angefügt bevor der ankerpunkt und der andere punkt folgt.

Für den letzten Punkt wird die letzte Schiene in der Liste verwendet, um für ihre Faktenpunkte den am weitesten entfernten Punkt von dem vorherigen Punkt zu bestimmen und hinzuzufügen.

Hiernach müssen lediglich die Enden gekürzt werden. Dies übernimmt gänzliche eine Funktion welche A und B nimmt und A’ modifiziert zurückgibt so das A’ um 0.1 näher an B ist als A. Dafür werden zunächst 2 Punkte generiert welche auf der entsprechenden Achse einmal +0.1 und zum anderen -0.1 versetzt wurden. Von diesen beiden Möglichkeiten wird der gewählt, welcher näher an B ist, umso A‘ zu bestimmen.

4.5 Blöcke zusammenfassen

Dieses Teilsystem hat das Ziel aus der Liste an Strecken eine Liste an Blöcken zu formen, wobei jeder Block wiederum Referenzen auf die Strecken hat, die zu diesem Block gehören. Dabei kommen die im vorherigen Abschnitt berechneten Kollisionslinien zum Einsatz.

Für das Zusammenfassen ist der Fakt relevant, dass jede Strecke zu genau einem Block gehört. Außerdem gilt, das Strecken, welche in zu einem Block gehören mit mindestens einer anderen Strecke des Blocks kollidieren und sich zwischen allen Mitgliedern eines Blocks eine Verwandtschaft an Kollisionen ergibt.

Mit einem vergleichbar einfachen Algorithmus werden die Strecken zu Blöcken gruppiert. Eine Übersicht bietet das untenstehende Struktogramm:

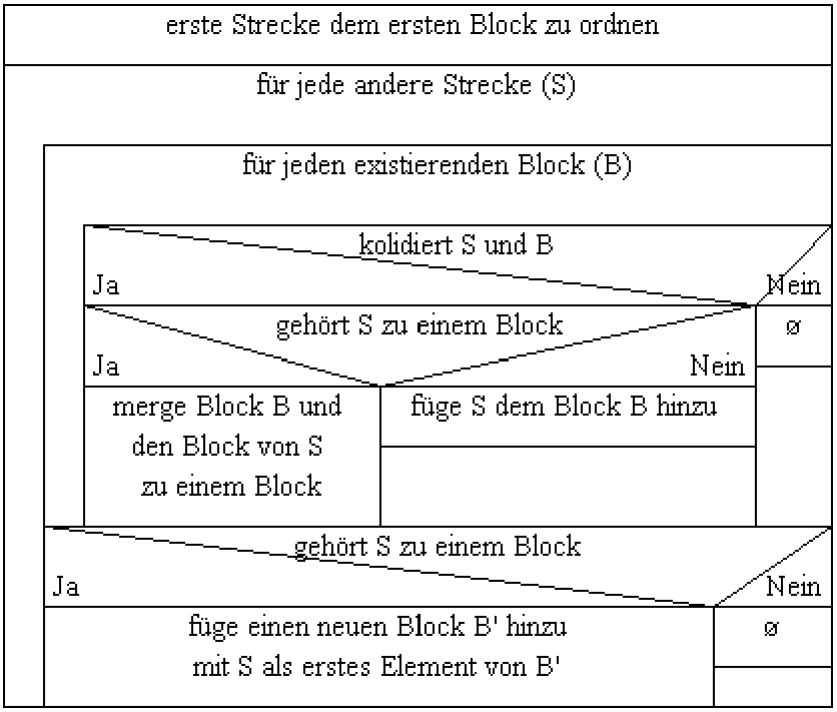

Abbildung 28: Struktogramm zum Ablauf von Block Gruppierung

Zunächst wird die erste Strecke einem ersten Block zugeordnet. Danach wird für jede Strecke die Sortierung ausgeführt. Hierfür wird zunächst für jede Strecke S geprüft, ob diese mit einem existierenden Block B kollidiert. Sofern S mit genau einem Block kollidiert, wird diese dem Block hinzugefügt. Es ist allerdings möglich, dass S mit mehreren Blöcken kollidiert. In diesem Fall sind diese mehreren Blöcke ein zusammenhängender Block und werden dann verbunden. Sofern nach dem Durchgehen aller existierenden Blöcke keine Kollisionen gefunden wurde und somit S immer noch blocklos ist, so wird ein neuer Block mit der Strecke als erstes Element erstellt.

