Vergleichende Analyse von A*-Suche und Constraint Programming zur Lösung des Rush-Hour-Puzzles

Bachelorarbeit

Jana Kristina Wüsten
468696

21. Februar 2025

Gutachter: Prof. Dr. Benjamin Blankertz
Dr. Stefan Fricke

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Neurotechnologie

Kurzfassung

Die vorliegende Arbeit befasst sich mit der effizienten Lösung des populären Schiebepuzzles Rush Hour und untersucht dabei verschiedene Lösungsansätze hinsichtlich ihrer Performanz. Das Hauptziel dieser Arbeit ist es, herauszufinden, welcher Algorithmus in der Lage ist, das Puzzle sowohl mit der minimalen Anzahl an Zügen als auch in möglichst kurzer Zeit zu lösen. Die zentrale Frage lautet daher: *Welcher Ansatz ist besser geeignet, um das Rush-Hour-Puzzle unter verschiedenen Bedingungen effizient zu lösen?* Um diese Frage zu beantworten, werden zwei grundlegende Lösungsansätze verglichen: Zum einen wird der klassische algorithmische Ansatz der A*-Suche mit unterschiedlichen Heuristiken untersucht. Zum anderen wird ein alternativer Ansatz durch Constraint Programming mit dem Theoremlöser Z3 vorgestellt, wobei das Puzzle als System von Constraints modelliert wird.

Die Methodik umfasst die Implementierung beider Lösungsansätze und deren Evaluierung anhand relevanter Benchmarks, um die Effizienz und die Lösungsgeschwindigkeit zu messen. Die Ergebnisse zeigen, dass, unter realistischen Spielszenarien ohne Vorkenntnisse über die Schwierigkeit einer Instanz, der Constraint-Programming-Ansatz im Allgemeinen stabilere und schnellere Ergebnisse liefert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Rush Hour

Bei Rush Hour handelt es sich um ein Schiebepuzzle, das bereits in den 1970er Jahren von dem Japaner Nob Yoshigahara erfunden wurde. Es wird als physisches Puzzle seit 1996 von dem US-amerikanischen Spielehersteller ThinkFun Inc. vertrieben, gleichzeitig existieren diverse Derivate von Rush Hour als Smartphone- oder Web-Apps, die teilweise auch unter anderen Namen wie Unblock Me[1] oder Move the Block[2] zu finden sind.

1.1.1 Spielfeld und Regeln

Ein klassisches Rush-Hour-Puzzle ist in Abbildung 1.1 gezeigt. Es besteht aus einem 6×6-Spielfeld mit verschiedenfarbigen Blöcken bzw. im Kontext der Spielthematik Fahrzeugen und einem roten Block bzw. Auto. Die Fahrzeuge nehmen unterschiedlich viele adjazente Spielfelder ein; Fahrzeuge, die zwei benachbarte Felder auf dem Spielbrett in Anspruch nehmen, werden als Autos bezeichnet, wohingegen Fahrzeuge, die drei Felder umfassen, als LKW gelabelt werden. Jedes Fahrzeug muss entweder horizontal oder vertikal platziert werden, wobei die Ausrichtung eines Fahrzeuges im Verlauf einer Spielrunde unveränderlich ist. Das rote Auto ist immer horizontal ausgerichtet und befindet sich in allen klassischen Rush-Hour-Konfigurationen in der dritten Reihe des Spielfeldes. Horizontal ausgerichtete Fahrzeuge können in einem Zug nur nach rechts oder links bewegt werden, vertikale analog nur nach oben oder unten. Hierbei können sie nicht auf oder über Felder bewegt werden, die bereits von anderen Fahrzeugen in Anspruch genommen werden.

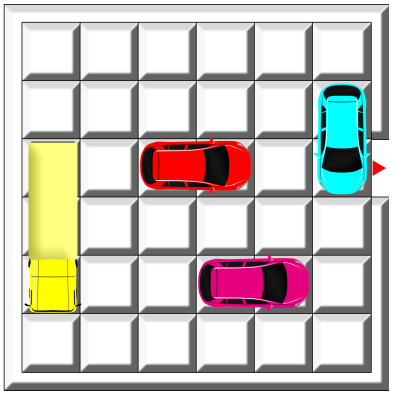


Abbildung 1.1: Mögliche Startkonfiguration von Rush Hour mit drei Autos und einem LKW.

Spielziel von Rush Hour ist es, das rote Auto zum Ausgang zu bewegen, indem die anderen Fahrzeuge so verschoben werden, dass der Weg zwischen dem roten Auto und dem Ausgang frei ist. Bei

besagtem Ausgang handelt es sich um eine Öffnung am rechten Spielfeldrand, die in derselben Reihe wie das rote Auto platziert ist. Ein sekundäres Spielziel, das im Rahmen dieser Arbeit allerdings eine entscheidende Rolle einnimmt, ist die Lösung dieses Problems in möglichst wenigen Zügen. Als Maß für die Schwierigkeit eines Puzzles wird die benötigte minimale Zuganzahl der Literatur folgend angenommen, denn „the minimum number of moves appears to be the only reliable indicator of difficulty“[3, S. 29].

1.1.2 Interessante Startkonfigurationen

Die physische Version des Rush-Hour-Puzzles kommt mit 40 initialen Konfigurationen in Form von Karten; es existieren jedoch 476.118 interessante Startkonfigurationen, die in der Rush-Hour-Datenbank von Michael Fogleman[4] eingesehen werden können. Als „interessant“ werden dabei alle Konfigurationen wie in Abbildung 3.1 betrachtet, die

1. keine überflüssigen Fahrzeuge enthalten (überflüssig sind solche Fahrzeuge, deren Entfernen an der Schwierigkeit und Länge der Lösung nichts verändert),
2. keine Folge von Fahrzeugen enthalten, die eine ganze horizontale oder vertikale Reihe des Spielfeldes blockieren und somit nicht bewegt werden können,
3. maximal schwierig sind, d.h. kein Fahrzeug kann mehr in einer Weise bewegt werden, die die Lösung erschwert und
4. keine anderen horizontalen Fahrzeuge rechts vom roten Auto zulassen[5].

(a) Puzzle mit 6 Zügen. (b) Puzzle mit 18 Zügen. (c) Puzzle mit 31 Zügen.

Abbildung 1.2: Auswahl interessanter Rush-Hour-Startkonfigurationen.

1.2 Literaturbesprechung

Das Puzzle Rush Hour ist in der Forschung im Bereich Informatik kein unbekannter Gegenstand, sondern wurde an verschiedenen Stellen bereits untersucht. So haben Flake und Baum schon 1999

gezeigt, dass das generalisierte Rush-Hour-Puzzle PSPACE-vollständig ist[6]. Bei der generalisierten Variante von Rush Hour handelt es sich um eine „variant of the original game with two simple modifications. First, the grid size can be a rectangle of arbitrary width and height. Second, the exit for the target car can be at any location on the perimeter of the grid (Original zitiert mit Anpassungen)“[6, S. 897]. Doch nicht nur das generalisierte Rush-Hour-Puzzle ist PSPACE-vollständig. Hearn und Demaine[7] zeigen einige Jahre später, nämlich im Jahre 2005, dass sogar die einfachste klassische Rush-Hour-Konfiguration mit Autos und LKW PSPACE-vollständig ist.

Vor dem Hintergrund der PSPACE-Vollständigkeit des Rush-Hour-Puzzles wundert es nicht, dass das Spiel als eine Art Benchmark für eine Reihe verschiedener algorithmischer Lösungsansätze genutzt wurde. Ein naiver Lösungsweg für Rush Hour ist die Breadth-First-Search bzw. Breitensuche. Sie ist im Kontext des Puzzles trotz ihrer Simplizität „demonstrated to be efficient“[8, S. 78]. Hauptman et al. nutzen in ihren Arbeiten außerdem einen IDA*-basierten Solver zur Lösung des Puzzles. Sie stellen dabei fest, „that the improvement attained with heuristics increased substantially when evolution entered into play“[9, S. 961] und brachten damit die Sinnhaftigkeit genetischer Algorithmen für die Lösung des Rush-Hour-Puzzles in den Diskurs ein. Bei der IDA*- und der A*-Suche führten die Heuristiken zuvor zu ambivalenten Ergebnissen[9]. Auch Schmid, Dornberger und Hanne bemerken, dass bisher keine perfekte Heuristik zur Lösung von Rush Hour mit heuristischen Suchalgorithmen existiert[8].

Eine weitere in der Literatur auftretende Lösung für das Spiel ist die Nutzung von Constraint Programming und Answer-Set-Programming. So veröffentlichen Cian, Dreosse und Dovier eine Modellierung von Rush Hour in der Constraint-Modellierungssprache MiniZinc. Auch eine Modellierung im Kontext von Answer-Set-Programming wird von ihnen vorgestellt. Abschließend nutzen Cian et al. vorrangig den Constraint-Solver Chuffed und die ASP-Löser Clingo und DLV, um eine Auswahl von Instanzen des Rush-Hour-Puzzles zu lösen. Sie kommen dabei zu dem Ergebnis, dass die ASP-Solver bei Instanzen mit maximal 30 Zügen in der minimalen Lösung höchst performant sind. Der Constraint-Programming-Ansatz ist im Speziellen bei Instanzen mit mehr Zügen erfolgreich. Hier können je nach gewähltem Solver Lösungen in unter einer Sekunde gefunden werden; bei denselben Spielinstanzen und einem Lösungsversuch mit ASP-Programming kommen indes Laufzeiten von ca. 20 Minuten durchaus vor[10].

1.3 Über diese Arbeit

Die vorliegende Arbeit verfolgt die Zielsetzung, sich verschiedenen Lösungsmöglichkeiten für das Rush-Hour-Puzzle in einem systematischen Vergleich zu nähern. Sie wirft die Frage auf, mit welchem Ansatz das Schiebepuzzle nicht nur in der minimalen Anzahl an Zügen, sondern zudem schnellstmöglich gelöst werden kann. Hierzu nähert sie sich diesem Problem aus zwei verschiedenen Richtungen. Zum einen werden zwei klassische Suchalgorithmen eingesetzt: Breitensuche als uniformierte Suche und A* mitsamt verschiedenen Heuristiken als heuristischer Ansatz. Zum anderen wird eine Möglichkeit vorgestellt, Rush Hour durch verschiedene Constraints zu erfassen, sodass die Instanzen von Microsofts Theoremlöser Z3 gelöst werden können. Ziel der Arbeit ist es, mit beiden Ansätzen einen Löser zu entwickeln, der für jede Instanz von Rush Hour (sofern vorhanden) die minimale Lösung in möglichst kurzer Zeit findet. Es stellt sich die Frage, welcher Ansatz hierzu besser geeignet ist und

wie die Ansätze im direkten Vergleich performen.

Bevor die in dieser Arbeit genutzten Suchalgorithmen vorgestellt werden, wird in einem ersten Kapitel die Spielsimulation des Rush-Hour-Puzzles beleuchtet.

In Kapitel 3 wird auf die Grundlagen der beiden Suchalgorithmen Breitensuche und A*-Suche eingegangen, wobei vorrangig deren Funktionsweise beleuchtet werden soll. Ein besonderer Fokus liegt dabei auf dem heuristischen Suchalgorithmus A*, der im Vergleich zur Breitensuche komplexer ist. Jeweils im Anschluss an deren Vorstellung wird die Implementierung der beiden Algorithmen in den Blick genommen. Insbesondere werden auch die verschiedenen Heuristiken, die im Zuge der Implementierung der A*-Suche genutzt wurden, thematisiert.

Darauf folgend wird ein Schlaglicht auf den zweiten im Rahmen dieser Arbeit diskutierten Lösungsansatz geworfen, indem zunächst die Idee hinter Constraint Programming sowie der genutzte Theoremlöser Z3 vorgestellt werden. Konkretisiert wird die Thematik durch die Betrachtung der für Rush Hour aufgestellten Constraints, auf deren Basis Z3 die Puzzleinstanzen lösen soll.

Bevor in Kapitel 5 dieser Arbeit die Ergebnisse der beiden bereits erwähnten Lösungsansätze für das Rush-Hour-Puzzle präsentiert und miteinander verglichen werden, werden außerdem die für ebendiesen Vergleich relevanten Benchmarks bzw. Parameter eingeführt. Auch die Wahl der Programmiersprache Python wird kurz diskutiert.

Im letzten Teil der Ausführungen werden die zuvor präsentierten Ergebnisse zur Diskussion gestellt. Ein besonderes Augenmerk wird dabei darauf geworfen, unter welchem Umständen welcher Solver für das Rush-Hour-Puzzle am performantesten ist und wie sich diese Beobachtungen in den Kontext dieser Arbeit beziehungsweise die Funktionsweise der betrachteten Lösungsansätze einordnen lassen. Diese Interpretation wird mit der bestehenden Literatur zum Rush-Hour-Puzzle verglichen, sodass der Beitrag dieser Arbeit resümiert und diese in einen Kontext eingeordnet werden kann. Gleichzeitig werden auch die Limitationen der Arbeit aufgezeigt und es wird herausgestellt, wie der Algorithmenvergleich zu noch aussagekräftigeren Ergebnissen hätte führen können und welche Optimierungsideen im Rahmen dieser Arbeit nicht betrachtet wurden.

Abschließend werden die Ergebnisse des Algorithmenvergleiches zusammengefasst. Ein Ende finden die Ausführungen, indem auf die Möglichkeiten und Fragen eingegangen wird, die diese Arbeit für die künftige Forschung eröffnet.

2 Spielsimulation

Damit das Rush-Hour-Puzzle mithilfe der Suchalgorithmen BFS und A* gelöst werden kann, muss das Spiel im Code auf eine geeignete Weise repräsentiert werden. Hierzu müssen nicht nur die Spielfiguren (Autos und LKW) und das Spielbrett auf sinnvolle Art und Weise dargestellt; auch die Spiellogik muss abgebildet werden.

2.1 Spielfiguren

Die Spielfiguren des Rush-Hour-Puzzles, also die zu bewegenden Autos und LKW, werden in der Klasse `Car` in der Datei `game.py` modelliert. Sie definiert die Eigenschaften sowie das Verhalten der Fahrzeuge. Ein `Car`-Objekt besitzt folgende Attribute:

- **`color`**: Die Farbe des Autos (die Farben werden mit entsprechenden numerischen Werten im `Enum`-Objekt `Color` repräsentiert)
- **`x`**, **`y`**: Die Startposition des Autos auf dem Spielfeld. Die Koordinaten `(x, y)` definieren die linke bzw. obere Ecke des Fahrzeugs.
- **`length`**: Die Länge des Autos. Sie kann entweder 2 (Auto) oder 3 (LKW) sein.
- **`orientation`**: Die Orientierung des Autos (die Orientierung wird mit entsprechenden numerischen Werten im `Enum`-Objekt `Orientation` repräsentiert und ist entweder `HORIZONTAL` oder `VERTICAL`).

Wichtige Methoden:

- **`positions()`**: Diese Methode berechnet alle Gitterpositionen, die das Auto belegt, basierend auf seiner Startposition, Länge und Orientierung. Sie gibt die von einem Fahrzeug belegten Koordinaten in einer Liste von Tupeln zurück.
- **`update_cordinates(new_position)`**: Diese Methode aktualisiert die Position des Autos, indem sie die Attribute `x` und `y` auf eine neue Position setzt, die ihr als Parameter übergeben wird.

2.2 Spielbrett

Die Klasse `Board` repräsentiert das Spielfeld, auf dem die Fahrzeuge bewegt werden. Ein Spielbrett definiert sich durch die folgenden Attribute:

- **`cars`**: Eine Liste von `Car`-Objekten, die alle Fahrzeuge auf dem Spielbrett enthält.
- **`size`**: Die Größe des quadratischen Gitters (Standardwert: 6×6).

Wichtige Methoden:

- **`initialize_board()`**: Diese Methode initialisiert das Spielbrett, indem sie die Positionen aller Fahrzeuge in einem 2D-Array speichert. Jede Zelle im Array enthält entweder den Wert 0 (leer) oder die ID der Farbe des Fahrzeuges, das die jeweilige Zelle belegt. Falls Fahrzeuge überlappen oder außerhalb des Gitters platziert werden, wird eine Exception ausgelöst, da es sich hierbei nicht um eine gültige Rush-Hour-Konfiguration handeln kann.
- **`is_won()`**: Diese Methode überprüft, ob das Spiel gewonnen wurde. Dies ist der Fall, wenn das rote Auto `RED` das rechte Ende des Spielfeldes erreicht.
- **`check_move(grid, move)`**: Diese Methode überprüft eine potenzielle Bewegung eines Fahrzeugs. Hierfür nimmt sie das Spielfeld in Form eines 2D-Arrays sowie einen zu testenden Spielzug entgegen. Ein Spielzug wird hierbei als Tupel der Form `[Car, Zuglänge, Richtung]` übergeben. Die Funktion validiert, ob das Fahrzeug in die angegebene Richtung bewegt werden kann, ohne dabei von anderen Fahrzeugen blockiert zu werden oder die Grenzen des Spielbretts zu überschreiten.
- **`do_move(move)`**: Diese Methode führt einen gültigen Spielzug aus, indem sie die Position des bewegten Autos auf dem Spielfeld entsprechend der angegebenen Schrittweite und Richtung aktualisiert.

