

KI Projekt

Schiebepuzzle auf dem AKSEN-Board

Jonas Kappa

23. Februar 2021

Inhaltsverzeichnis

1. Einleitung	1
2. Systemumgebung	2
3. Algorithmen für Suchprobleme	3
3.1. Breitensuche	3
3.2. Tiefensuche	3
3.3. Iterative Tiefensuche	3
3.4. A*	4
4. Implementationen	5
4.1. Grundlagen und Voraussetzungen	5
4.2. Version 1 Tiefensuche mit ID	9
4.2.1. Beschreibung	9
4.2.2. Speicherverbrauch	12
4.2.3. Zeitmessung	13
4.3. Version 2 Tiefensuche mit ID	14
4.3.1. Beschreibung	14
4.3.2. Speicherverbrauch	17
4.3.3. Zeitmessung	19
4.4. A*	20
4.4.1. Beschreibung	20
4.4.2. Speicherverbrauch	24
4.4.3. Zeitmessung	26
5. Fazit	28
A. Listings	29
B. Verschiedenes	31
B.1. Ablauf der Programmausführung auf dem AKSEN-Board	31
B.2. Ablauf der Programmausführung auf dem PC (Windows)	31

Abbildungsverzeichnis

1.	Graphische Repräsentation des Zustandes $\{8, 1, 2, 0, 4, 3, 7, 6, 5\}$	5
2.	Baumdiagramm des Schiebepuzzles	7
3.	In grau, alle Knoten welche sich zeitgleich in der Agenda bei der Tiefensuche befinden	8
4.	Das Ergebnis der Durchführung von Version 1 des Tiefensuche-Algorithmus mit ID	12
5.	Suchbaum der Tiefensuche zur Erklärung der globalen Aktionsliste	15
6.	Puzzle mit angegebenen Indizes	16
7.	Das Ergebnis der Durchführung von Version 2 des Tiefensuche-Algorithmus mit ID	17
8.	Manhattan-Distanz von grün nach rot	20
9.	Linked List	21
10.	Linked List mit neuem Eintrag	22
11.	Das Ergebnis der Durchführung von A*	24

Listings

1.	Berechnung der Inversionen	10
2.	Äußere while -Schleife TS-ID Version 1	10
3.	Tiefensuche Version 1	11
4.	Innere while -Schleife der Version 2	14
5.	Eintrag wird aus der Agenda genommen Version 2 Tiefensuche mit ID	14
6.	Berechnung der Aktionsmöglichkeiten in Version 2 Tiefensuche mit ID	16
7.	Berechnung Heuristik	21
8.	Funktion calcChildrenAndAddToAgenda Version 1 Tiefensuche mit ID	29
9.	Das Hinzufügen eines neuen Agendaeintrages mit einer Linked List	30

Tabellenverzeichnis

1.	Zeitmessung der Version 1 Tiefensuche mit ID	13
2.	Zeitmessung der Version 2 Tiefensuche mit ID	19
3.	Zeitmessung A*	26

1. Einleitung

Das Schiebepuzzle ist ein bekanntes Spiel, bei welchem man von einer Ausgangsstellung durch Verschieben von Teilen zu einer Zielstellung gelangen möchte. In unserem Fall besteht das Puzzle aus 3x3 Feldern und besitzt somit acht Teile und ein freies Feld. Auf dieses freie Feld können die entsprechenden Teile horizontal und vertikal verschoben werden, solange bis die gewünschte Zielstellung eintritt. Dabei ist das neue freie Feld immer an der Position, welche das Teil innehatte, welches auf das freie Feld geschoben wurde. Man kann leicht erkennen, dass sich dieses Spiel gut als Beispiel für die Aktionsplanung anbietet. In jeder Stellung gibt es mindestens zwei und höchstens vier Möglichkeiten ein Teil zu verschieben. Wir können also von einer durchschnittlichen Verzweigungsrate von 3 ausgehen. Bei einem durchschnittlichen Lösungsweg der Länge 20, ergibt das eine Gesamtmenge von 3^{20} , also rund 3,5 Milliarden Knoten in der untersten Ebene des Suchbaumes.

Diese Arbeit beschäftigt sich damit, wie man dieses Suchproblem auf einem eingeschränkten System lösen kann und wie lange die gefundenen Lösungswege für ihre Bearbeitung brauchen. Der Speicherverbrauch fließt ebenfalls in die Betrachtung mit ein. Außerdem wird die Qualität der gefundenen Lösungen bewertet, bezüglich der gefundenen und der optimalen Lösungslänge.

2. Systemumgebung

Wir befinden uns auf dem AKSEN-Board. Dieses Board wurde an der TH-Brandenburg entwickelt und dient zum Einsatz in der Robotik und den Embedded Systems, sowohl in der Lehre als auch im professionellen Umfeld.[05] Für unser Projekt bedeutet dies, dass wir in einigen Punkten eingeschränkt werden. Die erste Einschränkung erfahren wir durch die Programmiersprache.