4.6 Tiernan Anwendung

Die Implementierung des Tiernan Algorithmus orientiert sich ziemlich strikt an seiner theoretischen Definition. Dabei existiert zum einen das Objekt Graf, um die verschiedenen Variablen des Algorithmus zu halten. Die Implementierung des Algorithmus funktioniert auf das Interface Grafabel und kann so funktionieren für Strecken und Blöcke. Der Funktion übergeben wird zum einen eine Liste an Grafabel

Objekten sowie eine Funktion, welche für jedes Grafabel eine Liste an direkten Nachbarn angibt.

Durch Aufruf und Anwendung des Tiernan Algorithmus entsteht zunächst eine Liste aller elementaren zirkulare Abhängigkeiten. Diese Liste erweitern wir mit jeder Kombination an möglichen zirkularen Abhängigkeiten. Dies verdeutlicht folgendes Beispiel:

Wir nehmen an, dass zwei zirkulare Abhängigkeiten A und B existieren, die Zahlen Nodes sind, welche die zirkulare Abhängigkeit bilden:
A = [1, 2, 3, 4, 5] B = [3, 6, 7,4]
So kann auch eine Union von A und B erzeugt werden mit
C = [1, 2, 3, 6, 7, 4, 5]

Nach dem die elementaren zirkularen Abhängigkeiten um ihre Kombinationen erweitert wurden, werden alle jene zirkularen Abhängigkeiten wegelassen, welche ausschließlich aus Kettensignalen bestehen. Die restlichen zirkularen Abhängigkeiten können nun genauer analysiert werden, um festzustellen, ob diese auch einen echten Dreadlocks bilden können.

4.7 Analyse der zirkularen Abhängigkeiten

Das System der Analyse bekommt als Eingabe immer genau eine zirkulare Abhängigkeit und gibt zurück einen Boolean welcher angibt ob diese zirkularen Abhängigkeiten ein Deadlock ist. Eine zirkulare Abhängigkeit, welche übergeben wird, besteht aus einer Liste an Blöcken.

Danach baut die Analyse auf dem Konzept einer Gesamtstrecke auf wie in Abschnitt 2.1.6 definiert. Damit eine zirkulare Abhängigkeit ein Deadlock ist muss jede Gesamtstrecke, welche Teil der zirkularen Abhängigkeit ist, mindesten ein normales Signal innerhalb des Deadlocks aufweisen.

Für die Lösung des Problems wird zunächst eine Hilfsfunktion getEdge definiert und erläutert.
GetEdge nimmt als Input 3 Blöcke: V, I, N, und gibt zurück eine oder zwei Strecken, welche im Block I ist und eine Verbindung von V nach N darstellt.

Diese Funktion gibt 1 Edge zurück, sofern es genau eine klare Ergebnisstrecke existiert. In dem Fall, dass zwei Strecken benötigt werden, um vom Block V nach N über I zu kommen ist der Block I der Punkt, wo von einer Gesamtstrecke zu einer anderen Gesamtstecke gewechselt wird.

Diese Hilfsfunktion wird zunächst für alle Blöcke aufgerufen, wobei I der aktuelle Block, V die vorherige und N der nachfolgende Block ist. Alle Rückgabewerte werden in eine Liste geschoben diese beinhaltet alle Strecken welche Teil der zirkularen Abhängigkeit sind. Jedes Mal, wenn die Funktion zwei Strecken zurückgibt, wird der Index in eine weitere Liste aufgenommen. Diese Indexliste spaltet nun die Streckenliste in die Gesamtstecken. Darauf wird die Streckenliste rotiert, so dass der Anfang einer Gesamtstrecke bei 0 liegt, auch werden die Indices entsprechend versetzt.