2.3 Spielzustände

Sowohl die Breitensuche als auch die A*-Suche nehmen Zustände als Parameter entgegen. Hierfür wird die `State`-Klasse implementiert. Sie beschreibt den Zustand des Spiels zu einem bestimmten Zeitpunkt. Die Klasse enthält sowohl die aktuelle Konfiguration der Fahrzeuge als auch zusätzliche Parameter, die es ermöglichen, die Suche effizient zu gestalten. Ein `State`-Objekt ist durch folgende Attribute gekennzeichnet:

- **`cars`**: Eine Liste von `Car`-Objekten, die den aktuellen Spielzustand definieren.
- **`parent`**: Das übergeordnete `State`-Objekt, aus dem dieser Zustand hervorgegangen ist.
- **`move`**: Der Zug, der den Übergang zum Spielzustand verursacht hat. Er wird als Tupel der Form `[Car, Zuglänge, Richtung]` angegeben.
- **`g`**: Der Wert für die Kosten, um den Zustand zu erreichen.
- **`heuristic_type`**: Der Typ der Heuristik, die zur Berechnung des geschätzten Kostenwertes (`h`) verwendet wird.
- **`board_size`**: Die Größe des Spielfeldes. Der Standardwert ist 6.
- **`h`**: Die geschätzten Kosten, um das Ziel zu erreichen. Dieser Wert wird durch die Wahl der Heuristik bestimmt.
- **`f`**: Die Gesamtkosten des Zustandes. Wird berechnet, indem die bisherigen Kosten (`g`) und die geschätzten zukünftigen Kosten (`h`) summiert werden.

Wichtige Methoden:
Innerhalb der `State`-Klasse werden verschiedene Methoden sowie die für A* genutzten Heuristiken definiert. Auf die Heuristiken wird in Abschnitt 3.2.1 detailliert eingegangen, weshalb sie an dieser Stelle nicht betrachtet werden.

- **`heuristic()`**: Diese Methode berechnet den heuristischen Wert `h` des Zustands. Die Heuristik wird anhand des Attributs `heuristic_type` ausgewählt.
- **`is_blocking(blocking_car, red_car)`**: Diese Methode überprüft, ob das Auto, das im Parameter `blocking_car` angegeben wird, das rote Auto blockiert. Sie gibt `True` zurück, wenn dies der Fall ist, und `False` ansonsten.
- **`__lt__(other)`**: Diese Methode überschreibt den Kleiner-Als-Operator `<`, um zwei `State`-Objekte miteinander zu vergleichen. Der Vergleich erfolgt standardmäßig anhand des `f`-Werts. Ein Spielzustand mit einem niedrigeren `f`-Wert wird bevorzugt, da er einen kostengünstigeren Weg zum Zielzustand darstellt.
- **`is_goal()`**: Diese Methode überprüft, ob der aktuelle Spielzustand der Zielzustand ist. Dies ist das Fall, wenn das rote Auto am rechten Rand des Spielfeldes angekommen ist.
- **`generate_successors()`**: Diese Methode erzeugt alle Nachfolgezustände des aktuellen Spielzustands und gibt sie in einer Liste zurück. Für jedes Fahrzeug auf dem Spielfeld wird hierzu jede mögliche Bewegung (vorwärts oder rückwärts) getestet und ein neuer Zustand mit den angepassten Positionen der Fahrzeuge wird erzeugt. Für jeden gültigen Zug wird dabei ein neues `State`-Objekt erstellt, das die aktualisierten Positionen der Fahrzeuge, den aktuellen Zustand als Elternteil, den durchgeführten Zug sowie die neuen Kosten enthält.

3 Suchalgorithmen

3.1 Breitensuche

Die Breitensuche (auch Breadth-first search bzw. BFS) ist ein uniformierter Suchalgorithmus, der in Zusammenhang mit Graphen Verwendung findet. Ausgehend von einem Startknoten erkundet die BFS systematisch alle adjazenten Knoten, bevor sie eine Ebene tiefer in den Graphen vordringt. Auf diese Weise kann garantiert werden, dass alle Knoten in einer bestimmten Entfernung vom Startknoten vollständig erkundet werden, bevor weiter entfernte Knoten betrachtet werden[11, S. 554]. Die Breitensuche nutzt dabei eine Queue als zentrale Datenstruktur. Der Algorithmus fügt besuchte Knoten der Warteschlange hinzu und verarbeitet diese in der Reihenfolge ihres Einfügens nach dem FIFO-Prinzip[12, S. 577]. Nach der Bearbeitung eines Knotens wird dieser als besucht markiert, um Zyklen zu vermeiden und die einmalige Bearbeitung jedes Knotens sicherzustellen[13, S. 213f.].

Im Kontext von Schiebepuzzlen wie Rush Hour kann die Breitensuche zur Generierung aller möglichen Spielzüge genutzt werden. Auf diese Weise kann sichergestellt werden, dass auf jeden Fall die minimale Lösung einer gegebenen Rush-Hour-Instanz gefunden wird. Der größte Nachteil der Breitensuche ist die Speicherintensität: Da alle Nachfolgerknoten eines States im Speicher gehalten werden müssen, kann es bei dichten oder großen Graphen zu hohen Speicheranforderungen kommen. Bei allen in dieser Arbeit betrachteten Rush-Hour-Startzuständen auf einem 6×6-Spielfeld kann dieser Nachteil insofern vernachlässigt werden, als dass garantiert werden kann, dass eine feste Obergrenze für die Anzahl an zu besuchenden und somit zu speichernden Knoten existiert.

3.1.1 Implementierung

Die Breitensuche für das Rush-Hour-Puzzle ist in `bfs.py` implementiert, um mit einer schrittweisen Exploration aller möglichen Spielzustände eine Lösung für jeden möglichen Startzustand des Schiebepuzzles zu finden.

In der Funktion `bfs`, die als Argument eine Rush-Hour-Instanz in Form eines `State`-Objektes entgegennimmt, wird zu Beginn eine Warteschlange `open_queue` initialisiert, die nur den initialen Spielzustand enthält. Als Datenstruktur wurde eine Double-Ended-Queue verwendet. Diese hat den zentralen Vorteil, dass zu untersuchende Nachfolgezustände am Ende der Warteschlange eingefügt, während die Zustände, die verarbeitet werden sollen, effizient vom Anfang der Queue entfernt werden können. Zusätzlich wird ein Dictionary `closed_dict` genutzt, um bereits untersuchte Zustände zu speichern. Auf diese Weise wird vermieden, dass ein und derselbe Zustand mehrfach untersucht wird, was die Effizienz des Algorithmus steigert. Um einen Spielzustand dabei eindeutig zu identifizieren, werden die Attribute aller `car`-Objekte (Farbe, Position, Länge und Orientierung) eines States extrahiert und in einem sortierten Tupel abgespeichert. Das Tupel wird als Schlüssel im Wörterbuch verwendet, wobei jedem Schlüssel das bisherige Ergebnis der Kostenfunktion `g` als Wert zugewiesen wird.

Der Algorithmus iteriert nun über alle Zustände in der Queue. Für jeden Zustand wird zunächst überprüft, ob dieser ein Zielzustand ist, indem die Methode `is_goal()` aufgerufen wird. Wenn ein solcher gefunden wird, wird der Pfad zu diesem Zustand mittels der Methode `reconstruct_path()` rekonstruiert und als Lösung zurückgegeben.

Falls der aktuelle Zustand kein Zielzustand ist, werden alle möglichen Nachfolgezustände des aktuell betrachteten States mit der Funktion `generate_successors()` generiert. Jeder Nachfolgezustand wird basierend auf den Attributen seiner `car`-Objekte in ein eindeutiges Tupel konvertiert. Daraufhin kann geprüft werden, ob der Zustand bereits im Wörterbuch `closed_dict` enthalten ist. Falls ja, wird dieser Zustand ignoriert, da er bereits untersucht wurde. Andernfalls wird der Nachfolgezustand der Warteschlange hinzugefügt und als besucht markiert, indem er in das Dictionary aufgenommen wird.

Dieser Prozess wird solange wiederholt, bis entweder ein Zielzustand erreicht wird oder die Warteschlange leer ist. Im letzteren Fall gibt der Algorithmus `None` zurück, was bedeutet, dass keine Lösung existiert. Durch die iterative Betrachtung aller Zustände in Kombination mit der Vermeidung von Doppelbesuchen garantiert die Implementierung, dass eine Lösung - sofern sie existiert - mit minimalen Kosten gefunden wird.

3.2 A*-Suche

Bei der A*-Suche handelt es sich um einen heuristischen Suchalgorithmus, der als Weiterentwicklung des Dijkstra-Algorithmus angesehen werden kann und somit auch Elemente der in Abschnitt 3.1 vorgestellten Breitensuche enthält. Die A*-Suche „requires more information about the graph, in order to make a more informed decision when selecting the next vertex to visit“[14, S. 27]. Diese informierte Entscheidung wird bei der A*-Suche getroffen, indem bei der Traversierung des Graphen sowohl die bisherigen Kosten als auch eine Schätzung der noch verbleibenden Kosten zum Zielknoten berücksichtigt werden. Auf diese Weise wird die Suche in vielversprechende Richtungen gelenkt. Konkret verwendet der A*-Algorithmus hierzu eine Kostenfunktion $f(n)$, wobei n den aktuell betrachteten Knoten darstellt, die wie folgt definiert ist:

$$f(n) = g(n) + h(n),$$

wobei $g(n)$ die Kosten sind, um den Knoten n vom Startknoten aus zu erreichen, und $h(n)$ eine Heuristik (siehe Abschnitt 3.2.1) darstellt, die die verbleibenden Kosten von Knoten n bis zum Zielknoten abschätzt.

Bei der A*-Suche wird der f-Wert für alle Knoten im Suchraum miteinander verglichen. Hierzu wird die Datenstruktur einer Prioritätswarteschlage genutzt, die es ermöglicht, die Knoten basierend auf ihrem f-Wert in eine Queue, die sogenannte Open List, einzusortieren. Diese Open List enthält also alle Knoten, die noch untersucht werden müssen. Anschließend wird der Knoten mit dem kleinsten f-Wert aus der Warteschlange ausgewählt, weiter untersucht und in die Closed List verschoben. Hierbei handelt es sich um eine Datenstruktur, in der Knoten, die bereits untersucht wurden, abgespeichert

werden, um Redundanzen in der Suche zu vermeiden.

Bei der Auswahl von Knoten aus der Prioritätswarteschlange kann es vorkommen, dass mehrere Knoten denselben f-Wert haben. Eine einfache Möglichkeit, mit dieser Situation umzugehen, ist es, die Knoten in der Reihenfolge ihres Einfügens in die Queue zu bearbeiten. Eine allgemein effizientere Möglichkeit ist das sogenannte **Tie-Breaking**. Hierbei werden zusätzliche Kriterien herangezogen, um die Reihenfolge der Knotenverarbeitung festzulegen. Eine gängige Strategie ist es, Knoten mit einem höheren h-Wert zu bevorzugen, da dies dazu beiträgt, die Suche in Richtung des Ziels zu lenken. Alternativ kann auch der g-Wert für die Entscheidung herangezogen werden, um kürzere bisherige Pfade zu priorisieren[15].

3.2.1 Heuristiken

Maßgeblich für die Effizienz des A*-Algorithmus ist die Wahl einer geeigneten Heuristik. Eine Heuristik ist definiert als eine Funktion, die eine Schätzung der verbleibenden Kosten ausgehend vom aktuell betrachteten Knoten bis zum Zielzustand liefert. In diesem Kontext sind zwei zentrale Eigenschaften der Heuristik von Bedeutung:

- **Zulässigkeit**: Eine Heuristik h wird als zulässig bezeichnet, wenn sie niemals die tatsächlichen Kosten von einem Knoten zum Ziel überschätzt. Formal ausgedrückt bedeutet dies, dass für jeden Knoten n gilt:

  $$h(n) \leq h^*(n),$$

  wobei $h^*(n)$ die tatsächlichen minimalen Kosten von Knoten n zum Zielzustand sind.

  „With an admissible heuristic, A* is cost-optimal“[16, S. 106], da der Algorithmus so niemals fälschlicherweise eine teurere Lösung als die minimale bevorzugt.

- **Konsistenz**: Eine Heuristik h ist konsistent oder monoton, wenn für jeden Knoten n und dessen Nachbarknoten n' gilt, dass die Heuristik von n niemals die tatsächlichen Kosten des Übergangs zu einem Nachbarn überschätzt:

  $$h(n) \leq c(n, n') + h(n'),$$

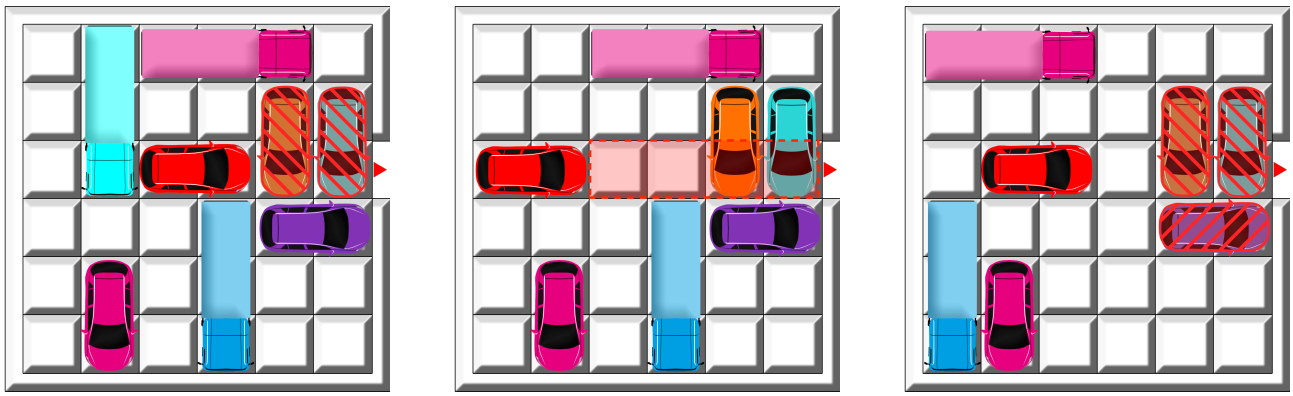
  wobei $c(n, n')$ die tatsächlichen Kosten des Übergangs von Knoten n zum Nachbarn n' sind.

  Die Konsistenz einer Heuristik impliziert Zulässigkeit, da die Heuristik auf dem gesamten Suchraum nie die minimalen Kosten überschätzt und durch die oben genannte Formel die Schätzungen immer korrekt bleiben. Die Konsistenz stellt sicher, dass A* direkt den optimalen Pfad zu einem Zustand findet, wodurch die Effizienz der Suche erhöht wird[16, S. 106].

Heuristiken bei Rush Hour

Auch bei der Anwendung der A*-Suche auf das Rush-Hour-Puzzle spielt die Wahl der Heuristik selbstverständlich eine Rolle für die Effizienz der Suche. Hauptman et al. haben gezeigt, dass je nach Wahl der Heuristik und initialem Spielstand die Anzahl der besuchten Spielstände bzw. Knoten bei der Lösung einer Rush-Hour-Instanz stark voneinander abweicht: „The effect of these heuristics on search efficiency was inconsistent, alternating between decreasing the number of nodes traversed by 70% (for certain initial configurations) and increasing this number by as much as 170% (for other configurations)“[9, S. 956]. Es überrascht daher nicht, dass sich in der Literatur eine Reihe verschiedener Heuristiken unterschiedlicher Komplexität für Rush Hour finden lassen[14, 9]. Im Rahmen dieser Arbeit, die sich schwerpunktmäßig einem Vergleich verschiedener Lösungsansätze und nicht der Optimierung von A* für Rush Hour widmet, werden folgende Ansätze für Heuristiken betrachtet:

- **`constant_heuristic`**: Diese Heuristik ist die naivste Heuristik für Rush Hour. Sie unterscheidet nur zwischen blockierten und nicht blockierten Zuständen und gibt 0 zurück, wenn das rote Auto nicht blockiert ist, anderenfalls gibt sie 1 zurück.
- **`count_cars_between`**: Diese Heuristik ist ebenfalls trivial. Sie zählt die Fahrzeuge, die sich zwischen dem roten Auto und der Ausfahrt befinden[14, S. 40]. Sie dient als grobe Schätzung der Blockaden auf dem direkten Weg zum Ziel.
- **`goal_distance`**: Diese Heuristik folgt einem ebenfalls naiven Approach, indem sie die Anzahl der Felder zwischen dem roten Auto und der Ausfahrt zählt. Sie liefert somit eine Schätzung der minimalen Anzahl an Zügen, die notwendig sind, um das Ziel zu erreichen.
- **`count_blocking_cars`**: Diese Heuristik zählt alle Fahrzeuge, die das rote Auto direkt oder indirekt blockieren. Fahrzeuge, die andere blockierende Autos versperren, werden ebenfalls berücksichtigt, um eine umfassendere Schätzung der Blockaden zu erhalten.
- **`movable_blocking_cars`**: Diese Heuristik prüft, welche Fahrzeuge das rote Auto blockieren und ob sie initial aus dem Weg bewegt werden können. Fahrzeuge, die sich nicht bewegen lassen, erhalten eine Strafbewertung, um ihre blockierende Wirkung zu betonen.



(a) `count_cars_between = 2`
`constant_heuristic = 1`

(b) `goal_distance = 4`

(c) `count_blocking_cars = 3`
`movable_blocking_cars = 12`

Abbildung 3.1: Heuristiken für Rush Hour am Beispiel.

3.2.2 Implementierung

Algorithm 1 A*-Algorithmus für Rush Hour

```
1:  procedure A*(initial_state)
2:      open_list ← ∅
3:      closed_dict ← ∅
4:      visited_states_count ← 0
5:      push initial_state into open_list
6:      closed_dict[initial_state] ← g(initial_state)
7:      while open_list ≠ ∅ do
8:          current_state ← pop open_list
9:          visited_states_count ← visited_states_count + 1
10:         if current_state ist Zielzustand then
11:             return RECONSTRUCTPATH(current_state), visited_states_count
12:         end if
13:         for all successor ∈ GENERATESUCCESSORS(current_state) do
14:             if successor ∈ closed_dict and closed_dict[successor] ≤ g(successor) then
15:                 continue
16:             end if
17:             push successor into open_list
18:             closed_dict[successor] ← g(successor)
19:         end for
20:     end while
21:     return None, visited_states_count
22: end procedure
```

Die A*-Suche für das Rush-Hour-Puzzle ist in der Funktion `a_star` in der Datei `a_star.py` implementiert und soll den Lösungsweg mit den minimalen Zügen von einem Rush-Hour-Startzustand zu einem Zielzustand finden.