Ein Programm für das AKSEN-Board muss in C geschrieben sein und mit Hilfe des *sdcc* (Small Device C-Compiler) in eine *.ihx*-Datei übersetzt werden. Diese *.ihx*-Datei wird dann über eine serielle Schnittstelle auf das AKSEN-Board geflashed und kann dort ausgeführt werden. Das C-Programm muss eine Methode **void** *AksenMain(void)* beinhalten. Diese dient als Startpunkt des Programms, ähnlich zur *main* in C. Im Anhang B.1 und B.2 sind die Abläufe der Ausführung beschrieben.[05, S. 17]

Das AKSEN-Board beschränkt uns auch im Speicherplatz. Die Größe des Speicherplatzes für Programme beträgt 32KB. Unsere *.ihx*-Datei darf also nicht mehr Speicherplatz als die besagten 32 KB verbrauchen. Konsultieren wir das Handbuch, finden wir heraus, dass das geschriebene C-Programm im Durchschnitt doppelt so groß sein darf, da 1 Byte C-Code ungefähr 0,5 Byte *.ihx*-Code entspricht.[05, S. 47]

Weiterhin sind wir durch den Speicherplatz für globale Variablen eingeschränkt. Dieser beträgt 8 KB. Abgezogen werden hierbei:

- 512 Byte für die AKSEN-Bibliothek
- 100 Byte (maximal) für die globalen Variablen des *sdcc*
- 16 Byte für Bibliotheksparameter
- 256 Byte pro laufendem Prozess

$$\text{Verfügbarer globaler Speicher} = 8000 - 512 - 100 - 16 - \text{Prozessanzahl} \cdot 256$$

Da wir für unser Problem nur einen Prozess benötigen haben wir 7116 Byte Speicherplatz für globale Variablen zur Verfügung.[05, S. 46–47]

Wie wir gerade eben gesehen haben, wird für jeden Prozess 256 Byte Speicherplatz reserviert. Das ist der Stack, welcher für lokale Variablen, Parameter und Rücksprungadressen zur Verfügung steht. Nachdem das AKSEN-Board die Methode *AksenMain* aufgerufen hat stehen dieser noch 221 Byte zur Verfügung.[05, S. 46]

Fassen wir zusammen. Für Programme stehen uns 32 KB, für globale Variablen 7116 Byte und für lokale Variablen, Parameter und Rücksprungadressen 221 Byte zur Verfügung.

Da das AKSEN-Board nicht sehr leistungsstark im Vergleich zu einem Desktop-PC ist, sind wir auch im Bereich der Geschwindigkeit eingeschränkt. Dadurch werden wir angehalten nicht nur speichereffiziente, sondern auch sehr performante Algorithmen zu entwickeln.

3. Algorithmen für Suchprobleme

3.1. Breitensuche

Bei der Breitensuche handelt es sich um einen Suchalgorithmus, welcher den kürzesten Weg von einem Startknoten A zu einem Zielknoten Z in einem Graphen findet. Dabei wird so vorgegangen, dass erst alle Knoten einer Ebene abgearbeitet werden, bevor in die nächste Ebene vorgedrungen wird. Hierfür wird eine Warteschlange als Agenda benutzt. Das erste Element, also A, wird aus der Warteschlange genommen und es wird überprüft, ob es Z entspricht. Wenn dem so ist, ist der Algorithmus zu Ende. Falls nicht, wird der Knoten expandiert und alle expandierten Knoten werden der Warteschlange hinten angehangen. Der Algorithmus läuft solange bis der Zielknoten gefunden wurde oder sich keine Knoten mehr in der Agenda befinden.

Die Vorteile der Breitensuche liegen darin, dass der Algorithmus vollständig und, für Kanten mit gleichen Kosten, optimal ist.

Nachteilig ist der hohe Speicherplatzverbrauch in der Agenda. Da bei einer Breitensuche alle Knoten einer Ebene in der Agenda liegen haben wir einen Speicherplatzverbrauch von $\mathcal{O}(m^n)$ für $m = \text{Verzweigungsrate}$ und $n = \text{Suchtiefe}$.

3.2. Tiefensuche

Im Gegensatz zur Breitensuche arbeitet sich die Tiefensuche erst einen Pfad bis ganz nach unten durch. Erst dann werden abzweigende und zuletzt alternative Pfade durchsucht. Um dieses Verhalten zu erreichen wird ein Stapel als Agenda genutzt. Das erste Element A wird oben vom Stapel genommen und es wird überprüft, ob es Z entspricht. Ist dies der Fall, ist der Algorithmus zu Ende. Falls nicht, wird der Knoten expandiert und alle expandierten Knoten werden auf den Stapel oben drauf gelegt. Der Algorithmus läuft solange bis der Zielknoten gefunden wurde oder sich keine Knoten mehr in der Agenda befinden.

Die Vorteile der Tiefensuche liegen im Speicherplatzverbrauch der Agenda. Da immer nur ein Pfad nach unten gegangen wird ist auch der Speicherplatzverbrauch in der Agenda linear und beträgt $\mathcal{O}((m - 1)n + m - 1)$.

Nachteilig ist, dass die Tiefensuche nicht vollständig ist, wenn der Graph unendlich groß ist oder kein Test auf Zyklen durchgeführt wird. Dies kann dazu führen, dass Ergebnisse nicht gefunden werden, obwohl sie existieren. Leicht zu erkennen ist auch, dass die Tiefensuche nicht optimal ist. Das kommt dadurch, dass es in einem Suchbaum durchaus mehrere Wege zum Ziel führen können, welche aber unterschiedlich lang sind. Wenn dann der Pfad mit dem längeren Weg zuerst durchsucht wird, wird auch der Lösungsweg längerer Länge zuerst gefunden und als Ergebnis ausgegeben.