Nun kann mit Hilfe der Indexliste jeweils Bereiche der Streckenliste überprüft werden, ob jede Gesamtstrecke ein Teil der zirkularen Abhängigkeit ist. Sofern alle ein normales Signal aufweisen, wird diese zirkulare Abhängigkeit als Deadlock festgestellt. Denn nun ist möglich für Züge in die zirkulare Abhängigkeit so einzufahren, dass diese sich mehr blockieren.

4.8 Parameter für die CLI

Um dem Nutzer Möglichkeiten der Konfiguration des Programms zu geben, wurden für die CLI Eingabe Flags erstellt. Diese Flags können gesetzt werden, mehr Informationen zu erhalten und andere Features zu konfigurieren.

Um zusätzliche Diagramme zu erhalten, kann der User <-g> nutzen. Dadurch erhält er zusätzlich zum normalen Output Diagramme die Mithilfe von Graphviz erstellt wurden. In diesen Diagrammen werden interne Datenstrukturen des Programms abgebildet, weshalb diese Option besonders fürs Debuggen nützlich ist. Als externes Programm hat Graphviz eine lange Runtime besonders weil die Diagramme abhängig von der Größe der Kreuzung stark wachsen. Deshalb ist diese Ausgabe standardmäßig deaktiviert.

Weitere Debuginformationen kann der Nutzer mithilfe von <-d> erhalten. Dadurch werden die Debugstatements angezeigt, die bei normalem Programmablauf

versteckt bleiben. Diese Daten sind nur nützlich für Nutzer, die sich mit dem Code auskennen, ansonsten überfüllen sie den Bildschirm.

Das Diagramm, das dem User beim Ausführen angezeigt wird, kann mithilfe von <-i> deaktiviert werden. Dann wird es nur als Datei abgespeichert und nicht sofort geöffnet. Diese Option ist nützlich, wenn das Diagramm mit einem anderen Programm als dem Standardprogramm angezeigt werden soll.

Das Diagramm hat zufällig gewählte Farben für unterschiedliche, von Signalen getrennte, Strecken. Diese Funktion kann deaktiviert werden, wenn sie nicht benötigt wird, um das Diagramm dadurch übersichtlicher zu machen.

5 Kritischer Rückblick

In diesem Kapitel wird die Implementierung des Pilotprojektes kritisch angeschaut und Verbesserungsvorschläge gegeben. Es werden die Algorithmen bewertet und ein Zukunftsausblick gegeben.

5.1 Bidirektionale Schienensysteme

In Factorio geben die Signale die Fahrtrichtung an. Dadurch können Züge Strecken nur in eine Richtung befahren. Wenn nun zwei Signale genau gegenüber voneinander gesetzt werden, können die Züge in beide Richtungen fahren. Genaueres ist in Kapitel 2.1.8 “Verhalten von Signalen” erklärt.

Ein gerichteter Graph von einem bidirektionalen Schienennetz würde zum Beispiel so aussehen.