Der Algorithmus erhält als Eingabe eine Rush-Hour-Instanz in Form eines `State`-Objekts, das die initiale Spielkonfiguration repräsentiert. Zu Beginn wird eine Prioritätswarteschlange `open_list` initialisiert, die mithilfe der `heapq`-Bibliothek als Min-Heap realisiert ist. Diese Struktur erlaubt es, den Knoten mit dem kleinsten f-Wert effizient auszuwählen. Die Wahl des Min-Heaps für die `open_list` im A*-Algorithmus basiert auf dessen Effizienz bei der Handhabung der Prioritätswarteschlange. Da der Min-Heap eine binäre Baumstruktur ist, bei der jedes Elternknotenpaar kleiner oder gleich seinen Kindknoten ist, befindet sich das Element mit dem kleinsten Wert stets an der Wurzel und kann in $O(1)$ extrahiert werden. So kann in jedem Iterationsschritt der Zustand mit dem geringsten f-Wert effizient verarbeitet werden.

Zunächst wird der Startzustand in die `open_list` eingefügt, wobei sein f-Wert als Priorität verwendet wird. Das Hinzufügen eines neuen Elements zum Heap erfolgt in logarithmischer Laufzeit. Da die A*-Suche bei jedem Schritt alle Nachfolgezustände eines aktuellen Zustands in die `open_list` einfügen muss, garantiert der Min-Heap eine effiziente Aktualisierung der Queue.

Zusätzlich wird das Dictionary `closed_dict` verwendet, in dem bereits besuchte Zustände sowie deren Kosten gespeichert werden. Hierbei werden die Attribute aller im Zustand enthaltenen Fahrzeuge in einem sortierten Tupel zusammengefasst und als Schlüssel im Dictionary gespeichert. Dieses Tupel repräsentiert den Spielzustand eindeutig. Der Wert des Eintrags ist der bisherige Wert der Kostenfunktion $g(n)$, der die kürzeste bisher bekannte Distanz vom Startzustand zum gespeicherten Zustand repräsentiert.

Das `closed_dict` dient an dieser Stelle als eine sogenannte **Transposition Table**. Diese Technik ist entscheidend, um Redundanzen in der Suche zu vermeiden und den Algorithmus zu beschleunigen, insbesondere bei stark verzweigten Suchräumen wie dem Rush-Hour-Puzzle. Die Funktion des `closed_dict` ähnelt der einer Hashmap, da die Suche, das Einfügen und das Abrufen von Einträgen in $O(1)$ erfolgen. Dies ermöglicht es, schnell zu überprüfen, ob ein Zustand bereits untersucht wurde und ob dessen aktueller Kostenwert g günstiger ist als ein zuvor schon gespeicherter Wert. Falls ein Zustand mit niedrigeren Kosten erreicht wird, wird der Eintrag im `closed_dict` aktualisiert, um den Algorithmus stets auf dem kürzesten bekannten Pfad fortzusetzen.

Die Verwendung des `closed_dict` als Transposition Table hat also mehrere Vorteile:

- **Reduktion von Redundanzen:** Zustände, die bereits untersucht wurden, werden übersprungen, was sowohl Zeit als auch Speicherplatz spart.
- **Optimierung durch Kostenprüfung:** Durch die Speicherung der g-Werte wird sichergestellt, dass nur der günstigste Weg zu einem Zustand weiterverfolgt wird.
- **Effizienz durch Hashing:** Die Hashing-Operation des Dictionaries gewährleistet eine konstante Zugriffszeit.
- **Flexibilität bei der Zustandsprüfung:** Die eindeutige Repräsentation als sortiertes Tupel verhindert Fehler bei der Identifikation von Zuständen, unabhängig von der Reihenfolge der Fahrzeuge.

Der Algorithmus arbeitet iterativ, indem er den Zustand mit dem kleinsten f-Wert aus der `open_list` entfernt und verarbeitet. Wenn Zustände denselben f-Wert besitzen, wird aufgrund der Tie-Breaking-Strategie (abhängig vom gewählten `__lt__()`-Operator) entschieden:

1. **Kein Tie-Breaking**: Der Zustand mit dem kleinsten f-Wert wird nach dem FIFO-Prinzip gewählt.
2. **Zweistufiges Tie-Breaking**: Der Zustand mit dem kleinsten f-Wert wird gewählt. Wenn mehrere Zustände denselben f-Wert haben, wählt der Algorithmus den nächsten State basierend auf dem größten bisherigen g-Wert aus. Auf diese Weise werden weniger Alternativen verfolgt.
3. **Dreistufiges Tie-Breaking**: Der Zustand mit dem kleinsten f-Wert wird gewählt. Bei mehreren identischen f-Werten wird der kleinste g-Wert bevorzugt. Wenn auch identische g-Werte vorliegen, folgt tertiäres Tie-Breaking basierend auf dem kleinsten h-Wert.

Nachdem der nächste Zustand aus der Queue ausgesucht wurde, wird zunächst geprüft, ob er ein Zielzustand ist, indem die Methode `is_goal()` aufgerufen wird. Falls dies der Fall ist, wird der Pfad zum Zielzustand mithilfe der Hilfsfunktion `reconstruct_path()` berechnet und als Lösung zurückgegeben.

Falls der aktuelle State kein Zielzustand ist, werden mithilfe der Methode `generate_successors()` alle möglichen Nachfolgezustände generiert. Für jeden Nachfolgezustand wird erneut ein eindeutiges Tupel basierend auf den Fahrzeugattributen erstellt. Daraufhin wird überprüft, ob der Spielzustand bereits im `closed_dict` enthalten ist und ob der für ihn gespeicherte g-Wert kleiner oder gleich dem g-Wert des Nachfolgers ist. Ist dies der Fall, wird der Nachfolgezustand ignoriert, da eine kürzere oder gleichwertige Lösung bereits gefunden wurde. Andernfalls wird der Zustand in die `open_list` eingefügt und sein g-Wert wird im `closed_dict` aktualisiert.

Dieser Prozess wird solange wiederholt, bis entweder ein Zielzustand erreicht wird oder die `open_list` leer ist. Im letztgenannten Fall gibt der Algorithmus `None` zurück, was gleichbedeutend damit ist, dass keine Lösung für die gegebene Puzzle-Instanz existiert.

4 Constraint Programming

Constraint Programming ist ein Ansatz zur Problemlösung, der auf der Definition und effizienten Lösung von Constraints basiert. Als Constraints werden hierbei Bedingungen oder Regeln verstanden, die die zulässigen Werte oder Kombinationen von Werten in einem gegebenen Problem beschreiben. Constraint Programming wird in verschiedenen Bereichen wie Terminplanung, der Optimierung von Ressourcen oder logischen Rätseln wie unter anderem Rush Hour eingesetzt, da diese Probleme sich durch eine Vielzahl von Bedingungen hinreichend charakterisieren lassen.

Der Programmierprozess im Bereich Constraint Programming zeichnet sich durch einen „Two Phases Approach“[17, S. 3] aus. Um ein Problem mithilfe von Constraint Programming lösen zu können, muss dieses Problem in einer ersten Phase als Constraint Satisfaction Problem dargestellt werden. Ein solches besteht typischerweise aus drei Hauptkomponenten: einer Menge von Variablen, den Domänen bzw. Wertebereichen dieser Variablen und einer Menge von Constraints, die die gültigen Wertzuweisungen einschränken[18]. In einer zweiten Phase folgt die Lösung der Constraint Programming-Probleme, indem Wertzuweisungen für die Variablen so gefunden werden, dass alle Constraints dabei erfüllt sind.

Das Programmierparadigma Constraint Programming ist im Allgemeinen durch eine deklarative Problembeschreibung gekennzeichnet; der Fokus liegt also darauf, *was* erfüllt sein muss, anstatt *wie* das spezifische Problem zu lösen ist. Dies erlaubt es, Probleme kompakt und verständlich zu beschreiben. Theoremlöser wie Z3 können genutzt werden, um für einen derart modellierten Sachverhalt eine passende Variablenbelegung zu finden.

4.1 Z3 Theoremlöser

Z3 ist ein moderner Theoremlöser, der von Microsoft Research entwickelt wurde und sich durch die Möglichkeit auszeichnet, verschiedene Arten logischer Probleme effizient zu lösen. Er basiert auf der Satisfiability Modulo Theory, die eine Erweiterung des klassischen Satisfiability-Problems darstellt und eine „technology for the fully automated solution of logical formulas“[19, S. xxi] bietet.

„The main problem in SMT is determining whether a first-order formula is satisfiable in a model that also satisfies one or more fixed background theories“[19, S. 1]. Während SAT-Probleme darauf abzielen, die Erfüllbarkeit boolescher Formeln zu überprüfen, ermöglicht Z3 durch die Integration verschiedener Werkzeuge wie arithmetischen Gleichungen, Mengen oder Arrays die Lösung weitaus komplexerer Problemstellungen[20]. Der Z3-Solver arbeitet, indem er die gegebenen logischen Constraints analysiert und prüft, ob diese erfüllbar sind oder nicht. Dabei wird ein iterativer Ansatz verwendet, der zwischen der SAT-Suche und der Überprüfung domänenspezifischer Einschränkungen wechselt.

Z3 wird in einer Vielzahl von Anwendungen eingesetzt, darunter Model-Checking und Optimierungsprobleme. Die Effizienz von Z3 resultiert aus der Kombination von spezialisierter Datenstrukturierung und Algorithmen, die die Lösung von Problemen auch bei großen Eingaben praktikabel machen [20].

4.2 Constraints für Rush Hour

Auch das Rush-Hour-Puzzle lässt sich durch eine Reihe verschiedener Constraints so beschreiben, dass es mithilfe von Solvern wie Z3 gelöst werden kann. Cian et al.[10] haben eine Modellierung von Rush Hour vorgestellt, der in dieser Arbeit weitgehend gefolgt wird.

Bevor die Constraints im Einzelnen betrachtet werden können, ist es nötig, sich vor Augen zu führen, welche statischen Eingaben für eine Modellierung des Schiebepuzzles benötigt werden. Es ist sinnvoll, die Eigenschaften jedes Fahrzeuges, die einen initialen Zustand in Gänze beschreiben, als Eingabeparameter zu nutzen. Deshalb werden im Folgenden drei Arrays eingeführt, die die Eigenschaften der verschiedenen Fahrzeuge abbilden:

- **size**: In diesem Array ist die Größe jedes Autos bzw. LKW enthalten; es enthält demnach als Werte entweder 2 oder 3.
- **fixed_row_col**: Hier wird für jedes Fahrzeug die Zeile (bei horizontalen Fahrzeugen) oder Spalte (bei vertikalen Fahrzeugen) gespeichert, die unveränderlich ist. Die Werte im Array liegen im Intervall $[-6, 6]$, wobei negative Zahlen indizieren, dass es sich um ein vertikales Fahrzeug handelt.
- **initial**: Dieses Array enthält für jedes Fahrzeug die kleinste Zelle im Spielbrettgitter, die es belegt.

Die Modellierung von Rush Hour benötigt außerdem zwei zweidimensionale Arrays, die die Positionen der Fahrzeuge und die Züge abbilden:

- **pos**: Mit $\text{pos}[i, j]$ wird die kleinste Zelle beschrieben, die von Fahrzeug i zum Zeitpunkt j belegt wird.
- **move**: Dieses Array bildet die Züge ab. Der Wert an Stelle $\text{move}[i, j]$ ist 0, wenn Fahrzeug i zum Zeitpunkt j nicht bewegt wird, anderenfalls gilt $\text{move}[i, j] = \delta$, wenn Fahrzeug i zum Zeitpunkt j um δ Schritte bewegt wird.

Nachdem die Variablen modelliert wurden, können die Domänen der Variablen sowie die einzelnen Spielregeln von Rush Hour mithilfe von diversen Constraints abgebildet werden.

Constraints für ein gültiges Spielfeld

In einer ersten Constraint wird der Startzustand abgebildet, indem dafür gesorgt wird, dass für alle Fahrzeuge im Array *pos* zum Zeitpunkt $j = 1$ ihre initiale Position gespeichert wird, die durch das statische Array *initial* übergeben wird:

$$\forall v \in [1, \text{vehicles}]\ (\text{pos}[v, 1] = \text{initial}[v]) \tag{4.1}$$

Um sicherzustellen, dass kein Fahrzeug das Spielfeld verlässt, wird eine weitere Constraint formuliert, die für alle Fahrzeuge v und jeden Zeitpunkt s überprüft, ob die Position des letzten vom Fahrzeug belegten Feldes innerhalb der Spielfeldgrenzen liegt. Für jedes Fahrzeug v wird geprüft, ob die Summe der Position des ersten Feldes pos[v,s] und der Fahrzeuglänge size[v] abzüglich eins kleiner oder gleich der Spielfeldgröße 6 ist. Diese Constraint gewährleistet, dass kein Fahrzeug das Rush-Hour-Spielfeld an der rechten oder unteren Grenze überschreitet:

$$\text{pos[v, s]} + \text{size[v]} - 1 \leq 6 \tag{4.2}$$

Die Spielbedingung, dass sich zwei Fahrzeuge nicht überlappen dürfen, wird ebenfalls durch eine Constraint definiert. Diese stellt für je zwei Fahrzeuge v_1 und v_2, die sich entweder in derselben Spalte oder derselben Zeile befinden, sicher, dass sich ihre Positionen zu keinem möglichen Zeitpunkt s überschneiden. Dafür muss die Bedingung gelten, dass entweder das letzte von v_1 belegte Feld vor dem ersten von v_2 liegt oder umgekehrt:

$$\begin{aligned} &\forall v_1, v_2 \in [1, \text{r}],\ s \in [1, \text{t}] \text{ mit } v_1 < v_2 \ \wedge\ \text{fixed\_row\_col}[v_1] = \text{fixed\_row\_col}[v_2] : \\ &\quad \text{pos}[v_1, \text{s}] + \text{size}[v_1] - 1 < \text{pos}[v_2, \text{s}] \ \vee\ \text{pos}[v_2, \text{s}] + \text{size}[v_2] - 1 < \text{pos}[v_1, \text{s}] \end{aligned} \tag{4.3}$$

Nun wird inhaltlich dieselbe Constraint für Paare orthogonaler Fahrzeuge formuliert. In diesem Fall wird explizit vermieden, dass sie ein Kreuz bilden, also sich gegenseitig in einer Art und Weise schneiden, die bei Rush Hour nicht erlaubt ist. Dabei wird für jedes Fahrzeugpaar v_1 und v_2, bei dem eines der Fahrzeuge vertikal und das andere horizontal ist, überprüft, ob sich ihre Positionen so überschneiden, dass sie ein Kreuz bilden:

$$\begin{aligned} &\forall v_1, v_2 \in [1, \text{r}],\ s \in [1, \text{t}] \text{ mit } \text{fixed\_row\_col}[v_1] > 0 \ \wedge\ \text{fixed\_row\_col}[v_2] < 0 : \\ &\neg\ (\text{pos}[v_1, \text{s}] \leq -\text{fixed\_row\_col}[v_2] \ \wedge\ -\text{fixed\_row\_col}[v_2] \leq \text{pos}[v_1, \text{s}] + \text{size}[v_1] - 1 \\ &\quad \wedge \text{pos}[v_2, \text{s}] \leq \text{fixed\_row\_col}[v_1] \ \wedge\ \text{fixed\_row\_col}[v_1] \leq \text{pos}[v_2, \text{s}] + \text{size}[v_2] - 1) \end{aligned} \tag{4.4}$$

Constraint für das Spielziel

Auch das Ziel von Rush Hour, das rote Auto an den rechten Rand des Spielfeldes zu bewegen, muss durch eine Constraint abgebildet werden. Hierfür benötigt man ebenfalls die *pos*-Variable, denn es muss einen beliebigen und noch nicht bekannten Zeitpunkt t geben, an dem das rote Auto (Annahme: das rote Auto wird in den statischen Arrays, die als Parameter genutzt werden, als erstes beschrieben) die fünfte Zelle des Rush-Hour-Spielbretts belegt:

$$\text{pos}[1, \text{t}] = 5 \tag{4.5}$$

Constraints für gültige Züge

Auch die Züge werden mithilfe von Constraints modelliert. Es muss zunächst sichergestellt werden, dass pro Zeitschritt genau eine Bewegung durchgeführt wird. Hierzu wird für Zeitpunkt s überprüft, dass die Summe der Bewegungen der Fahrzeuge ungleich Null ist und somit genau ein Zug pro Zeitschritt durchgeführt wird. Dazu wurde die Indikatorfunktion $\mathbb{I}$ eingeführt, die für jedes Fahrzeug v und jeden Zeitpunkt s den Wert 1 zurückgibt, wenn das Fahrzeug zu diesem Zeitpunkt eine Bewegung ausführt, und den Wert 0, wenn keine Bewegung ausgeführt wird. Diese Funktion wird genutzt, um sicherzustellen, dass genau eine Bewegung pro Zeitschritt stattfindet.

$$\forall s \in [1, \text{t} - 1] : \tag{4.6}$$
$$\sum_{v=1}^{\text{r}} \mathbb{I}(\text{move}[v, \text{s}] \neq 0) = 1$$

Auch der Zug selbst muss durch eine Constraint abgebildet werden. Die folgende Constraint beschreibt, wie sich die Position eines Fahrzeugs von einem Zeitpunkt s zu einem späteren Zeitpunkt $s+1$ ändern kann. Sie besagt, dass die Position des Fahrzeugs v zum Zeitpunkt $s+1$ gleich der Position desselben Fahrzeugs zum Zeitpunkt s plus der Bewegung $\text{move}[v, s]$ zum Zeitpunkt s ist. Außerdem berücksichtigt sie die Trägheit, d.h. ein Fahrzeug bleibt in der gleichen Position, wenn es nicht bewegt wird:

$$\forall v \in [1, \text{r}],\ s \in [1, \text{t} - 1](\text{pos}[v, \text{s} + 1] = \text{pos}[v, \text{s}] + \text{move}[v, \text{s}]) \tag{4.7}$$