3.3. Iterative Tiefensuche

Die iterative Tiefensuche, oder auch Tiefensuche mit schrittweiser Vertiefung, ist die Kombination der Breiten- und Tiefensuche und verbindet die Vorteile der Breitensuche

mit den Vorteilen der Tiefensuche. Der grundlegende Suchalgorithmus funktioniert wie bei der Tiefensuche. D.h. es wird auch ein Stapel als Agenda genutzt. Jedoch ist die Tiefensuche nun durch eine maximale Suchtiefe begrenzt. Wenn die Suche bis zur maximalen Tiefe kein Ergebnis hervorgebracht hat, wird die maximale Tiefe um 1 erhöht und die Tiefensuche beginnt wieder von vorn.

Dadurch, dass die Tiefensuche in der maximalen Suchtiefe beschränkt wird, kann sie sich nicht in Zyklen und der Unendlichkeit verlieren und ist somit vollständig. Ebenso wie bei der Breitensuche ist auch die iterative Tiefensuche optimal, wenn die Kosten für alle Kanten gleich sind. Da der zugrunde liegende Algorithmus die Tiefensuche ist, ist der Speicherverbrauch der Agenda der selbe wie bei der Tiefensuche an sich.

Ein Nachteil, welcher bei der iterativen Tiefensuche jedoch aufkommt, ist die schlechtere Laufzeit. Dadurch, dass bei jeder weiteren Vertiefung der Suchbaum wieder neu aufgebaut werden muss, wird je nach Lösungstiefe der Baum entsprechend oft durchsucht.

3.4. A*

Im Gegensatz zur Breiten- und Tiefensuche hat A* keine wirkliche Suchrichtung. In A* wird eine sortierte Liste in der Agenda benutzt. Die Sortierung erfolgt anhand zweier Werte. Zum einen die Kosten $g()$, die es benötigt um zu dem entsprechenden Knoten zu gelangen. Zum anderen die Heuristik $h()$, welche besagt wie gut, bzw. wie nah am Zielknoten, ein Knoten ist. Der Wert für die Sortierung $f()$ eines Knotens berechnet sich also $f() = g() + h()$. Je kleiner $f()$, desto besser ist der Knoten und wird entsprechend weiter vorne in die Liste eingesortiert.

Wenn ein Knoten A vorne von der Liste entnommen wird, wird überprüft, ob dieser dem Zielknoten entspricht. Ist dies der Fall, ist der Algorithmus zu Ende. Wenn nicht, dann wird der Knoten expandiert. Für die expandierten Knoten wird $f()$ berechnet und die Knoten werden abhängig von ihrem Wert in die Agenda eingesortiert.

Der Erfolg dieses Algorithmus hängt stark davon ab, wie gut die Berechnung von $h()$ gelingt, bzw. wie gut die sich daraus ergebenden Werte sind. Falsche Werte können schnell zu falschen Fährten führen und den Algorithmus ineffizient machen. Gute Heuristiken können die Geschwindigkeit jedoch auch massiv erhöhen. Um dies zu gewährleisten darf die heuristische Funktion die tatsächlichen Kosten zum Ziel nie überschätzen. Wenn $h()$ diese Eigenschaft erfüllt, heißt sie *zulässig*.[He20, S. 44]

Der A*-Algorithmus ist vollständig, optimal und optimal effizient.[RN02] Optimal effizient bedeutet, dass es keinen anderen Algorithmus gibt, welche mit der gleichen Heuristik eine schnellere Lösung liefert.

Dadurch, dass sich A* wie eine Breitensuche verhalten kann, kann auch ein Speicherplatzverbrauch entstehen, welcher dem einer Breitensuche ähnelt.

8	1	2
	4	3
7	6	5

Abbildung 1: Graphische Repräsentation des Zustandes $\{8, 1, 2, 0, 4, 3, 7, 6, 5\}$

4. Implementationen

4.1. Grundlagen und Voraussetzungen

Bisher sind wir immer von einer Verzweigungsrate von 3 ausgegangen. Aber wie groß ist die Verzweigungsrate tatsächlich maximal und wie viele verschiedene Puzzlezustände gibt es eigentlich?

Fangen wir mit der Anzahl aller erreichbaren Puzzlezustände an. Wir wissen aus der Kombinatorik, dass es $9! = 362880$ Möglichkeiten gibt 9 unterschiedliche Teile anzurorden. Jedoch kann nur die Hälfte aller Zustände, nämlich $\frac{9!}{2} = 181440$, von einem beliebigen Startzustand erreicht werden. Das liegt an der symmetrischen Gruppe, in welcher sich jeder Zustand befindet. Dabei gibt es die Gruppe der geraden und der ungeraden Inversionen. Nur die Zustände in der geraden Gruppe können Zustände aus der geraden Gruppe erreichen und umgekehrt.