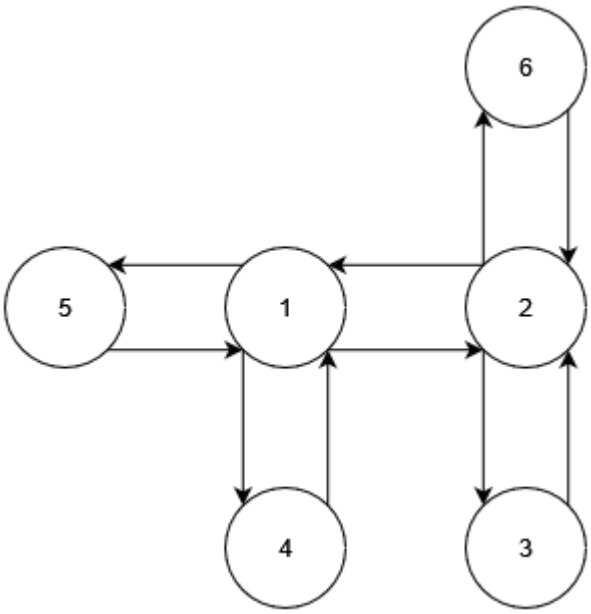

Abbildung 29: Bidirektionaler Graph

Dies entspricht dem Graph aus Kapitel 2.4, wenn das Schienennetz in Kapitel 2.3 bidirektional wäre. Es ist zu erkennen, dass nun ganz viele zirkulare Abhängigkeiten existieren. Viele davon führen zu Deadlocks, wenn zwei Züge genau aufeinander zufahren. Zum Beispiel könnte ein Zug in Block 5 darauf warten, dass der Zug in Block 1 diesen verlässt. Der Zug in Block 1 wartet darauf, dass der Zug in Block 5 diesen verlässt. Sie stehen sich genau gegenüber und ein Deadlock ist entstanden. Diese zirkularen Abhängigkeiten können nicht von denen unterschieden werden, die durch Designfehler im Schienennetz existieren.

Aus diesem Grund kann das Pilotprojekt bidirektionale Schienennetze nicht analysieren. Zeitlich war es nicht möglich eine Lösung für dieses Problem zu finden, daher wurde das Analysieren von eindirektionalen Schienennetzen priorisiert.

Es ist unklar, ob für das Analysieren von bidirektionalen Schienennetzen ein anderer Ansatz gewählt werden muss oder ob das Pilotprojekt angepasst werden kann, um auch nützliche Ergebnisse für ein bidirektionales Schienennetz zu liefern.

5.2 Tiernan Algorithmus

Der implementierte Algorithmus zum Finden der zirkularen Abhängigkeiten ist der Algorithmus von Tiernan. Der Algorithmus skaliert in O-Notation O(n* e(c + 1)).

Wobei n die Anzahl an Knoten, e die Anzahl an Verbindungen und c die Anzahl an Kreisen ist [11].

Da der Tiernan Algorithmus ein alter und simpler Algorithmus ist gibt es verbesserte Versionen und andere Algorithmen mit besseren Skalierungen. Ein Beispiel dafür ist der Algorithmus von Johnson [11]. Er skaliert in O-Notation O((n + e)(c + 1)), was um einiges besser ist als der von Tiernan[11].

Dadurch, dass der Algorithmus von Tiernan simpler ist, kann er auch leichter auf den Anwendungsfall angepasst werden. Als Pilotprojekt ist die Funktionalität die wichtigste Komponente. Da das Programm mithilfe des Tiernan funktioniert und die Anpassung auf einen anderen Algorithmus nicht in den Zeitrahmen passt, muss die etwas schlechtere Performance hingenommen werden.

5.3 Factorio 2.0

Das Spiel Factorio erhält Updates vom Entwickler. Es ist ein Update angekündigt, in dem das Schienensystem überarbeitet wird [15]. Dabei gibt es zwei große Änderungen, durch die das Pilotprojekt nicht mehr kompatibel sein wird. Es wird der Kurvenradius erhöht, wodurch die Verbindungspunkte der Schienen sich verschieben. Durch diese Änderung wird es halbe Kurven und zusätzliche Richtungen für die diagonalen Schienen geben. Die zweite große Änderung sind die Stützschienen. Das sind Schienen, die nicht auf dem Boden verlaufen, sondern von Stützen getragen werden und damit ohne Kreuzung über anderen Schienen navigieren können.

Für beide Änderungen muss die Fakten-Datenbank angepasst werden und die neuen Entitys aufgenommen werden bzw. alte entfernt oder umgestaltet werden. Außerdem muss die Kollisionsbox-Berechnung angepasst werden, zum einen an die neuen Schienenformen, zum anderen an die dritte Dimension der Stützschienen. Es kann sein, dass noch an anderen Stellen Anpassungen gemacht werden müssen, damit das Pilotprojekt wieder kompatibel ist, aber die beiden oben genannten Änderungen werden am meisten Arbeit erfordern.