Eine weitere Constraint, die sicherstellt, dass Fahrzeuge während eines Zuges nicht über andere Fahrzeuge springen können, komplettiert die Modellierung der gültigen Rush Hour-Züge. Zunächst wird eine Regel für Fahrzeuge in derselben Spalte oder Zeile definiert. Sie besagt, dass es nicht erlaubt ist, dass ein Fahrzeug v_1 über ein anderes Fahrzeug v_2 springt. Konkret heißt das, dass sich das Fahrzeug v_1 zum Zeitpunkt $s + 1$ nach v_2 bewegt, ohne dass das Fahrzeug v_2 nachgibt:

$$\forall s \in [1, \text{t} - 1],\ v_1, v_2 \in [1, \text{r}] \text{ mit } v_1 < v_2 \ \wedge\ \text{fixed\_row\_col}[v_1] = \text{fixed\_row\_col}[v_2] : \tag{4.8}$$
$$\neg\,(\text{pos}[v_1, \text{s}] \leq \text{pos}[v_2, \text{s}] \ \wedge\ \text{pos}[v_1, \text{s} + 1] > \text{pos}[v_2, \text{s} + 1]) \ \wedge$$

$$\neg\,(\mathrm{pos}[v_2, \mathrm{s}] \leq \mathrm{pos}[v_1, \mathrm{s}] \;\wedge\; \mathrm{pos}[v_2, \mathrm{s}+1] > \mathrm{pos}[v_1, \mathrm{s}+1])$$

In einem nächsten Schritt werden die Constraints formuliert, die die Vermeidung von Sprüngen auch bei orthogonalen Fahrzeugpaaren sicherstellen. Die erste dieser Constraints prüft, ob ein vertikales Fahrzeug v_1 und ein horizontales Fahrzeug v_2 an den Zeitpunkten s und $s+1$ so positioniert sind, dass sie sich in einer Kreuzung schneiden könnten. Falls v_2 in der gleichen Spalte wie v_1 liegt, wird modelliert, dass sich die Position von v_1 so verändert, dass es zu beiden Zeitpunkten weiterhin außerhalb der „Schnittzone" von v_2 bleibt. Dies verhindert illegale Sprünge des vertikalen Fahrzeugs über das horizontale Fahrzeug:

$$\begin{aligned}
&\forall s \in [1, \mathrm{t}-1],\ v_1, v_2 \in [1, \mathrm{r}] \text{ mit } \mathrm{fixed_row_col}[v_1] < 0 \;\wedge\; \mathrm{fixed_row_col}[v_2] > 0 : \\
&\quad (\mathrm{pos}[v_2, \mathrm{s}] \leq -\mathrm{versus}[v_1] \;\wedge\; -\mathrm{versus}[v_1] \leq \mathrm{pos}[v_2, \mathrm{s}] + \mathrm{size}[v_2] - 1) \implies \\
&\quad (\mathrm{pos}[v_1, \mathrm{s}] < \mathrm{fixed_row_col}[v_2] \implies \mathrm{pos}[v_1, \mathrm{s}+1] < \mathrm{fixed_row_col}[v_2] \wedge \\
&\quad\ \mathrm{pos}[v_1, \mathrm{s}] > \mathrm{fixed_row_col}[v_2] \implies \mathrm{pos}[v_1, \mathrm{s}+1] > \mathrm{fixed_row_col}[v_2])
\end{aligned} \tag{4.9}$$

Diese Constraint modelliert das Vermeiden von Sprüngen im umgekehrten Fall - einem horizontalen Fahrzeug v_1 und einem vertikalen Fahrzeug v_2:

$$\begin{aligned}
&\forall s \in [1, \mathrm{t}-1],\ v_1, v_2 \in [1, \mathrm{r}] \text{ mit } \mathrm{fixed_row_col}[v_1] > 0 \;\wedge\; \mathrm{fixed_row_col}[v_2] < 0 : \\
&(\mathrm{pos}[v_2, \mathrm{s}] \leq \mathrm{fixed_row_col}[v_1] \;\wedge\; \mathrm{fixed_row_col}[v_1] \leq \mathrm{pos}[v_2, \mathrm{s}] + \mathrm{size}[v_2] - 1) \implies \\
&\quad (\mathrm{pos}[v_1, \mathrm{s}] < -\mathrm{versus}[v_2] \implies \mathrm{pos}[v_1, \mathrm{s}+1] < -\mathrm{fixed_row_col}[v_2]) \wedge \\
&\quad\ \mathrm{pos}[v_1, \mathrm{s}] > -\mathrm{versus}[v_2] \implies \mathrm{pos}[v_1, \mathrm{s}+1] > -\mathrm{fixed_row_col}[v_2])
\end{aligned} \tag{4.10}$$

Abschließend wird eine Constraint eingeführt, die verhindert, dass dasselbe Fahrzeug v in aufeinanderfolgenden Zeitschritten s und $s+1$ bewegt wird. Hierzu wird modelliert, dass die Multiplikation zweier aufeinanderfolgender Züge desselben Fahrzeuges immer 0 ergeben muss (dies impliziert, dass das Fahrzeug zu mindestens einem der beiden Zeitpunkte nicht bewegt wurde). Diese Regel schränkt redundante Zustände ein, die entstehen könnten, wenn ein Fahrzeug mehrfach hintereinander bewegt wird, ohne dass dies die Problemlösung voranbringt, da im klassischen Rush Hour ein Zug beliebig viele Felder umfassen kann:

$$\forall v \in [1, r],\ s \in [1, \mathrm{steps}-2]\,(\mathrm{move}[v, \mathrm{s}] \cdot \mathrm{move}[v, \mathrm{s}+1] = 0) \tag{4.11}$$

4.2.1 Implementierung

Die Implementierung der Constraints zur Lösung des Rush-Hour-Puzzles erfolgt mit der SMT-Solver-Bibliothek Z3 in der Datei `constraint_programming.py`. Die zentrale Funktion `z3_solver` definiert das Problem als Optimierungsproblem, in dem die Anzahl der Züge minimiert werden soll.

Zusätzlich zu den bereits in Abschnitt 4.2 vorgestellten Datenstrukturen für die Fahrzeuggrößen (`car_sizes`), die Fixierung auf eine Zeile oder Spalte (`car_fixed_rowcols`) sowie der kleinsten initial belegten Zelle (`car_initial`), wird in der Implementierung eine weitere Datenstruktur, `car_orientation`, eingeführt. Diese speichert die Ausrichtung jedes Fahrzeugs, wobei der ursprünglichen Idee der Modellierung folgende 1 für horizontale Fahrzeug und -1 für vertikale Fahrzeuge genutzt wird.

Diese Erweiterung ist notwendig, um die Orientierung von Fahrzeugen zu unterscheiden, wenn diese in der nullten Reihe oder Spalte positioniert sind (und somit bei ausschließlicher Nutzung von `car_fixed_rowcols` ein Vergleich einer positiven und negativen Null erforderlich wäre). Ohne die Unterscheidung durch `car_orientation` könnten horizontale und vertikale Fahrzeuge fälschlicherweise als identisch behandelt werden. Um eine eindeutige Zuordnung der Attribute auch bei vielen Fahrzeugen innerhalb einer Rush-Hour-Konfiguration zu erleichtern, werden die Eigenschaften der Autos und LKW nicht in Arrays, sondern in Dictionaries abgebildet. Jedes Fahrzeug wird dabei durch die Fahrzeugfarbe als eindeutigen Schlüssel identifiziert.

Zusätzlich zu der initialen Startkonfiguration in Form der vorgestellten Dictionaries nimmt der Solver die Variable `move_limit` als Parameter entgegen. Sie spielt eine zentrale Rolle in der Modellierung und Lösung des Puzzles, da sie die maximale Anzahl an erlaubten Zügen und somit die Suchtiefe begrenzt. Die Einschränkung der Zuganzahl ermöglicht es dem Z3-Solver, gezielt nach der minimalen Lösung zu suchen, indem die Anzahl der tatsächlichen Bewegungen durch die Variable `moves` minimiert wird. Falls keine Lösung mit der aktuellen Obergrenze `move_limit` gefunden wird, kann der Wert iterativ erhöht werden. Dieses Verfahren stellt sicher, dass der Solver zunächst effiziente Lösungen mit wenigen Zügen bevorzugt, bevor aufwendigere Berechnungen durchgeführt werden. Insbesondere bei den Instanzen, die nur sehr wenige Züge benötigen, wird der Overhead so drastisch verringert.

Innerhalb der `z3_solver`-Funktion werden die Constraints analog zu ihrer Vorstellung in Abschnitt 4.2 dieser Arbeit implementiert. Da die Indizierung in der Informatik (Indizes beginnend bei 0) sich jedoch von der Indizierung innerhalb der dargestellten Constraints (Indizes beginnend bei 1) unterscheidet, werden stellenweise Vergleichsoperatoren angepasst. Da es sich hierbei nur um ein Implementierungsdetail handelt, das keine Auswirkungen auf die Logik des Codes bzw. der Constraints hat, wird an dieser Stelle auf die erneute Vorstellung der Constraints verzichtet.

Das Ziel des SMT-Solvers ist es, die Anzahl der Züge zu minimieren. Dazu wird die Variable `moves`, die die maximal benötigte Anzahl an Zügen darstellt, durch `s.minimize(moves)` minimiert.

5 Ergebnisse

Bevor die Ergebnisse der einzelnen Solver für das Rush-Hour-Puzzle vorgestellt werden, soll an dieser Stelle kurz darauf eingegangen werden, welche Benchmarks hierfür relevant sind. Als zentraler Faktor für die Bewertung der Leistungsfähigkeit eines Solvers wird in dieser Arbeit das Maß **Zeit** herangezogen. Dies erscheint aus folgenden Gründen sinnvoll: Während die Ergebnisse von Suchalgorithmen wie der Breitensuche oder der A*-Suche auch anhand der besuchten Knoten gemessen werden können, ist dies bei Constraint Programming nicht der Fall. Um dem vergleichenden Aspekt dieser Arbeit Rechnung zu tragen, ist ein einheitlicher Benchmark für alle Solver allerdings von hoher Relevanz. Auch für den Vergleich mit bestehender Literatur ist die Nutzung des Benchmarks Zeit insofern sinnvoll, als dass unter anderem in [9, 10] die Zeit als zentrale Maßeinheit für die Performanz der Solver genutzt wurde. Hierbei muss jedoch bedacht werden, dass die Zeit grob „linearly related to the number of nodes (i.e., less nodes implies less time)“[9, S. 960] ist.

Für die Optimierungen von A*, die lediglich für die Wahl der performantesten im Rahmen dieser Arbeit diskutierten A*-Variante relevant sind, bietet sich als Benchmark außerdem die Anzahl der besuchen Spielstände an.

Alle vorgestellten Ergebnisse wurden auf einem Apple MacBook Air mit M2 Chip (8-Core CPU mit 4 Performance-Kernen und 4 Effizienz-Kernen) und 8 GB LPDDR5 RAM (6400 MHZ) erzielt.

Wahl der Programmiersprache

Die Ergebnisse wurden durch eine Implementierung in Python erzielt. Python ist eine der populärsten Programmiersprachen für algorithmische und wissenschaftliche Analysen. Für den Vergleich von Algorithmen wie Constraint Programming, Breitensuche und A* bietet Python spezifische Vorteile, insbesondere durch die Unterstützung von genutzten Bibliotheken wie dem Z3-Solver. Die hohe Abstraktionsebene von Python ermöglicht es, Algorithmen effizient zu implementieren. Zwar verfügt Python im Vergleich zu kompilierten Sprachen wie C++ oder Java über eine insgesamt niedrigere Ausführungsgeschwindigkeit, doch dieser Nachteil wiegt im Kontext eines Algorithmenvergleiches weniger schwer: Da alle in dieser Arbeit untersuchten Solver in Python implementiert sind, bleibt die relative Vergleichbarkeit der Laufzeiten gewährleistet. Die Analyse fokussiert sich primär auf algorithmische Effizienz und nicht auf absolute Ausführungszeiten.

Wahl der Rush-Hour-Puzzle zum Testen der Solver

Um die in dieser Arbeit vorgestellten Solver in möglichst diversen Rush-Hour-Konfigurationen zu testen, ist es nötig, eine sinnvolle Auswahl an initialen Konfigurationen als Testbasis zu wählen. Aus der Rush-Hour-Datenbank von Fogleman[4] werden hierzu 100 Puzzle in den Blick genommen. Dieses *Sample1* enthält eine Auswahl von mindestens zwei Puzzles für jede mögliche Anzahl an minimalen

Zügen (ein Zug bis 51 Züge) und inkludiert somit diverse Schwierigkeitsgrade von Rush Hour. Auch das mit 51 Zügen schwierigste Level ist in dieser Auswahl initialer Spielzustände enthalten.

Um sich das Verhalten der schwerpunktmäßig betrachteten Algorithmen im Detail zu veranschaulichen, wird außerdem *Sample2* mit 1323 initialen Spielkonfigurationen eingeführt. Hierbei handelt es sich um die abschließende Menge aller Rush-Hour-Puzzle, die mit 26 Zügen zu lösen sind. Diese Zusatzuntersuchung hat den Vorteil, dass Zufallskomponenten bei der Auswahl der Instanzen keine Rolle spielen, da alle nach Fogleman[4] vorhandenen Puzzles dieser Zuggröße in den Blick genommen werden und somit keine selektive Auswahl getroffen wird.

Es bleibt anzumerken, dass die Puzzles in der Datenbank und somit auch in den Samples als String dargestellt werden. Da die hier untersuchten Algorithmen nicht mit einer Stringrepräsentation arbeiten, werden die Strings zunächst mithilfe von Parser-Funktionen in das benötigte Format umgewandelt. Diese befinden sich in der Datei `mapping.py`. Da das Mapping keine weitere Relevanz für den Vergleich hat, werden die Funktionen nicht näher vorgestellt.

5.1 A*-Suche

5.1.1 Heuristiken bei A*

Zunächst werden die verschiedenen bereits in Abschnitt 3.2.1 vorgestellten Heuristiken für die A*-Suche in den Blick genommen. Im direkten Vergleich gelöster Rush-Hour-Konfiguratinen fällt auf, dass nicht alle Heuristiken für jede Puzzleinstanz die minimale Lösung finden:

Heuristik	Schwerstes Level Zeit	Schwerstes Level Knoten	Durchschnitt Zeit	Durchschnitt Knoten	Gesamtzeit	Minimale Lösungen
constant_heuristic	2,5802	2838	5,8112	6172,98	581,12	100%
count_cars_between	2,6443	2835	5,2313	5349,47	523,13	100%
count_blocking_cars	2,4340	2328	4,8750	4292,1	487,50	76%
goal_distance	2,9411	3037	5,4765	5765,12	547,65	39%
movable_blocking_cars	2,7708	2576	5,5866	4690,33	558,66	79%

Tabelle 5.1: Vergleich verschiedener Heuristiken für A*.

Im Folgenden werden daher nur die beiden Heuristiken `constant_heuristic` und `count_cars_between` näher untersucht, die zuverlässig minimale Lösungen für das Spiel finden. In einem direkten Vergleich der beiden Heuristiken auf *Sample1* zeigt folgender Competitionplot anschaulich, dass die `count_cars_between`-Heuristik schnellere Ergebnisse liefert als die konstante Heuristik:

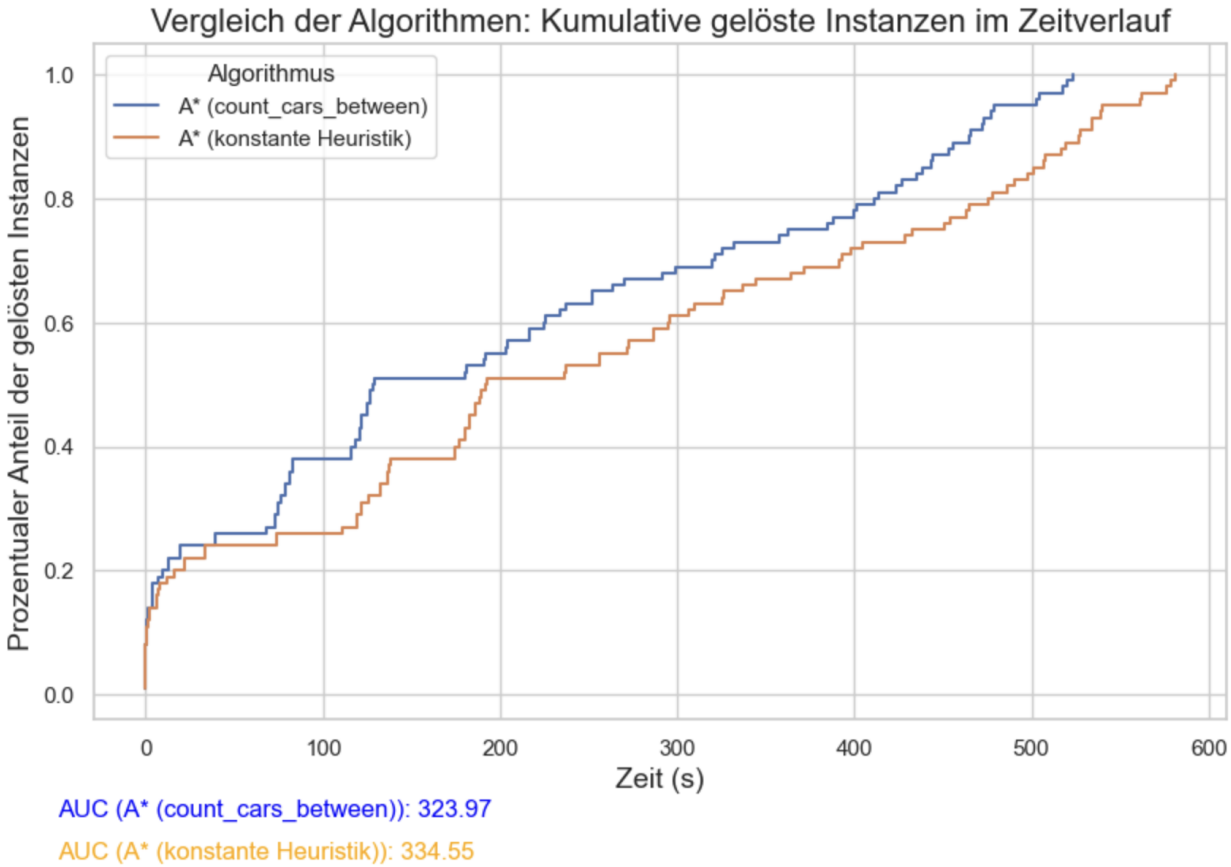


Abbildung 5.1: Grafischer Vergleich der konsistenten Heuristiken auf *Sample1*.