Wie berechnet man jedoch die Inversion eines Zustandes und warum kann man nur gerade Zustände von geraden Zuständen und ungerade von ungeraden Zuständen erreichen? Betrachten wir den Puzzlezustand aus Abbildung 1 als unseren Startzustand und schreiben ihn als Liste auf $\{8, 1, 2, 0, 4, 3, 7, 6, 5\}$. Um nun die Anzahl der Inversionen I zu berechnen gehen wir durch alle Teile (Zahlen) durch und vergleichen diese mit allen folgenden Zahlen. Für jede Zahl wird überprüft, ob sie kleiner oder größer ist als die folgenden. Für jede Zahl die sie größer ist erhöht sich die Anzahl der Inversionen um 1. Wichtig ist, dass das leere Feld (0) nicht beachtet wird! Betrachten wir die 4 in unserem Startzustand und vergleichen sie mit den folgenden Zahlen. Die 4 ist größer als die 3, jedoch kleiner als 7, 6 und 5. Somit erhöht sich die Anzahl der Inversionen um 1. Insgesamt ist die Anzahl der Inversionen für unseren Startzustand $I = 11$. Somit können nur Zustände erreicht werden, welche ebenfalls eine ungerade Anzahl an Inversionen haben.

Um zu verstehen, warum jeder Puzzlezustand nur in einen anderen Puzzlezustand seiner eigenen Gruppe überführt werden kann, schauen wir uns an was passiert, wenn ein Teil verschoben wird.

Wenn wir ein Teil horizontal, also nach links oder nach rechts auf das leere Feld verschieben, ändert sich nichts an der Anzahl der Inversionen. Das liegt daran, dass das leere Feld für die Inversionen ignoriert wird. Etwas komplexer sieht es mit vertikalen Verschiebungen aus. Wichtig zu erkennen ist, dass, wenn wir ein Teil nach unten oder oben verschieben, es *immer* zwei Plätze nach vorn (bei Verschiebung nach oben) oder zwei Plätze nach hinten (bei Verschiebung nach unten) springt. Somit kann die Anzahl

der Inversionen sich bei einer Verschiebung nach unten nur auf drei verschiedene Arten und bei einer Verschiebung nach oben auch nur auf drei verschiedene Arten ändern. Wobei die zweite Art bei beiden Verschiebungen die gleiche ist und die erste und dritte Art der beiden lediglich vertauscht. Deshalb reicht es aus lediglich die Verschiebung nach oben (ein Feld nach oben auf das leere Feld schieben) zu betrachten.

Fall 1: +2 Die Anzahl der Inversionen erhöht sich um +2 bei einer Verschiebung nach oben genau dann, wenn die beiden Zahlen, welche übersprungen werden, kleiner sind, als die Zahl des verschobenen Feldes. Für unseren Initialzustand $\{8, 1, 2, 0, 4, 3, 7, 6, 5\}$ sieht das folgendermaßen aus. Nur die 7 kann in unserem Startzustand nach oben auf das leere Feld geschoben werden. Somit erhalten wir $\{8, 1, 2, 7, 4, 3, 0, 6, 5\}$ als neuen Zustand. Wie wir sehen sind die beiden übersprungenen Teile (4 und 3) kleiner als 7 und somit liegt die Anzahl der Inversionen bei $I = 13$. Wenn eine Verschiebung nach unten geschieht müssen die beiden übersprungenen Zahlen größer sein, als die verschobene Zahl.

Fall 3: -2 Aus didaktischen Gründen ist es einfacher den dritten Fall vor dem zweiten Fall zu betrachten. Der dritte Fall ist genau umgekehrt zum Ersten. Wenn wir ein Teil nach oben verschieben und beide übersprungenen Zahlen größer sind als die Verschobene, verringert sich die Anzahl der Inversionen um 2. Für die Verschiebung nach unten müssen die beiden übersprungenen Zahlen kleiner sein.

Fall 2: +0 Was passiert, wenn eine der Übersprungenen Zahlen größer und eine kleiner ist, als die verschobene Zahl? Dann bleibt die Anzahl der Inversionen gleich. Bei einer Verschiebung nach oben sorgt eine größere Zahl für -1 (siehe Fall 3) und eine kleinere Zahl für +1 (siehe Fall 1). Diese gleichen sich aus und ergeben 0. Das gilt umgekehrt auch für die Verschiebung nach unten.

Wir sehen, dass für jede mögliche Verschiebung die Parität der neu entstanden Zustände die gleiche ist, wie die des Ausgangszustandes.[Ge19][Kö00]

Nun haben wir geklärt wie groß der Zustandsraum ist, in welchem wir uns bewegen. Bleibt noch die Frage offen wie groß die Verzweigungsrate tatsächlich ist. Das leere Feld kann an drei unterschiedlichen Positionen sein. Es kann in der Mitte, am Rand und in einer Ecke sein. Wenn es sich in der Mitte befindet gibt es 4 Zugmöglichkeiten. Am Rand gibt es 3 Möglichkeiten und in den Ecken stehen noch 2 Möglichkeiten zur Verfügung. Jedoch sind nicht alle der Möglichkeiten sinnvoll. Wenn wir uns nicht im ersten Zug, sondern in einem beliebigen anderen Zug befinden, ist die Anzahl der Möglichkeiten um 1 verringert, da es nicht sinnvoll ist, den gerade getätigten Zug wieder rückgängig zu machen. Damit ergeben sich 3 Züge für die Mitte, 2 für den Rand und 1 für die Ecke.