Durch das Hinzufügen von Stützschienen, wird es einfacher werden, Kreuzungen ohne Deadlocks zu designen. Kreuzungen mit Stützschienen können so geplant werden, dass nur Abzweigungen vorhanden sind. Die meisten Kreuzungen können durch die Stützschienen vermieden werden. Diese Kreuzungen führen schnell zu

zirkularen Abhängigkeiten, da dort Züge aus unterschiedlichen Richtungen aufeinander warten. Bei Abzweigungen kommen die Züge aus derselben Richtung, wodurch sie nicht so schnell zu zirkularen Abhängigkeiten werden. Trotzdem kann es in einer Kreuzung mit Stützschienen zu Deadlocks kommen. Wenn zum Beispiel ein Kreis mit den Schienen gebaut wird, gibt es wieder eine zirkulare Abhängigkeit, die zu einem Deadlock führen kann. Außerdem sind Deadlocks wegen zu vieler Züge möglich. Wenn in jedem Block ein Zug ist, so kann kein Zug in den nächsten Block fahren und es ist ein Deadlock vorhanden.

6. Fazit

Durch das Pilotprojekt konnte gezeigt werden, dass Deadlocks automatisch erkannt werden können. Allerdings ist das Pilotprojekt nicht umfassend. Auf die Analyse von bidirektionalen Schienennetzen wurde nicht eingegangen. Durch Zeitlimitationen konnte diese Funktionalität nicht sichergestellt werden und die Priorisierung wurde auf ein Pilotprojekt gelegt, das eine Teilfunktionalität besitzt. Es erscheint aber möglich, dies in einem Folgeprojekt zu erweitern.

Weitere angekündigte Funktionserweiterungen des Spiels Factorio scheinen die Wahrscheinlichkeit von Deadlocks herabzusetzen, machen aber eine Analyse immer noch nicht überflüssig. Anpassungen des Projektes an diese Funktionserweiterungen scheinen ebenfalls möglich zu sein.

Quellen

[1] Factorio Wiki, Blueprintbook https://wiki.factorio.com/Blueprint_book 10.04.2024

[2] Jacob Kohlruss, Untersuchung von Methoden zur Vermeidung von Deadlocks in synchronen Eisenbahnsimulationsprogrammen, 10.03.2024

[3] Christian Baun, Betriebssysteme kompakt, Seite 215, 12.04.2024

[4] Matthias Künzer, Gerichtete Graphen, Kapitel 3, 11.04.2024

[5] JC Tiernan - Communications of the ACM, 1970 -An efficient search algorithm to find the elementary circuits of a graph (acm.org) 15.03.2024

[6] Prof. Dr. Karl Stroetmann, An Introduction to Artificial Intelligence, Kapitel 2.2 S.15, 20.03.2024

[7] Factorio Forum, https://forums.factorio.com/viewtopic.php?f=93&t=49689 , 23.03.2024

[8] Factorio Dokumentation API Docs | Factorio , 14.04.2024

[9] Factorio Documentation LuaEntity - Runtime Docs | Factorio ,14.04.2024

[10] Factorio Forums Add multithreading/parallelization options - Factorio Forums ,15.04.2024

[11] Donald B. Johnso, Finding all the elementary Circuits of a directed Graph, 15.04.2024

[12] Google Gson GitHub - google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back , 02.04.2024

[13] Graphviz Webseite https://graphviz.org/

[14] marcouberti ZLIB compression and decompression in Kotlin / Android · GitHub ,1.11.2023

[15] Factorio Blog Friday Facts #377 - New new rails | Factorio , 13.03.2024

[16] https://www.mdpi.com/2079-9292/13/7/1377 ,15.04.2024

[17] https://link.springer.com/chapter/10.1007/978-3-031-49179-5_20 ,15.04.2024

[18] https://dl.acm.org/doi/abs/10.1145/3449726.3459463,15.04.2024