Es ist nicht schwierig zu erkennen, dass die blaue Kurve der `count_cars_between`-Heuristik im Zeitverlauf schneller ansteigt als die orangefarbene Kurve der konstanten Heuristik. Auffällig ist jedoch, dass die deutlich sichtbaren Plateaus beider Kurven gehäuft an denselben Stellen auf der y-Ache auftreten. Beide Heuristiken finden in allen Puzzles aus dem Sample eine minimale Lösung; gleichzeitig ist deutlich erkennbar, dass der Anstieg für die etwa ersten $20\ \%$ der Puzzle wesentlich drastischer ist als der restliche Anstieg der Kurven.

5.1.2 Tie-Breaking bei A*

Neben den Heuristiken werden im Zuge der Optimierung der A*-Suche für Rush Hour verschiedene Tie-Breaking-Strategien ebenfalls auf *Sample1* miteinander verglichen:

Tie-Breaking	Schwerstes Level Zeit	Schwerstes Level Knoten	Durchschnitt Zeit	Durchschnitt Knoten	Gesamtzeit
Kein Tie-Breaking	2,6443	2835	5,2313	5349,47	523,13
Zweistufiges Tie-Breaking	2,6841	2924	5,3197	5628,62	531,97
Dreistufiges Tie-Breaking	2,8228	3007	5,5105	5701,76	551,05

Tabelle 5.2: Vergleich verschiedener Tie-Breaking-Strategien für A*.

Die Unterschiede, die bei dem Vergleich festgestellt wurden, sind sowohl was die durchschnittliche Anzahl der besuchten Zustände pro Level als auch die durchschnittliche Lösungszeit pro Puzzle an-

geht marginal, die klassische A*-Variante ohne dedizierte Tie-Breaking-Strategie löst die Puzzle aus dem Sample dabei am schnellsten.

5.1.3 BFS im Vergleich

Um die Performanz von A* als graphbasiertem Suchalgorithmus bei der Lösung von Rush Hour besser einschätzen zu können, ist der Vergleich mit der ebenfalls graphbasierten Breitensuche, die in Abschnitt 3.1 diskutiert wird, auf *Sample1* naheliegend.

Algorithmus	Schwerstes Level Zeit	Schwerstes Level Knoten	Durchschnitt Zeit	Durchschnitt Knoten	Gesamtzeit
A*	2,6443	2835	5,2313	5349,47	523,13
BFS	2,7013	3025	7,3432	7699,9	734,32

Tabelle 5.3: Tabellarischer Vergleich von A*-Suche und BFS.

Es fällt auf, dass A* die Sampleinstanzen von Rush Hour durchschnittlich ca. $2.1\ s$ schneller löst als die Breitensuche und dabei auch deutlich weniger Zustände besucht. Es gibt aber Puzzle, wie zum Beispiel die schwierigste Rush-Hour-Instanz, bei der der Unterschied zwischen den beiden Suchalgorithmen quasi zu vernachlässigen ist. Ein Competitionplot verdeutlicht die Unterschiede zwischen den beiden Algorithmen deutlicher:

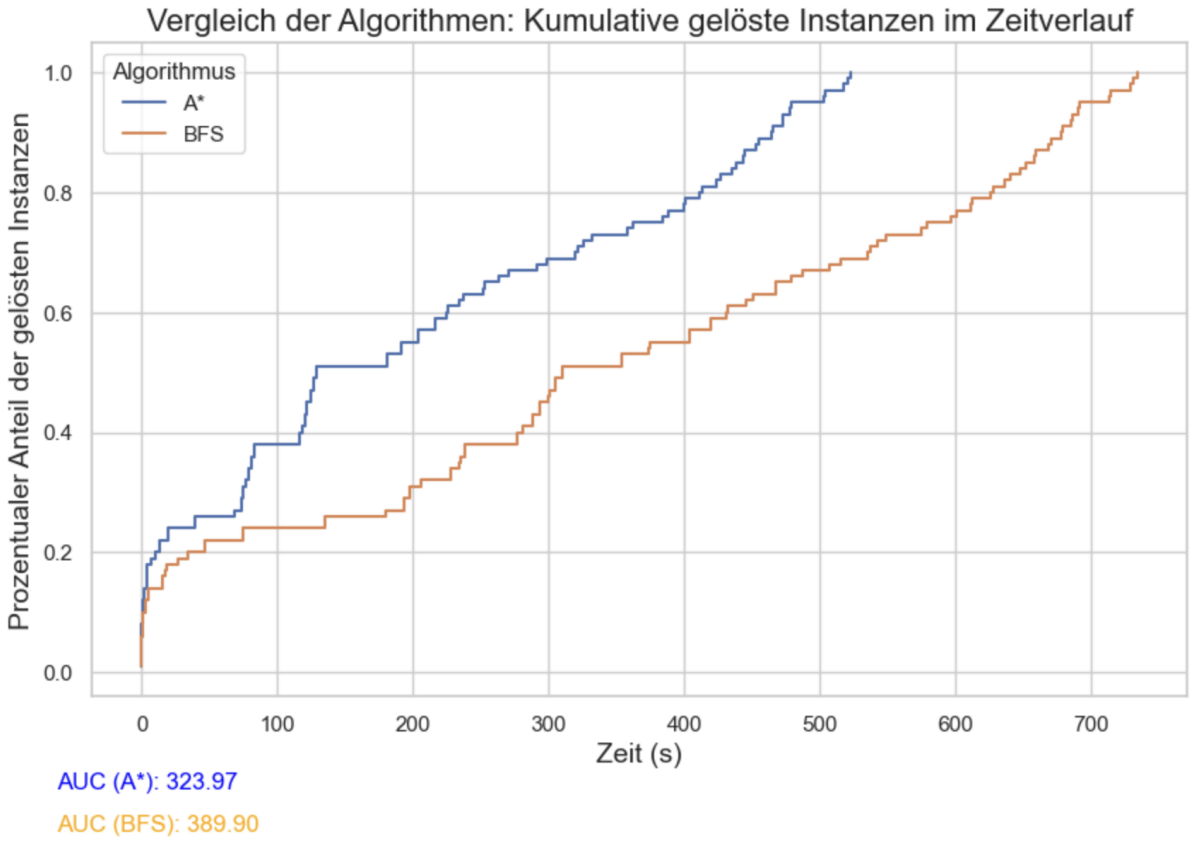


Abbildung 5.2: Grafischer Vergleich von A*-Suche und BFS.

Der Kurvenverlauf der beiden Kurven ist sehr ähnlich, in ihrer Form unterscheiden sie sich insbesondere durch eine Streckung der Kurve der Breitensuche in x-Achsenrichtung: Es ist auf den ersten Blick erkennbar, dass die Breitensuche an denselben Stellen auf der y-Achse wesentlich längere Plateaus besitzt als die A*-Suche.

5.2 Constraint Programming

5.2.1 Optionale Constraint

Bei der Implementierung der Constraints für Rush Hour ist Constraint 4.11 zum Zwecke einer möglichen Optimierung eingeführt worden. Das Diagramm zeigt das Verhalten des Z3-Solvers mit Anwendung der Constraint und ohne auf *Sample1*:

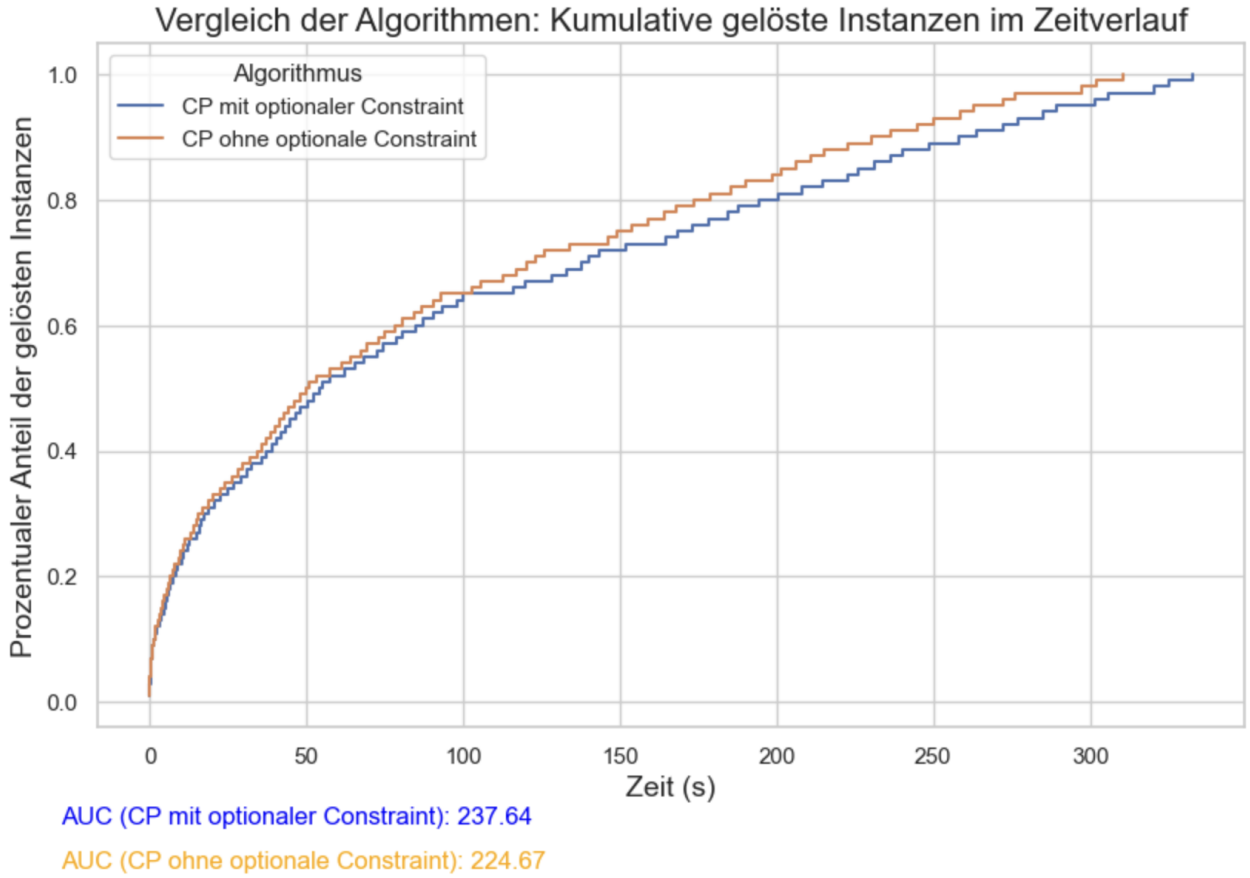


Abbildung 5.3: Grafischer Vergleich von CP mit optionaler Constraint und ohne.

Das Verhalten des Solvers bleibt unter Berücksichtigung der optionalen Constraint weitgehend unverändert im Vergleich zu seiner Ausführung ohne sie; die beiden Kurven ähneln sich stark. Eine im Verlauf beinahe sigmoidale Kurvenentwicklung ist erkennbar. Die orangene Kurve (Constraint Programming ohne optionale Constraint) schließt die Berechnungen geringfügig schneller ab.

5.2.2 Iterative Erhöhung

Da die Implementation des Solvers wie in Abschnitt 4.2.1 vorgestellt mit einer Variable `move_limit` arbeitet, die die maximale Anzahl von Zügen für die Lösung eines Rush-Hour-Puzzles im Vorhinein beschränkt, kann es bei Instanzen, die viele Züge benötigen, nötig sein, das Zuglimit iterativ zu erhöhen und den Solver infolgedessen mehrfach aufzurufen. Das Diagramm zeigt die Auswirkungen, wenn man die Variable `move_limit` iterativ erhöht oder den Solver auf *Sample1* statisch mit der maximal möglichen Anzahl an Zügen ($51 + 1$ aus Implementierungsgründen) ausführt:

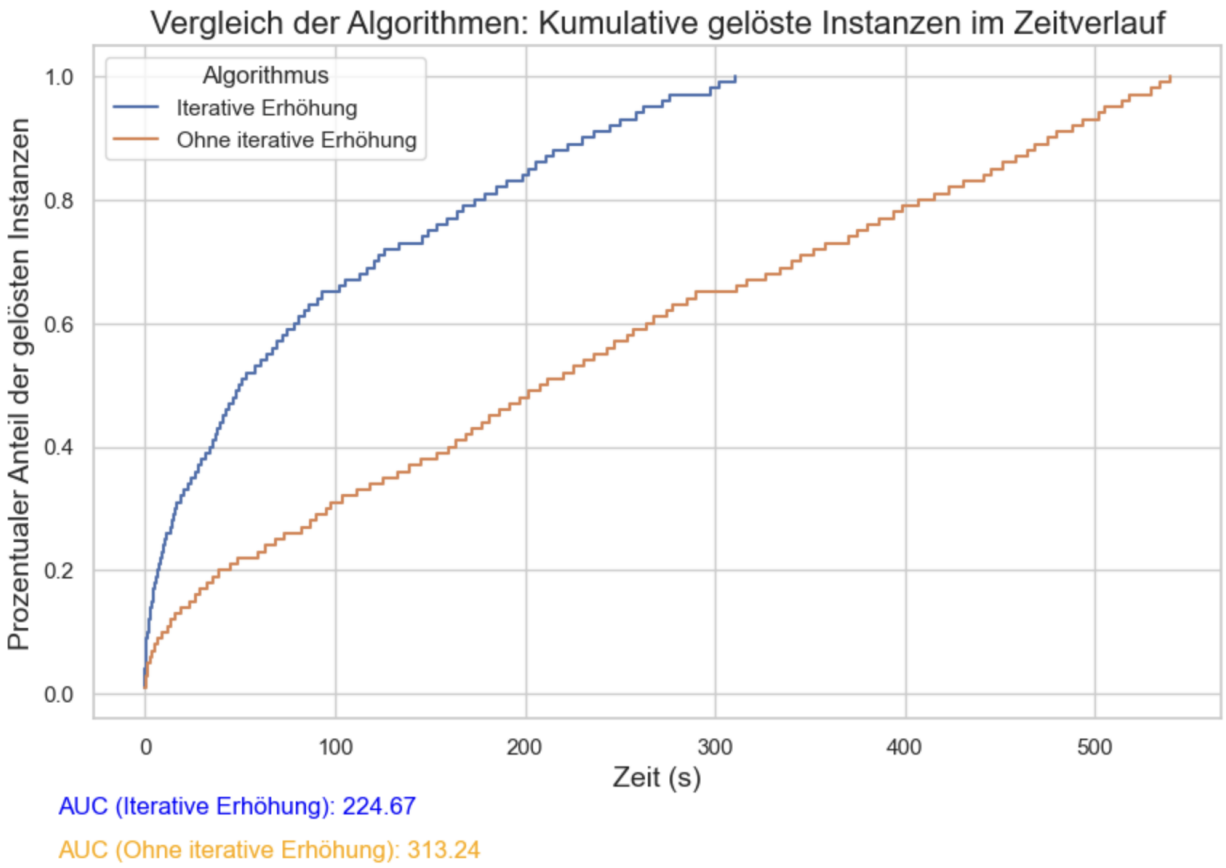


Abbildung 5.4: Auswirkung der iterativen Erhöhung von `move_limit`.

Ein deutlicher Unterschied zwischen den beiden Kurven ist erkennbar. Während der Constraint-Programming-Ansatz mit statischem Zuglimit durch einen beinahe linearen Anstieg der Kurve charakterisiert ist, zeigt sich bei der iterativen Erhöhung von `move_limit` zunächst ein wesentlich stärkerer Anstieg gelöster Instanzen.

Details der Auswirkungen lassen sich auch in der tabellarischen Gegenüberstellung der Resultate auf *Sample1* ablesen:

Algorithmus	Schwerstes Level Zeit	Durchschnitt Zeit	Gesamtzeit
CP mit iterativer Erhöhung	0,0921	3,1040	310,40
CP ohne iterative Erhöhung	0,0707	5,3942	539,42

Tabelle 5.4: Tabellarischer Vergleich von statischem und iterativ erhöhtem CP.