Nehmen wir an, dass sich das leere Feld in unserem Puzzle in der Mitte befindet. Somit haben wir für den ersten Zug 4 Möglichkeiten für neue Zustände. Den entsprechenden Baum dazu kann man in Abbildung 2 (S. 7) betrachten. Alle Puzzlezustände, welche sich in Tiefe 1 befinden haben das leere Feld in einer Kantenposition. Das heißt, dass es

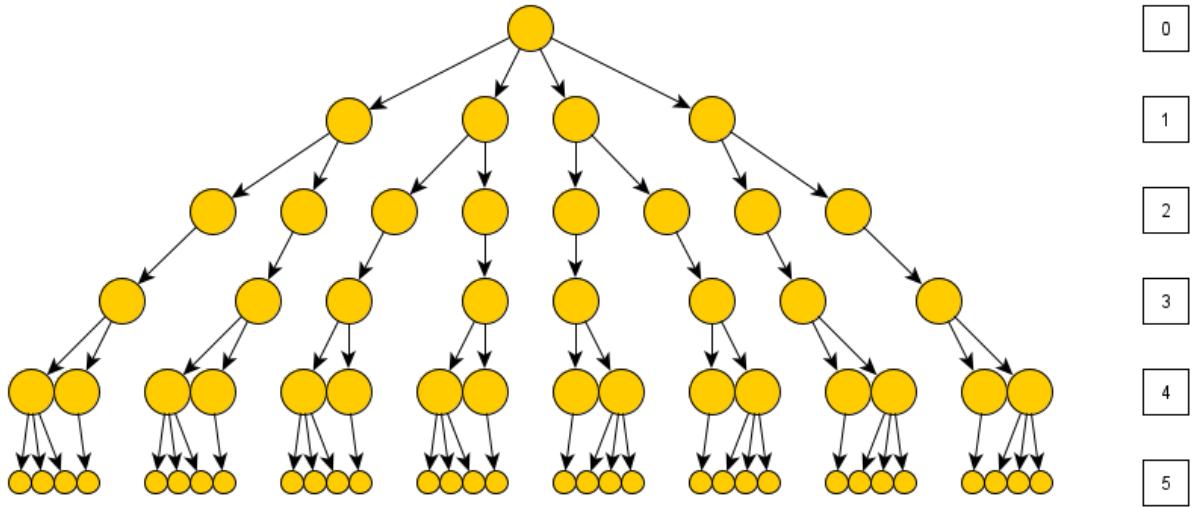


Abbildung 2: Baumdiagramm des Schiebepuzzles

für alle 4 Knoten nur 2 Möglichkeiten der Expansion gibt. Die Knoten, welche in Tiefe 2 liegen, haben ihr leeres Feld in einem Eckfeld. Damit haben sie nur eine Möglichkeit zu expandieren, womit in Tiefe 3 alle Knoten wieder ihr leeres Feld am Rand haben. Wie schon in Tiefe 1 gibt es für jeden Knoten wieder 2 Möglichkeiten. Diese 2 Möglichkeiten führen dazu, dass in Tiefe 4 die Hälfte aller Knoten ihr leeres Feld in der Mitte haben und die andere Hälfte wieder in der Ecke sind. In Tiefe 5 kann man sehen, dass die eine Hälfte um 3 und die andere um 1 erweitert wird.

Wir können in der Abbildung sehen, dass unsere Verzweigungsrate im Durchschnitt die 2 nie übersteigen wird. Also können wir auch was unsere Anzahl der Knoten in der Agenda angeht von einem Binärbaum ausgehen. Es darf jedoch nicht die erste Verzweigung vergessen werden. Diese ist für die Anzahl der Knoten in der Agenda ebenfalls wichtig.

Kommen wir vom theoretischen Part zu etwas mehr Praxis orientierten. Um unseren Algorithmus entwerfen zu können, müssen wir ein paar Festlegungen treffen, wie wir unsere Daten speichern. Ein Puzzle wird als Liste aus **char** repräsentiert. Jedes Feld bekommt einen Buchstaben *a* bis *h* zugewiesen und das leere Feld wird zum *o*. Für unser Beispiel aus Abb. 1 würde das `{h, a, b, o, d, c, g, f, e}` ergeben. Die **char**-Repräsentation wurde gewählt, da ein **char** lediglich ein Byte Speicher verbraucht. Ebenso werden die Aktionen durch den Datentyp **char** kodiert. Das *u* steht dabei für *up*, *d* für *down*, *l* für *left* und *r* für *right*. Dabei stellt die Richtung die Bewegungsrichtung des leeren Feldes dar und nicht die Bewegung eines Feldes auf das leere Feld.

In Kapitel 3 haben wir uns die unterschiedlichen Algorithmen für Suchprobleme angesehen. Nun müssen wir uns entscheiden, welche Algorithmen für uns in Frage kommen. Wie wir leicht feststellen, ist die Breitensuche nicht für uns geeignet, da sie viel zu schnell die Agenda mit Knoten füllt. Für diese große Menge an Knoten haben wir jedoch nicht genug Speicher auf dem AKSEN-Board zur Verfügung. Bereits in Tiefe 13 bräuchten wir $2^{13} = 8192$ Plätze in der Agenda. Selbst wenn wir die Daten, welche so einen Knoten

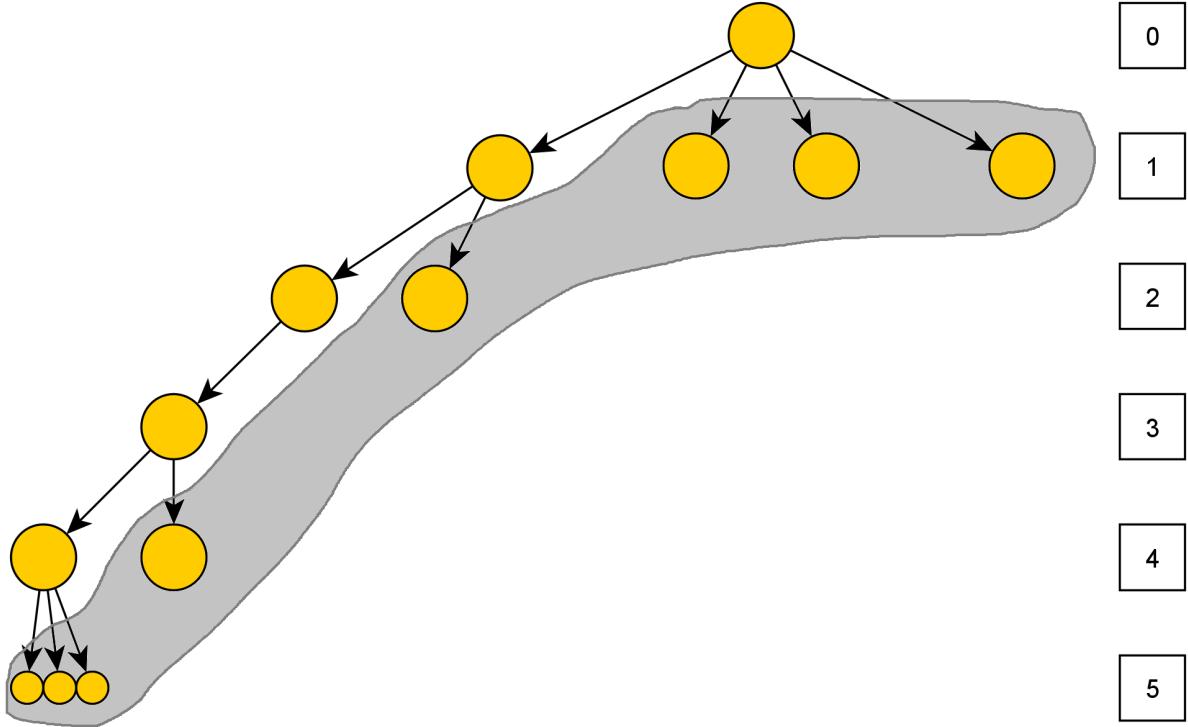


Abbildung 3: In grau, alle Knoten welche sich zeitgleich in der Agenda bei der Tiefensuche befinden

in der Agenda ausmachen, auf ein Byte komprimieren könnten, würde der Platz nicht ausreichen.

Wenden wir uns also zur Tiefensuche, welche mit wesentlich weniger Plätzen in der Agenda auskommt. Wie in 3.2 bereits beschrieben ist der Speicherplatzverbrauch linear und beträgt für unser Beispiel $\mathcal{O}((2 - 1)n + 2 - 1 + 2) = \mathcal{O}(n + 3)$ für $m = 2$. Die +2 am Ende kommt daher, dass im Worst-Case die erste Verzweigung aus 4 Knoten bestehen kann, wie man in Abb. 3 sieht. Dies führt dazu, dass anstatt des einen zusätzlichen Knotens in Tiefe 1, den man beim Binärbaum eigentlich hat, noch zwei weitere Knoten hinzukommen. Ebenfalls zu sehen ist, dass die Anzahl der Knoten in der Agenda nie $\mathcal{O}(n + 3)$ übersteigt. Bei gleicher Tiefe wie bei der Breitensuche brauchen wir $13 + 3 = 16$ Plätze in der Agenda. Das ist eine sehr starke Verbesserung und gibt uns die Möglichkeit die Daten, welche wir in einem Knoten speichern wollen, perfekt an unsere Bedürfnisse anzupassen. Jedoch hat die Tiefensuche auch einen Nachteil. Da unser Suchbaum erst während des Algorithmus aufgebaut wird und Zyklen möglich sind, wird die Tiefensuche im ersten Ast in die Unendlichkeit laufen. Um das zu verhindern gibt es zwei Möglichkeiten.

Möglichkeit 1 ist zu überprüfen, ob wir in einem Zustand schon einmal waren und wenn ja, diesen nicht zu expandieren. Dieses Verhalten kann auf zwei Arten erreicht werden. Man kann alle bereits besuchten Zustände speichern und die neu expandierten damit vergleichen. Das wäre jedoch wieder zu specheraufwendig und im späteren

Verlauf des Algorithmus zu viel Rechenaufwand. Die zweite Art wäre aus den aktuell gemerkten Aktionen die unterschiedlichen Puzzlezustände jedes mal neu zu berechnen und zu vergleichen. Der Speicheraufwand ist hierbei minimal, jedoch ist der Rechenaufwand noch viel höher. Jedoch gibt es ein noch viel größeres Problem. Selbst wenn wir mit dem Speicher und dem Rechenaufwand kein Problem hätten, würden wir vermutlich keine optimale Lösung bekommen. Der Grund darin liegt, dass man bei einem Schiebe-puzzle auf verschiedenen Wegen zum Ziel kommen kann, frei nach dem Motto „alle Wege führen nach Rom“. Vermutlich würde der Algorithmus so lange nach unten den ersten Ast entlang laufen, bis er den Zielzustand gefunden hat.