Bei *Sample2*, das diverse Instanzen derselben Schwierigkeit bzw. minimalen Zuglänge betrachtet, verhält es sich allerdings umgekehrt:

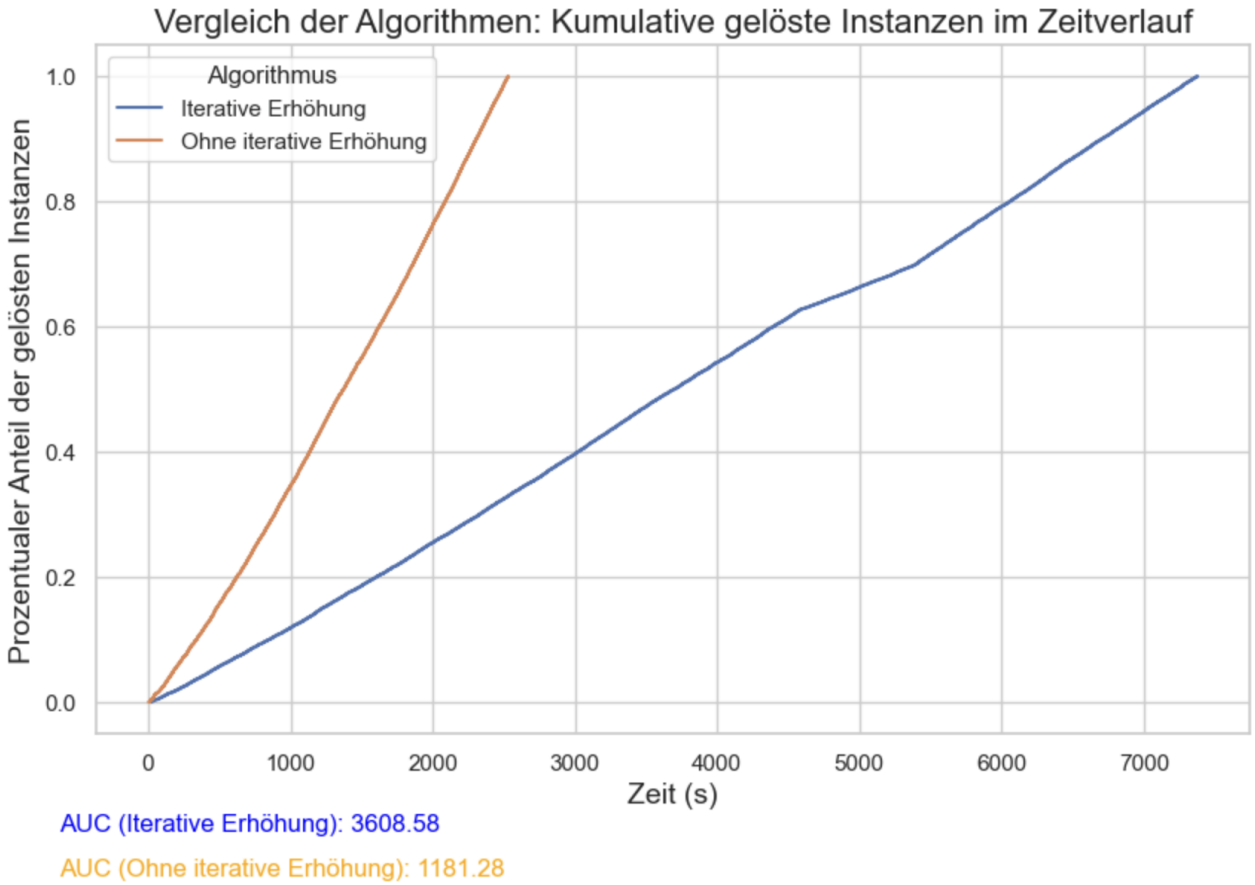


Abbildung 5.5: Vergleich von statischem und iterativ erhöhtem CP bei konstanter Schwierigkeit.

5.3 A* vs. Constraint Programming

Für dem finalen Vergleich der beiden in dieser Arbeit schwerpunktmäßig untersuchten Algorithmen werden die jeweils performantesten Varianten herangezogen. Bei der A*-Suche wird daher im Folgenden die Variante ohne Tiebreaking mit der Heuristik `count_cars_between` betrachtet und die Constraint-Programming-Lösung kommt ohne die optionale Constraint aus.

5.3.1 Rush-Hour-Instanzen unbekannter Schwierigkeit - *Sample1*

Sample1 wird in Kombination mit der iterativen Erhöhung des Constraint-Programming-Ansatzes für einen Vergleich beider Solver in einer realistischen, im Vorhinein unbekannt schwierigen Rush-Hour-Spielsituation angeführt:

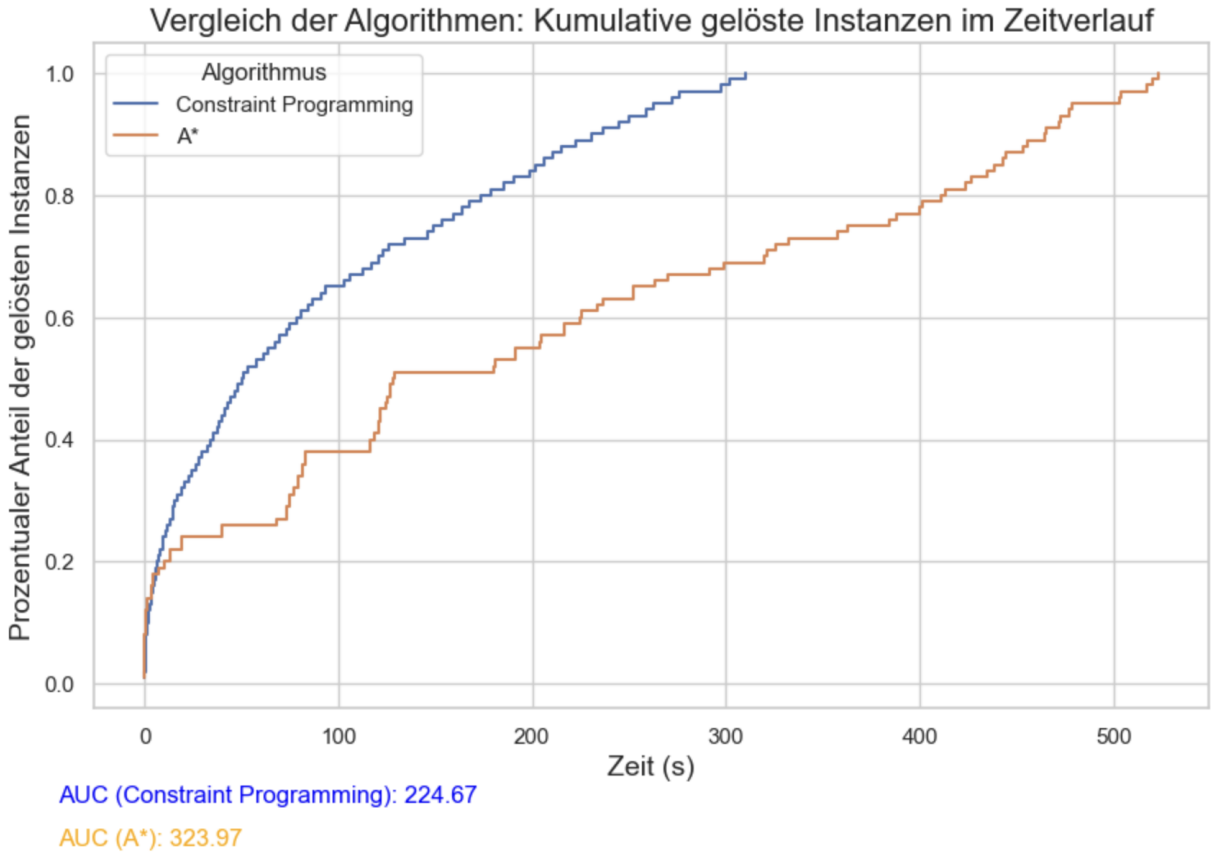


Abbildung 5.6: Vergleich von A* und Constraint Programming.

Es fällt auf, dass die Constraint-Programming-Kurve wesentlich stärker ansteigt als die Kurve der A*-Suche. Bei letzterer sind zudem wesentlich längere und häufigere Plateaus zu erkennen. Dass es bei A* mehr Ausreißer gibt, lässt sich auch im entsprechenden Boxplot ablasen:

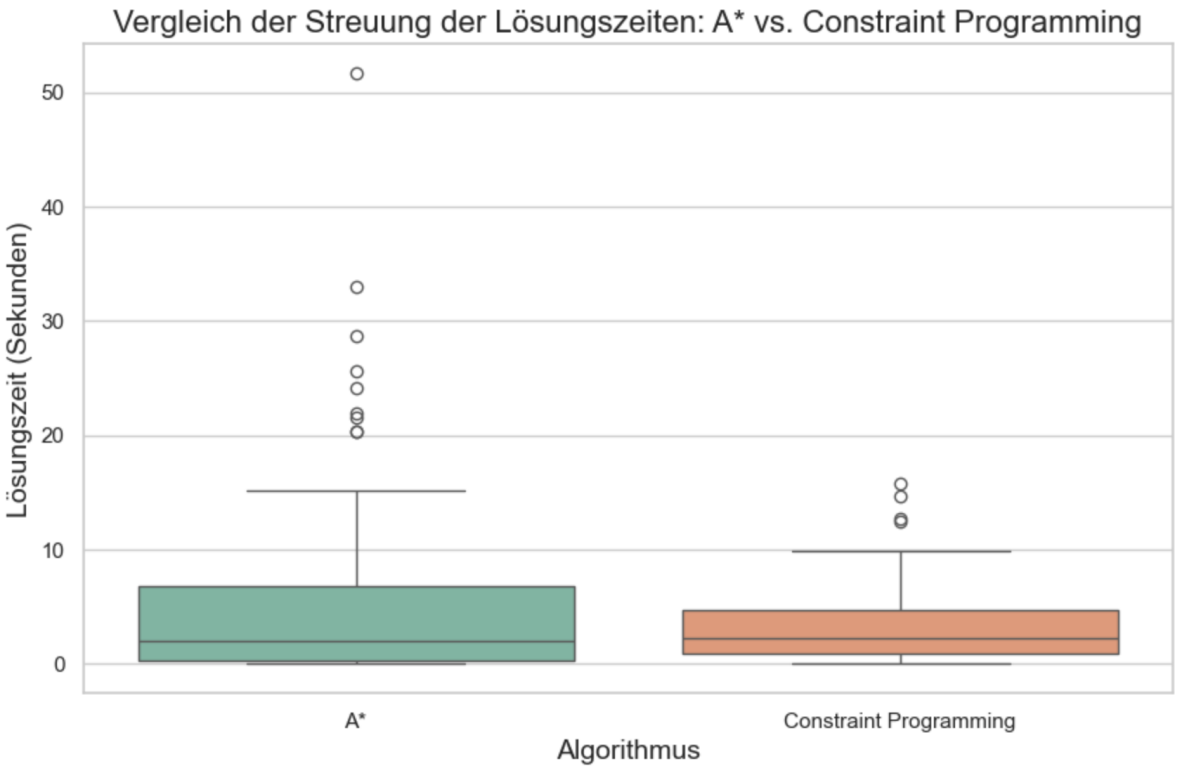


Abbildung 5.7: Streuung bei A* und Constraint Programming.

Der Interquartilsabstand bei der A*-Suche ist hier nicht nur beinahe doppelt so groß, auch die Antenne ist wesentlich länger. Der Median bei beiden Algorithmen ist nahezu identisch, wenngleich er bei dem auf Constraint Programming basierenden Solver minimal höher liegt.

In Scatterplots desselben Sachverhaltes fällt auf, dass die meisten Datenpunkte im oberen Bereich der Diagonale $x = y$ liegen, der Ansatz mit Constraint Programming also für mehr Rush-Hour-Konfigurationen langsamer ist als A*. Dies ist insbesondere bei den Puzzleinstanzen, in denen die minimalen Lösungen innerhalb der ersten $2\ s$ Sekunden gefunden werden, der Fall:

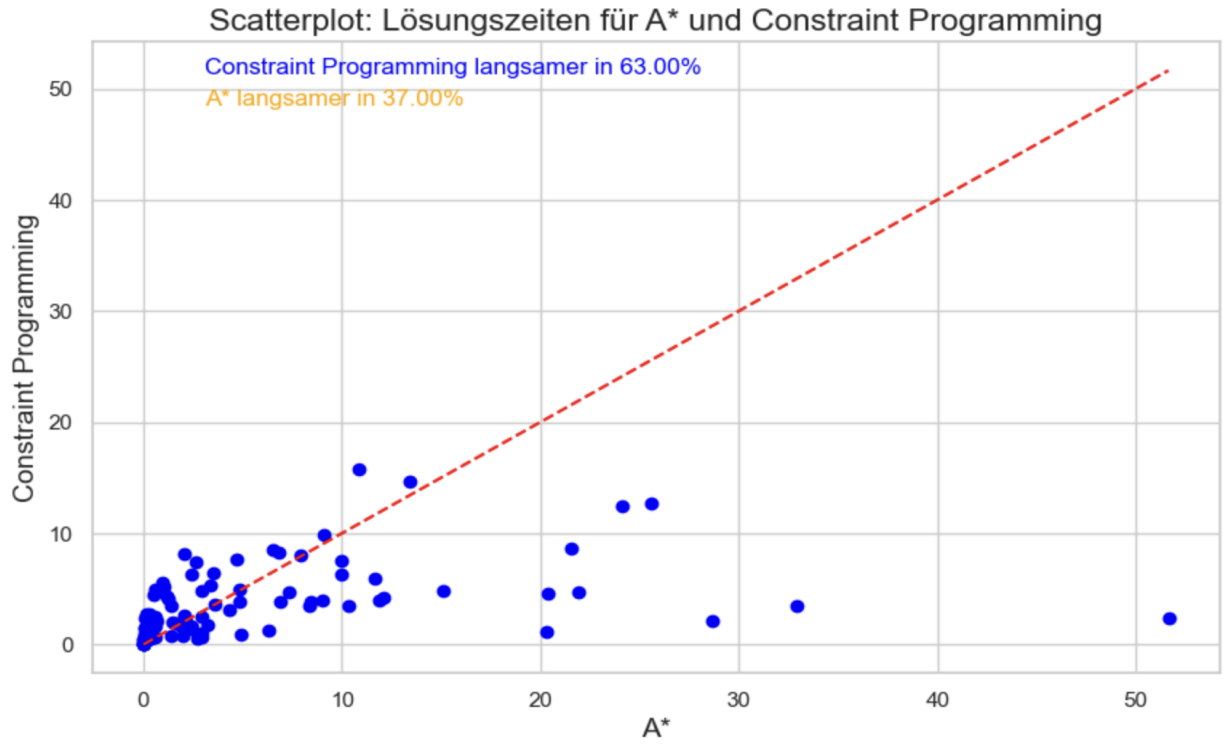


Abbildung 5.8: Lösungszeiten von A* und Constraint Programming im Vergleich.

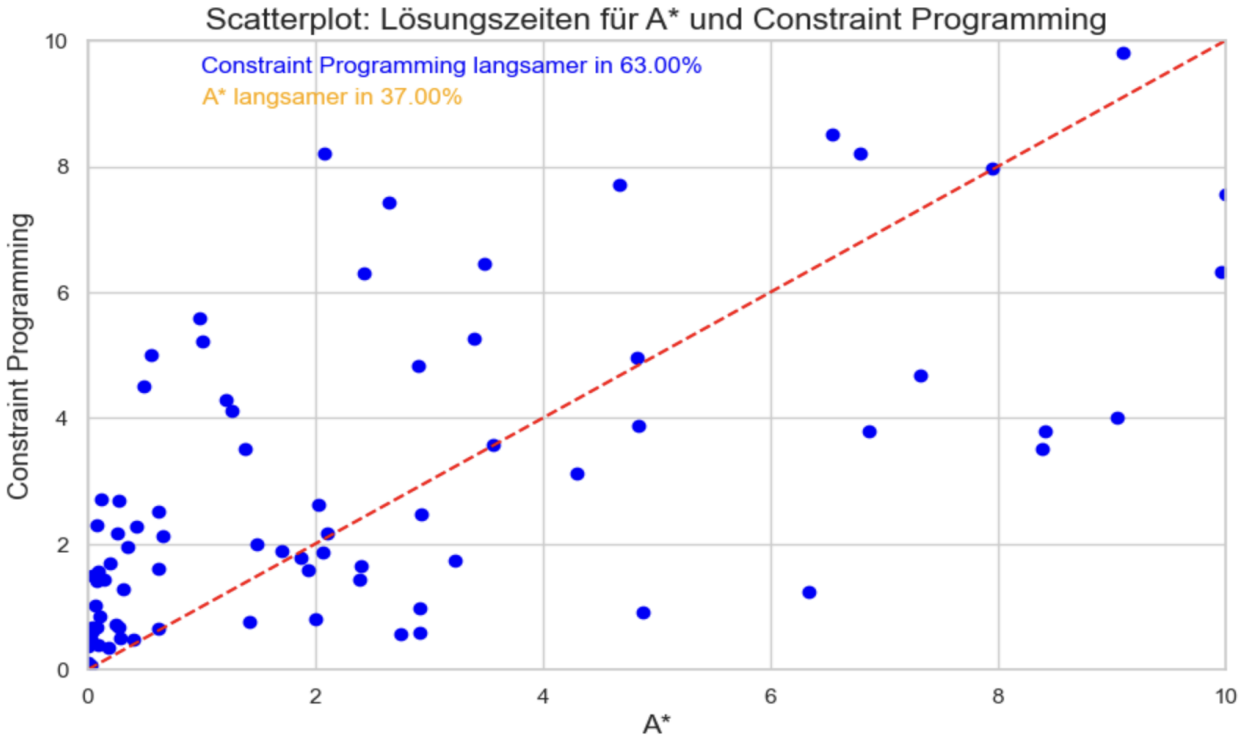


Abbildung 5.9: Lösungszeiten von A* und Constraint Programming im Detailvergleich.

5.3.2 Rush-Hour-Instanzen bekannter Schwierigkeit - *Sample2*

Das Verhalten beider Lösungsansätze, wenn die Schwierigkeit (also die Anzahl der Züge) des Rush-Hour-Levels im Vorhinein bekannt ist, kann an folgendem Diagramm abgelesen werden:

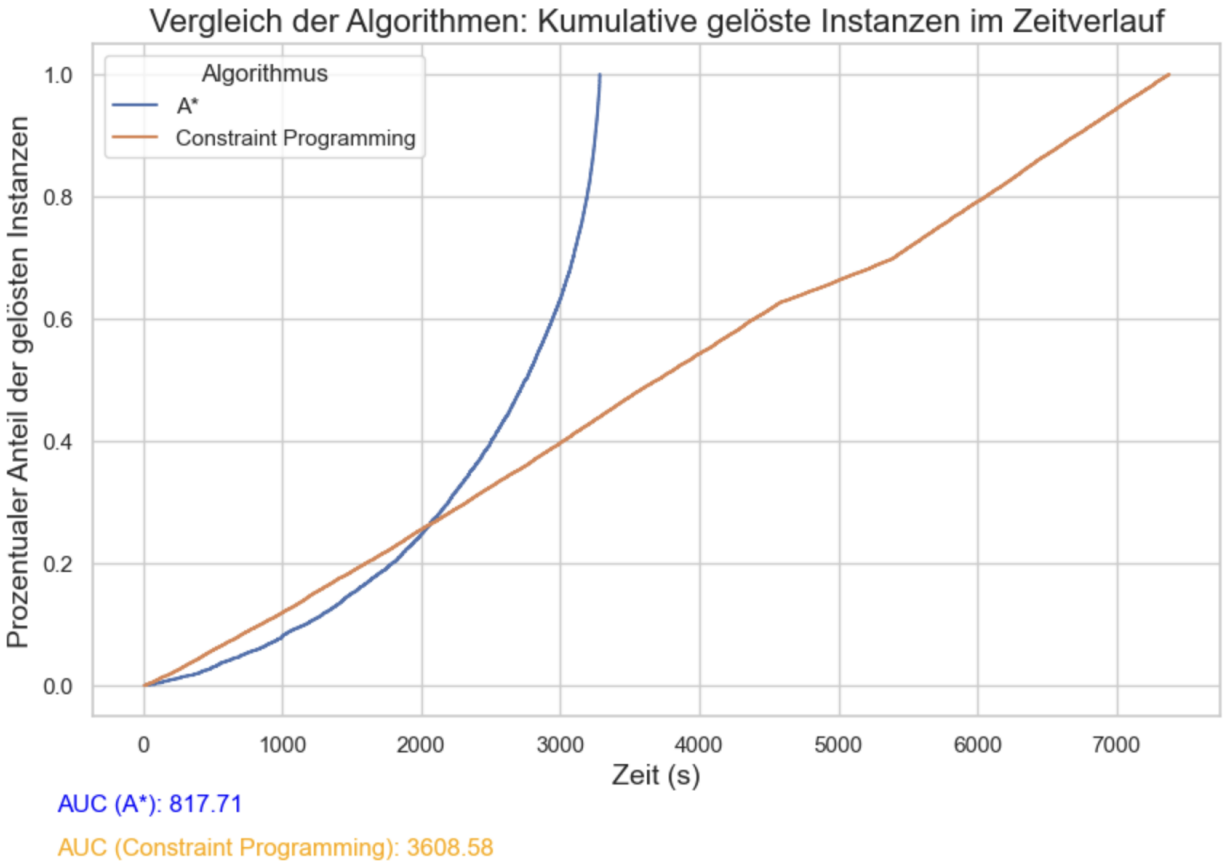


Abbildung 5.10: A* und iteratives Constraint Programming auf Puzzles bekannter Schwierigkeit.

Es ist abzulesen, dass die Kurve des A*-Algorithmus exponentiell steigt, wohingegen die Kurve des Constraint-Programming-Solvers sich durch einen linearen Anstieg auszeichnet. Die A*-Kurve erreicht dabei schneller den maximalen y-Wert.

Betrachtet man kontrastierend dazu die statische Variante von Constraint Programming, die die maximale Zuganzahl als Limit übergeben bekommt, ergibt sich folgender Plot:

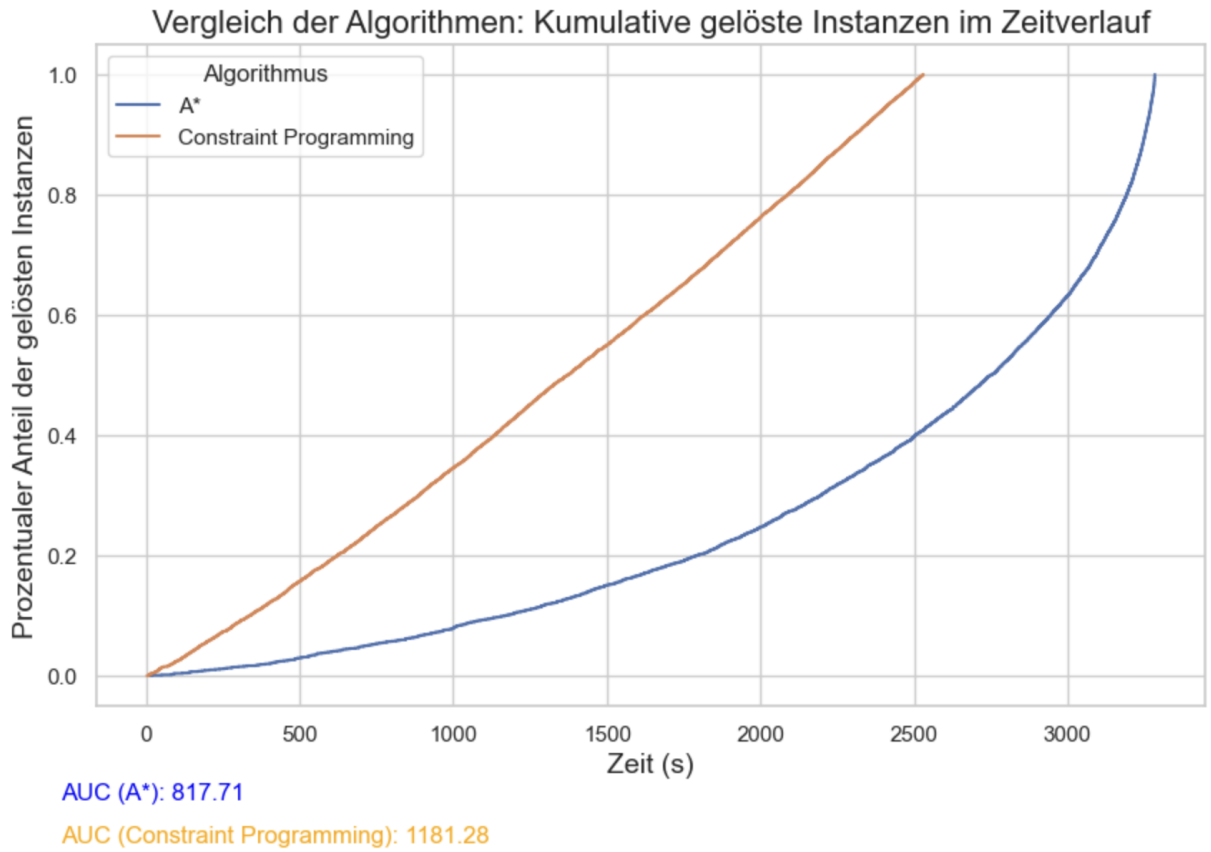


Abbildung 5.11: A* und statisches Constraint Programming auf Puzzles bekannter Schwierigkeit.

Der Kurvenverlauf beider Kurven ändert sich nicht; jedoch ist deutlich zu erkennen, dass der statische Constraint-Programming-Ansatz wesentlich früher mit der Lösung aller Levels fertig ist und den maximalen y-Wert erreicht. Ein Blick auf einen tabellarischen Vergleich beider Algorithmen bekräftigt diese Beobachtung:

Algorithmus	**Schwerstes Level Zeit**	**Durchschnitt Zeit**	**Gesamtzeit**
A*	12,1371	2,4825	3284,27
Statisches CP	2,6003	1,9126	2530,31

Tabelle 5.5: Tabellarischer Vergleich von A* und statischem Constraint Programming.

6 Diskussion

6.1 A*-Suche

6.1.1 Heuristiken bei A*

In Abschnitt 5.1.1 hat sich gezeigt, dass nicht jeder der vorgestellten Versuche, eine Heuristik für die Lösung von Rush Hour mit der A*-Suche zu entwickeln, sinnvoll ist: Drei der fünf untersuchten Ansätze einer Heuristik für Rush Hour finden nicht immer die minimalen Züge. Einer Heuristik, die nicht die minimalen Züge findet, mangelt es an Konsistenz, weswegen ein Blick auf die Konsistenz der einzelnen Heuristiken geworfen wird:

- **`constant_heuristic`**: Diese Heuristik gibt entweder 0 (falls das rote Auto nicht blockiert ist) oder 1 (falls es blockiert ist) zurück. Da sich der Wert der Heuristik nur um maximal 1 ändern kann und die Kosten für jede Bewegung mindestens 1 betragen, bleibt die Konsistenzbedingung stets erfüllt.
- **`count_cars_between`**: Die Heuristik zählt die Anzahl der Autos, die direkt zwischen dem roten Auto und der Ausfahrt stehen. Jedes dieser Autos muss mindestens einmal bewegt werden, um das Ziel zu erreichen. Da in einem einzelnen Zug höchstens ein Auto bewegt wird, kann sich $h(n)$ um höchstens 1 verringern, sodass gilt:

$$h(n) \leq 1 + h(n')$$

- **`goal_distance`**: Diese Heuristik misst nur die Entfernung des roten Autos zur rechten Seite des Rush-Hour-Spielbretts, ignoriert aber Blockaden. Es ist möglich, dass ein Auto bewegt wird, wodurch sich die Heuristik nicht ändert oder sogar erhöht, was gegen die Konsistenz verstößt.
- **`count_blocking_cars`**: Da indirekt blockierende Autos gezählt werden, kann es vorkommen, dass sich die Heuristik durch eine Bewegung stärker verändert als die tatsächlichen Kosten des Zuges.
- **`movable_blocking_cars`**: Diese Heuristik enthält eine Strafbewertung für unbewegliche Autos, was zu sprunghaften Änderungen führen kann.

Von den beiden konsistenten Heuristiken ist die Heuristik `count_cars_between` performanztechnisch zu bevorzugen. Sie besucht im Schnitt gut 800 States weniger pro Puzzle und kann dadurch die minimale Lösung durchschnittlich 0.5 s schneller finden als die konstante Heuristik. Dies ist wenig überraschend, da die konstante Heuristik nahezu keine Informationen über den tatsächlichen

Spielsachverhalt in die A*-Suche miteinfließen lässt. Der Unterschied zwischen der konstanten Heuristik und der Heuristik `count_cars_between` ist indes nicht besonders schwerwiegend. Teilweise existieren sogar Instanzen, bei denen die konstante Heuristik performanter ist, wie zum Beispiel das schwierigste Rush-Hour-Level. Die Inkonsistenz von Rush-Hour-Heuristiken bei verschiedenen Puzzles wurde schon von Hauptman et al. festgestellt und ist daher nicht überraschend[9]. Da das Entwickeln neuer, sinnvoller Heuristiken für Rush Hour insbesondere deshalb nicht trivial ist, da „standard methods for deriving heuristics, such as solving either sub-problems [...], or relaxed problems (e.g. using the Manhattan distance heuristic [...])“[9, S. 956] nicht direkt auf das Puzzle anwendbar sind, wurde in den Vergleichen, die den Schwerpunkt dieser Arbeit bilden, auf die Heuristik `count_cars_between` zurückgegriffen. Es bleibt zu erwähnen, dass performantere konsistente Heuristiken existieren[9], die allerdings den Rahmen dieser Arbeit sprengen.

Überraschenderweise fällt die nicht-konsistente Heuristik `count_blocking_cars` durch eine schnellere Ausführungsgeschwindigkeit auf, was direkt darauf zurückzuführen ist, dass sie durchschnittlich ca. 1000 Spielstände weniger pro Puzzle betrachtet. Eine mögliche Erklärung hierfür könnte sein, dass die Heuristik einen größeren h-Wertebereich hat ($[0$, Anzahl Autos - $1]$) und somit in mehr Fällen eindeutig definiert ist, welcher Zustand vom Algorithmus als nächstes aus der Prioritätenwarteschlange entnommen werden muss. Da die Heuristik aber nur in $76\ \%$ der Fälle zu einer minimalen Lösung führt, wird sie nicht näher untersucht.

In Abbildung 5.1 erkennt man die Vorteile von `count_cars_between` gegenüber der konstanten Heuristik. Obwohl der Kurvenverlauf sich stark ähnelt, beträgt die Area-Under-Curve (AUC) für die `count_cars_between`-Heuristik $323,97$ und ist damit $3,16\ \%$ kleiner als die AUC der konstanten Heuristik, was die bessere Performanz von `count_cars_between` unterstreicht. Aus diesem Grund werden sich die weiteren Ausführungen nach diesem Abschnitt nur noch mit der `count_cars_between`-Heuristik beschäftigen.

Interessanterweise können für beide Heuristiken bei denselben Rush-Hour-Puzzles Ausreißer beobachtet werden, was dafür spricht, dass die grundsätzliche Problematik dieser Level mit einer hohen Anzahl an von A* besuchten Spielzuständen zusammenhängt, die durch die Heuristiken nur geringfügig eingeschränkt wird. Dies führt bei graphbasierten Suchalgorithmen im Allgemeinen zu einer geringeren Ausführungsgeschwindigkeit, da mehr Knoten exploriert werden. Der starke Anstieg, den beide Kurven bei den ersten $20\ \%$ der Level machen, ist dadurch zu erklären, dass diese ersten Instanzen aus dem Sample maximal zehn Züge zur Lösung benötigen. Somit müssen naheliegenderweise durchschnittlich deutlich weniger Zustände erkundet werden als es bei späteren Levels mit bis zu 51 minimalen Zügen der Fall ist.

6.1.2 Tie-Breaking bei A*

Ein weiterer Optimierungsversuch besteht in der Nutzung verschiedener Tie-Breaking-Strategien wie in Abschnitt 3.2.2 beschrieben. Hierbei wird in Tabelle 5.2 jedoch deutlich, dass diese nur sehr geringe Auswirkungen auf die Laufzeit haben und die A*-Suche bei Rush Hour sogar ohne dedizierte Tie-Breaking-Strategie minimal besser performt. Eine von Corrêa1 et al. in [21] vorgeschlagene Tie-Breaking-Strategie, die immer optimal expandiert, existiert zwar laut ebendiesen für jedes A*-Problem, erfordert jedoch eine kostenadaptive Heuristik, die in dieser Arbeit für Rush Hour wie

bereits gezeigt nicht vorliegt. Da der mögliche Wertebereich der `count_cars_between`-Heuristik $[0, 4]$ nur klein ist, variieren die h-Werte nur geringfügig, was dazu führt, dass Tie-Breaking-Strategien, die die h-Werte als Entscheidungskriterium heranziehen, im Schnitt nur wenig zielführend sind. Da allgemein „tiebreaking based on the heuristic value [...] not necessary to achieve good performance“[22] ist, ziehen beide der hier untersuchten Implementierungen einer Tie-Breaking-Strategie auch den g-Wert heran. Dieser ist naturgemäß in Fällen wie Rush Hour durch die maximale Zuganzahl begrenzt und zeichnet sich deshalb je nach Startkonfiguration des Puzzles ebenfalls durch eine verhältnismäßig geringe Variannz aus. Bessere Ergebnisse wurden erzielt, wenn Zustände mit höheren g-Werten bevorzugt wurden. Da sich diese Arbeit schwerpunktmäßig mit einem Vergleich von A* und Constraint Programming beschäftigt, wird an dieser Stelle jedoch kein weiteres Schlaglicht auf die Tie-Breaking-Strategien geworfen, sondern die Variante der A*-Suche ohne Tie-Breaking-Strategie behandelt, da diese im Falle der hier thematisierten Lösung von Rush Hour zu den besten Ergebnissen führt.

6.1.3 BFS im Vergleich zu A*

Da die herkömmlichen Optimierungsmaßnahmen der A*-Suche wie in den Abschnitten 6.1.1 und 6.1.2 zu nur geringfügigen Verbesserungen führten, ist es zusätzlich interessant, sich einen Vergleich zwischen der wesentlich trivialeren Breitensuche und der (optimalsten) A*-Suche anzusehen. In Tabelle 5.3 fällt auf, dass A* trotz der nur geringen Optimierungen in dieser Arbeit immer noch durchschnittlich ungefähr 2350 Spielstände pro Level weniger besucht als die Breitensuche. Das lässt sich dadurch erklären, dass selbst in der trivialsten Version von A* ohne jedwede Optimierung der g-Wert bei Entscheidungen herangezogen wird, indem er direkt in den f-Wert einfließt und somit die Entnahme von Werten aus der Prioritysqueue beeinflusst. Bei einer einfachen Breitensuche ist dies nicht der Fall - auch wenn diese Optimierungen wie im hier diskutierten Fall eine Datenstruktur, die besuchte Zustände speichert, enthält[16].

6.2 Constraint Programming

6.2.1 Optionale Constraint

Die Abbildung 5.3 zeigt, dass die Nutzung der optionalen Constraint 4.11, die nur zu Optimierungszwecken eingeführt wurde, den Solver verlangsamt. Es fällt zwar auf, dass die Kurven im Diagramm einen ähnlichen Verlauf inklusive Plateaus auf derselben Höhe haben, dennoch liegt der Ansatz mit der Optimierungsconstraint performanztechnisch hinten und ist, wenn man die AUC-Werte ins Verhältnis setzt, etwa $5,46\ \%$ langsamer als die Variante ohne Constraint. Dieses Ergebnis überrascht, denn Cian et al. haben die Constraint bewusst zum Zwecke der Symmetriebrechung eingeführt[10]. Sie soll also verhindern, dass redundante Lösungen mehrfach vom Solver untersucht werden. Symmetrie wird auch in der Literatur als „key problem“[23, S. 329] von Constraint Programming angeführt[24], weshalb eine Constraint zur Verhinderung dessen sinnvoll erscheint. Eine mögliche Erklärung, warum die Constraint im hier diskutierten Falle nicht zur Optimierung beiträgt, ist die Wahl des Solvers Z3. Dieser ist für „linear optimization problems over SMT formulas, MaxSMT, and their combinations“[25, S. 1] entworfen worden. Da es sich bei der diskutierten Constraint jedoch um eine nichtlineare Constraint handelt, ist es denkbar, dass sie die Lösungsfindung erschwert und eine Effizienzsteigerung verhindert.