Genau hier kommt Möglichkeit 2 ins Spiel mit der iterativen Tiefensuche. Um zu verhindern, dass die Tiefensuche ins unendliche läuft wird eine Tiefenbegrenzung eingeführt. Diese wird denn schrittweise erhöht und somit kann die Tiefensuche immer weiter im Suchbaum vordringen. Damit ist die iterative Tiefensuche ein guter Kandidat um als Prototyp umgesetzt zu werden.

Aber auch A* kann sich als Algorithmus für unser Problem eignen. Auf diesen wird jedoch im Kapitel 4.4 nochmal näher eingegangen.

4.2. Version 1 Tiefensuche mit ID

4.2.1. Beschreibung

Die Idee für die erste Version ist jeden Knoten in der Agenda als eine Aktionsliste darzustellen. Also welche Aktionen zu dem aktuellen Puzzlezustand geführt haben. Anhand dieser kann der aktuelle Puzzlezustand berechnet werden. Dieser wird dann mit dem Finalzustand verglichen und aus diesem werden neue Zustände expandiert. Auf einen Vergleich um Zyklen auszuschließen wird verzichtet. Was jedoch, wie bereits erwähnt, bei allen Algorithmen gemacht wird, ist dass Aktionen verhindert werden, welche die zuvor getätigte Aktion wieder rückgängig machen. D.h. auf *up* darf nicht *down*, auf *left* nicht *right* und umgekehrt folgen. Dieses Verhalten wird dadurch erreicht, dass in jedem Knoten der letzte Eintrag die zuletzt getätigte Aktion ist.

Sehen wir uns den Algorithmus etwas genauer an. Zuerst wird überprüft ob eine Lösung überhaupt möglich ist. Dies erreichen wir über die Anzahl der Inversionen, deren Berechnung im Listing 1 S. 10 zu sehen ist. Wir gehen jedes Feld vom `initialPuzzle` und vom `finalPuzzle` durch und überprüfen zuerst ob es das leere Feld ist (Z. 5 u. 12). Ist es nicht das leere Feld vergleichen wir das aktuelle Feld mit allen Feldern danach und addieren pro gefundenem Buchstaben, welcher kleiner als der aktuelle Buchstabe ist, 1 zu `inversionNumberInitial`, bzw. `inversionNumberFinal` hinzu. Sowohl `inversionNumberInitial`, als auch `inversionNumberFinal` sind globale Variablen um Speicherplatz auf dem Stack zu sparen. In Zeile 20 werden dann die beiden Variablen modulo 2 gerechnet, das Ergebnis miteinander verglichen und in der globalen Variable `solutionIsPossible` gespeichert. Wenn eine Lösung möglich ist kann der Algorithmus weiterlaufen, ansonsten wird er gestoppt.

Nun startet der Algorithmus mit einer `while`-Schleife. Die Kondition ist in Listing 2 beschrieben. Solange die Lösung nicht gefunden wurde und die lokale maximale Tiefe

```

1 void checkIfSolutionIsPossible() {
2     unsigned char i;
3     unsigned char j;
4     for (i = 0; i < PUZZLE_LENGTH; i++) {
5         if (initialPuzzle[i] != 'o') {
6             for (j = i + 1; j < PUZZLE_LENGTH; j++) {
7                 if (initialPuzzle[i] > initialPuzzle[j]) {
8                     inversionNumberInitial += 1;
9                 }
10            }
11        }
12        if (finalPuzzle[i] != 'o') {
13            for (j = i + 1; j < PUZZLE_LENGTH; j++) {
14                if (finalPuzzle[i] > finalPuzzle[j]) {
15                    inversionNumberFinal += 1;
16                }
17            }
18        }
19    }
20    solutionIsPossible = inversionNumberInitial % 2 ==
21        inversionNumberFinal % 2;
}

```

Listing 1: Berechnung der Inversionen

`depth` kleiner als `MAX_DEPTH` ist, soll die Tiefensuche erneut mit neuer lokaler maximaler Tiefe durchgeführt werden.

Konkret bedeutet das, wenn wir eine `MAX_DEPTH` = 10, einen `DEPTH_ITERATOR` = 1, die aktuelle lokale maximale Tiefe `depth` = 0 (wie sie initial angelegt wird) haben und eine Lösung noch nicht gefunden wurde, dass die Kondition `true` ergibt und die Tiefensuche mit neuer lokaler maximaler Tiefe `depth` = `depth` + `DEPTH_ITERATOR` durchgeführt wird. Wenn die Tiefensuche mit `depth` = 1 komplett durchlaufen wurde und kein Ergebnis gefunden wurde, wird die Tiefensuche erneut mit `depth` = `depth` + `DEPTH_ITERATOR` durchgeführt. Sollte `depth` = `depth` + `DEPTH_ITERATOR` größer als `MAX_DEPTH` sein, wird `depth` auf `MAX_DEPTH` gesetzt. Sobald die lokale maximale Tiefe gleich oder größer `MAX_DEPTH` ist oder eine Lösung gefunden wurde, ist der Algorithmus

```

1 while (!found && depth < MAX_DEPTH) {
2     init(); // Funktion zum initialisieren der Agenda
3     depth = depth + DEPTH_ITERATOR;
4     if (depth > MAX_DEPTH) {
5         depth = MAX_DEPTH;
6     }
7     // Tiefensuche-Algorithmus
8 }