6.2.2 Iterative Erhöhung von `move_limit`

Bei der Betrachtung von Abbildung 5.4 fällt auf, dass die iterative Erhöhung des Zuglimits einen enormen Einfluss auf die Performanz des Constraint-Programming-Ansatzes hat. Der AUC-Wert der iterativen Variante ist $28.28\ \%$ kleiner als derjenige ohne und aus Tabelle 5.4 lässt sich ablesen, dass sich dies durch eine durchschnittlich Zeitersparnis von $2,2\ s$ pro Level äußert. Diese Effizienzsteigerung ist vor dem Hintergrund, dass das Sample diverse Rush-Hour-Puzzle, die verschieden viele Züge benötigen, enthält, nur wenig verwunderlich: Der Overhead bei allen Puzzles, die deutlich weniger als 51 Züge benötigen – somit bei allen außer den schwierigsten – ist durch die Erstellung der entsprechenden Variablen und Constraints in von der maximalen Zuganzahl abhängigen `for`-Schleifen enorm. Zwar muss für Instanzen, die eine mittlere Anzahl von Zügen benötigen, der Solver mehrfach mit unterschiedlichen Zuglimits aufgerufen werden; da man sich der Lösung aber von unten nähert, ist der Overhead vergleichsweise gering. Insbesondere für die Rush-Hour-Puzzles, die sich durch wenige Züge zur Lösung auszeichnen, ist der Anstieg wie in Diagramm 5.4 erkennbar bei dem Ansatz mit iterativer Erhöhung extrem stark. Hier muss der Solver dank initialem `move_limit` $m = 5$ nur ein mal laufen und findet für alle Instanzen mit bis zu vier Zügen eine Lösung. Der Overhead ist am größten, wenn nur ein Zug benötigt ist, und beträgt dann

$$\frac{m-1}{1} \times 100 = \frac{5-1}{1} \times 100 = 4 \times 100 = 400\ \%.$$

Betrachtet man hingegen den Overhead, der bei einem statischen Zuglimit von 52 für dieselbe Zuganzahl entsteht, fällt auf, dass dieser signifikant höher ist:

$$\frac{m-1}{1} \times 100 = \frac{52-1}{1} \times 100 = 51 \times 100 = 5100\ \%$$

Nichtsdestotrotz wird deutlich, dass die iterative Erhöhung des Zuglimits nur in den Fällen sinnvoll ist, in denen eine Streuung der Züge bis zur Lösung vorliegt oder ioder wenn im Voraus nicht bekannt ist, wie viele Züge zur Lösung benötigt werden (was bei Spielen wie Rush Hour allerdings typischerweise der Fall ist).

Betrachtet man nämlich die Tabelle 5.4, so fällt auf, dass das schwierigste Level in der Variante mit statischem Zuglimit ungefähr $31\ \%$ schneller gelöst wird als mit iterativer Erhöhung. Dies ist durch das perfekt auf die Zuganzahl des schwierigsten Levels abgestimmte Zuglimit einfach zu erklären und verdeutlicht sich, wenn wie in Abbildung 5.5 ein Sample mit konstanter Zuganzahl angesehen wird. Hier wird zum einen deutlich, dass Constraint Programming – ganz gleich, ob mit iterativer Erhöhung des Zuglimits oder nicht – größtenteils konstante Ausführungsgeschwindigkeiten bei Leveln gleicher Schwierigkeit hat. Dies ist der Grund für den beinahe linearen Anstieg beider Kurven mit wenigen und vernachlässigbaren Ausreißern, die sich in Form von Plateaus äußern. Dass der Ansatz mit statischem Zuglimit eine Performanzsteigerung von etwa $200\ \%$ mit sich bringt, ist folgendermaßen zu erklären: Während der statische Solver mit `move_limit = 27` für jede der 1323 Puzzleinstanzen aus dem Sample, die sich mit 26 Zügen lösen lassen, nur ein mal ausgeführt werden muss, wird die `move_limit`-Variable in der Implementierung der iterativen Variante drei mal angepasst, was die Ausführung natürlich verlangsamt:

1. Startwert: `move_limit = 5`

2. Erste Erhöhung: `move_limit += 10` → 15
3. Zweite Erhöhung: `move_limit += 10` → 25
4. Dritte Erhöhung: `move_limit += 10` → 35

6.3 A* vs. Constraint Programming

6.3.1 Rush-Hour-Instanzen unbekannter Schwierigkeit - *Sample1*

Mit einem Blick auf Abbildung 5.7 fällt auf, dass Constraint Programming bei Rush-Hour-Puzzles unbekannter Schwierigkeit eine allgemein sinnvollere Wahl ist. Der AUC-Wert des Solvers, der Constraint Programming nutzt, ist um etwa $44\ \%$ geringer als der des konkurrierenden A*-Solvers und lässt somit einen deutlichen Performanzunterschied erkennen. Dieser korreliert auch mit der allgemein geringeren Lösungsgeschwindigkeit; während A* erst nach ca. $523\ s$ alle Puzzle gelöst hat, ist Constraint Programming bereits nach ca. $311\ s$ fertig. Der Grund hierfür ist ebenfalls in der Abbildung ersichtlich, denn die A*-Suche stößt im Vergleich auf wesentlich längere Plateaus, hält sich also an einigen Puzzles sehr lange auf, während der Constraint-Programming-Solver eine geringere Streuung der Lösungszeiten für jedes Puzzle aus *Sample1* aufweist.

Dies lässt sich auch im Boxplot 5.7 ablesen: Bei dem A*-Solver liegen wesentlich mehr Datenpunkte oberhalb der Antenne, was eine insgesamt größere Streuung der Lösungszeiten bei der Nutzung von A* impliziert. Auch der Whisker selbst ist länger als bei einer Lösung von Rush Hour mit Constraint Programming. Neben dem Whisker kann der Interquartilsabstand als Maß für die Streuung herangezogen werden; auch hier ist erkennbar, dass die grüne Box (A*) beinahe doppelt so hoch ist wie die orangefarbene (Constraint Programming). Erklären lässt sich das unter anderem mit einem direkten Blick auf die Sampledaten: die Zeit, die der A*-Solver für die Lösung eines Levels benötigt, hängt davon ab, wie viele Spielstände vom Algorithmus besucht werden müssen, bevor eine minimale Lösung gefunden wird. Aufgrund der bereits erwähnten PSPACE-Vollständigkeit von Rush Hour[6] kann die Anzahl der möglichen Spielzustände exponentiell wachsen, was eine potentiell starke Knotenexpansion in der A*-Suche zur Folge hat.

Obwohl Constraint Programming wie in den Abbildungen 5.8 und 5.9 ersichtlich wird in $63\ \%$ der Fälle und insbesondere bei allgemein sehr schnell lösbaren Instanzen eine langsamere Lösung findet als die A*-Suche, ist es aufgrund der geringeren Streuung und der schnelleren durchschnittlichen Lösungszeit sinnvoll, den Constraint-Programming-Solver zu bevorzugen. Anders verhielte es sich, wenn eine Möglichkeit bestünde, die Anzahl der von A* zu besuchenden Spielstände abzuschätzen. Ob eine solche existiert und wie diese aussehen könnte, wird im Rahmen dieser Arbeit allerdings nicht untersucht. Es liegt jedoch nahe, dass für die Lösung von Levels mit wenigen Fahrzeugen allgemein weniger Spielstände besucht werden müssen, da eine geringe Anzahl an Fahrzeugen eine geringere Anzahl an möglichen Zügen impliziert[26].

6.3.2 Rush-Hour-Instanzen bekannter Schwierigkeit - *Sample2*

Vielleicht scheint es nach Abschnitt 6.3.1 überraschend, dass Abbildung 5.10 deutlich macht, dass die A*-Suche auf *Sample2* im Vergleich zum Constraint Programming mit iterativer Erhöhung des Zuglimits effizienter ist. Bei näherer Analyse von Abbildung 5.10 und der Zusammenstellung von *Sample2* kann dies jedoch erklärt werden:

Die Sampledaten sind nach absteigender Anzahl an zu besuchenden Knoten sortiert; dies korreliert im Allgemeinen auch mit der Exploration von weniger Rush-Hour-Spielzuständen innerhalb der A*-Suche. Daher steigt die A*-Kurve zunächst langsam, bei den Puzzles mit nur wenigen Knoten jedoch beinahe sprunghaft an. Im direkten Gegensatz dazu zeigt der lineare Verlauf der orange gefärbten Kurve (Constraint Programming), dass die Anzahl der möglichen Knoten für CP-Solver keine Rolle spielt. Die Level werden grundsätzlich in beinahe konstanter Zeit gelöst, da sie dieselbe Anzahl an Zügen benötigen, weshalb sich die Anzahl an Variablen und Constraints nur geringfügig aufgrund der unterschiedlich vielen Fahrzeuge unterscheidet.

Da die Puzzle 26 Züge benötigen, kommt bei der iterativen Erhöhung des Zuglimits wie schon in Abschnitt 6.2.2 angesprochen zum Tragen, dass der Solver pro Puzzle drei mal ausgeführt wird, was die Ausführungsgeschwindigkeit verringert. Betrachtet man im direkten Vergleich den statischen Constraint-Programming-Solver in Abbildung 5.11, der als Zuglimit die tatsächliche Zuganzahl übergeben bekommt (was insofern unrealistisch ist, als dass diese im Vorhinein im klassischen Rush-Hour-Spiel nicht bekannt ist), dann fällt auf, dass sich an der grundlegenden Form der beiden Kurven nichts ändert. Die bisherigen Erkenntnisse bleiben also bestehen und Constraint Programming hat vor allem in den Instanzen von Rush Hour, die viele Knoten erfordern, einen klaren Wettbewerbsvorteil gegenüber der A*-Suche. Was sich jedoch verändert, ist die nach wie vor weitgehende konstante Geschwindigkeit des CP-Solvers. Diese hat sich, da der Solver nur noch ein mal ausgeführt werden muss, deutlich verringert, weshalb die orangefarbene Kurve (Constraint Programming) nun deutlich eher ihr Maximum erreicht als die blaue (A*).

Ein Blick auf die Tabelle 5.5 verdeutlicht dies: Der A*-Solver löst alle Instanzen nach $3284,27\ s$, wohingegen der statische Constraint-Programming-Solver nur $2530,31\ s$ benötigt und somit insgesamt knapp $30\ \%$ schneller ist. Überraschenderweise spiegelt sich die allgemein höhere Performanz des statischen Constraint Programmings nicht im AUC-Wert wider, was verdeutlicht, dass dieser alleine kein sinnvolles Maß für die Performanz ist und Ausführungsgeschwindigkeiten schwerpunktmäßig in den Blick genommen werden müssen. Der geringere AUC-Wert von A* lässt sich durch die verschiedenartigen Kurven erklären. Ein linearer Anstieg wie bei Constraint Programming erzeugt eine größere Fläche unter der Kurve. Die A*-Suche hingegen wächst exponentiell, was bedeutet, dass sie anfangs langsamer ist. Erst spät (bei den Puzzles mit weniger Knoten) steigt die Geschwindigkeit von A* stark an. Dadurch bleibt der AUC-Wert insgesamt geringer.

7 Fazit

Diese Arbeit hat sich mit einem systematischen Vergleich verschiedener Lösungsansätze für das Puzzle Rush Hour beschäftigt. Schwerpunktmäßig betrachtet wurden dabei die A*-Suche sowie Constraint Programming. Die Diskussion der Ergebnisse in Kapitel 6 hat gezeigt, dass die Performanzunterschiede stark davon abhängen, in welcher Ausgangssituation man sich befindet. Es lässt sich folgern, dass Constraint Programming der A*-Suche im Allgemeinen vorzuziehen ist: Da bei Spielen wie Rush Hour im Vorfeld unklar ist, wie schwierig das jeweilige Level zu lösen ist, sind die hohe Konstanz und geringe Streuung des CP-Solvers ein klarer Vorteil gegenüber der A*-Suche, deren Lösungsgeschwindigkeit direkt von der Anzahl der zu besuchenden Spielstände abhängt. Ist im Vorhinein bekannt, wie viele Züge die minimale Lösung für das Puzzle umfasst, ist für ein einziges Level immer noch Constraint Programming performanter, wenn das Zuglimit entsprechend angepasst wird. Besteht die Möglichkeit, das Zuglimit manuell anzupassen, hingegen nicht, liefert A* für Spiele mit einer hohen Anzahl an minimalen Zügen schnellere Ergebnisse. Diese Ideen wurden in den Abschnitten 6.2.2 und 6.3.2 dieser Arbeit eingehend diskutiert, bilden allerdings keine realistische Spielsituation ab.

Während es in der Literatur für beide hier diskutierten Lösungsverfahren bereits separate Untersuchungen im Hinblick auf ihre Performanz bei Rush Hour gibt[9, 14, 10], mangelte es bisher an einer Gegenüberstellung. Diese Arbeit hat eine solche in den wissenschaftlichen Diskurs rund um Rush Hour eingebracht. Gleichzeitig muss an dieser Stelle angemerkt werden, dass wesentlich performantere A*-Versionen existieren als in dieser Arbeit vorgestellt – beispielsweise durch die Nutzung von Bitboards –, sodass es sich für die zukünftige Forschung lohnen würde, die Ergebnisse dieser Arbeit mit einer performanteren Version von A* zu reproduzieren und eventuell zu validieren. Auch ein Blick auf die Speicherintensität von A* bietet sich an. Im Zuge dessen scheint es für die weitere Untersuchung des Themas ebenfalls sinnvoll, andere CP-Solver neben Z3 in den Blick zu nehmen, um diese miteinander zu vergleichen und auch hier die performanteste Lösung zu finden. Ein überdies interessanter Forschungsaspekt bestünde darin, herauszufinden, ob es eine Möglichkeit gibt, noch vor der Lösung eines Rush-Hour-Puzzles mit einem der Solver die Schwierigkeit desselben abzuschätzen, um individuell den besten Solver für die entsprechende Situation auszuwählen. So könnte in Fällen, in denen nur wenige mögliche Spielstände abgeschätzt werden, auf die in diesen Situationen signifikant schnellere A*-Lösung zurückgegriffen werden. Da hierfür jedoch ebenfalls Berechnungen erforderlich sind, die den Suchbaum betreffen und entsprechend Zeit in Anspruch nehmen, bedarf eine solche Idee weiterer Untersuchung.

Abschließend lässt sich festhalten, dass beide Lösungsansätze insgesamt gute Ergebnisse liefern. In realistischen Spielszenarien, in denen keine weiteren Informationen zur Schwierigkeit einer Instanz vorliegen, stellt Constraint Programming jedoch die bevorzugte Lösung dar.

Literaturverzeichnis

[1] Unblock me im app store. https://apps.apple.com/de/app/unblock-me/id315019111. Access: 21. november 2024.

[2] Move the block im app store. https://apps.apple.com/de/app/move-the-block-slide-puzzle/id994065259. Access: 21. november 2024.

[3] Mark Stamp, Brad Engel, McIntosh Ewell, and Victor Morrow. Rush hour® and dijkstra's algorithm. *Graph Theory Notes of New York*, 40:23–30, 2001.

[4] Michael Fogleman. Rush hour database. https://www.michaelfogleman.com/rush/#DatabaseDownload. Access: 21. november 2024.

[5] Michael Fogleman. Interesting rush hour puzzles. https://www.michaelfogleman.com/rush/#InterestingPuzzles. Access: 21. november 2024.

[6] Gary William Flake and Eric B. Baum. Rush hour is pspace-complete, or "why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1):895–911, 2002.

[7] Robert A. Hearn and Erik D. Demaine. Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1):72–96, 2005. Game Theory Meets Theoretical Computer Science.

[8] Yannick Schmid, Rolf Dornberger, and Thomas Hanne. Solving the rush hour puzzle problem by different heuristics. In Gerhard-Wilhelm Weber, Jose Francisco Martinez Trinidad, Michael Sheng, Raghavendra Ramachand, and Deepak Kharb, Latika ans Chahal, editors, *Information, Communication and Computing Technology*, pages 68–79. Springer Cham, 1. edition, 2024.

[9] Ami Hauptman, Achiya Elyasaf, Sipper Moshe, and Assaf Karmo. Gp-rush: Using genetic programming to evolve solvers for the rush hour puzzle. GECCO '09, pages 955–962, New York, NY, USA, 2009. Association for Computing Machinery.

[10] L. Cian, T. Dreossi, and Agostino Dovier. Modeling and solving the rush hour puzzle. In *CEUR Workshop Proceedings*, volume 3204, pages 294–306. CEUR-WS, 2022.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, USA, 4. edition, 2022.

[12] Robert Sedgewick and Kevin Wayne. *Algorithmen: Algorithmen und Datenstrukturen*. Pearson Deutschland, Hallbergmoos, Deutschland, 4. edition, 2014.

[13] Steven S. Skiena. *The Algorithm Design Manual*. Springer Cham, Basel, Schweiz, 3. edition, 2020.

[14] Tim J. Houthuijs. The a* search algorithm and its application to the game unblock me, 2020.

[15] Augusto Corrêa, André Pereira, and Marcus Ritt. Analyzing tie-breaking strategies for the a* algorithm. pages 4715–4721, Juli 2018.

[16] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Deutschland, 3. edition, 2016.

[17] Krzysztof Apt. *Introduction*, pages 1–7. Cambridge University Press, 2003.

[18] Barbara M. Smith. Modelling. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, pages 375–404. Elsevier Science, Amsterdam, Niederlande, 1. edition, 2006.

[19] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[20] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, Deutschland, 2008. Springer.

[21] Augusto B. Corrêa, André G. Pereira, and Marcus Ritt. Tie-breaking strategies for efficient a* search. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4715–4721. IJCAI, 2018.

[22] M. Asai and A. Fukunaga. Tiebreaking strategies for a* search: How to explore the final frontier. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI)*, 2016.

[23] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, pages 329–376. Elsevier, 2006.

[24] Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000)*, pages 599–603. IOS Press, 2000.

[25] Nikolaj Bjørner, Dung Phan Anh, and Lars Fleckenstein. vz - an optimizing smt solver. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015*, Lecture Notes in Computer Science, pages 194–199. Springer, 2015.

[26] J. Bockholt, D. Braun, and T. Dierkes. Ein netzwerkanalytischer Ansatz zur Untersuchung der Komplexität des Rush-Hour-Spiels. Bachelorarbeit, Universität Kaiserslautern, 2023.