```

Listing 2: Äußere `while`-Schleife TS-ID Version 1

```

1  while (agendaPointer >= 0 && !agendaPointerOverflow) {
2      fillCurrentActionList(); // fuellt die globale Variable mit der
        aktuellen Aktionsliste
3      calcAndSetCurrentPuzzle(); // berechnet den aktuellen Puzzlezustand
        aus der Aktionsliste
4      clearCurrentAgenda(); // loescht den Agendaeintrag aus der Agenda
5      if (!checkCurrentIsFinal()) { // ueberprueft, ob der aktuelle
        Puzzlezustand der Finalzustand ist
6          currentDepth = getLengthOfActionList();
7          if (currentDepth < depth) {
8              agendaPointerOverflow = !calcChildrenAndAddToAgenda(); // 
                expandieren des Knotens und Ueberpruefung ob die Agenda
                ueberlaeuft
9          }
10     } else { // Loesung wurde gefunden
11         for (i = 0; i < MAX_DEPTH; i++) {
12             solutionActionList[i] = currentActionList[i];
13         }
14         agendaPointer = -1; // um aus der while-Schleife auszubrechen
15         found = true;
16     }
17 }

```

Listing 3: Tiefensuche Version 1

beendet.

Kommen wir zum Algorithmus der Tiefensuche, welcher in Listing 3 beschrieben ist. Zuerst wird der Knoten, welcher auf dem Stack ganz oben liegt aus der Agenda genommen und sein Puzzlezustand berechnet (Z. 2-4). Dann wird überprüft, ob dieser Puzzlezustand der finale Zustand ist. Sollte dies der Fall sein wird die aktuelle Aktionsliste in die Lösungsliste kopiert. Zusätzlich wird der `agendaPointer` = -1 gesetzt um aus der `while`-Schleife auszubrechen. Auch das globale Flag `found` wird auf `true` gesetzt um aus der äußeren `while`-Schleife auszubrechen (Z. 11-15).

Wenn der aktuelle Zustand nicht der finale Zustand ist, soll der aktuelle Zustand erweitert werden (Z. 6-9). Hierbei gibt die Funktion `calcChildrenAndAddToAgenda` einen `bool`-Wert zurück, welcher angibt ob die Agenda übergelaufen ist (`false`, Ausführung war nicht erfolgreich) oder nicht (`true`, Ausführung war erfolgreich).

Die Funktion `calcChildrenAndAddToAgenda` ist im Anhang A im Listing 8 (S. 29) beschrieben. Zuerst wird die xy-Position des leeren Feldes berechnet um die Verschiebungen besser berechnen zu können. Dieser Vorgang fällt in den anderen Algorithmen weg, da er etwas aufwendiger und nicht wirklich nötig ist. Abhängig davon wo sich das leere Feld befindet und welche Aktion als letztes getätigter wurde, werden die neuen Puzzle mit den Aktionen *r*, *l*, *d* und *u* erstellt und der Agenda hinzugefügt.

Wenn der `agendaPointer` nicht übergelaufen ist und auch nicht kleiner als 0 ist, wird der nächste Knoten von der Agenda genommen und der Algorithmus beginnt von vorn.

Wenn das Beispiel, im Ordner `Version 1 Tiefensuche mit ID`, auf dem AKSEN-Board durchgeführt wird erhalten wir das Ergebnis, welches in Abb. 4 S.12 dargestellt

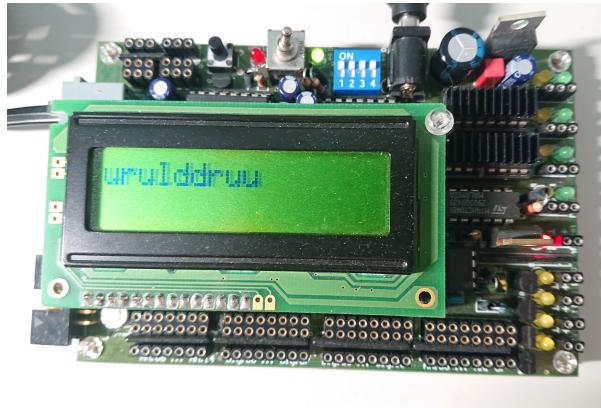


Abbildung 4: Das Ergebnis der Durchführung von Version 1 des Tiefensucheralgorithmus mit ID

ist. Zu sehen ist die Aktionsliste welche auf den Initialzustand $\{g, a, e, f, d, b, o, c, h\}$ angewendet werden muss um zum Finalzustand $\{d, o, e, f, g, b, c, a, h\}$ zu gelangen.

4.2.2. Speicherverbrauch

Für den globalen Speicher mit einer maximalen Suchtiefe von 30:

- 4 mal `int` = 8 Byte
- 3 mal `bool` = 3 Byte
- 1 mal Agenda bestehend aus `char` = $33 \cdot 30 = 990$ Byte
- 3 Puzzle bestehend aus jeweils 9 `char` = 27 Byte
- 2 mal Aktionsliste aus jeweils 30 `char` = 60 Byte
- Gesamt = 1088 Byte

Daraus können wir folgende Funktion zur Berechnung des globalen Speichers erstellen:

$$f(x) = x^2 + 5x + 38, \text{ für } x = \text{maximale Suchtiefe}$$

Wenn wir für $f(x)$ nun unsere maximal verfügbaren 7116 Byte im globalen Speicher einsetzen, erhalten wir rund 81,6 für x . Also kann dieser Algorithmus spechertechnisch bis maximal Tiefe 81 suchen.

Der maximale Speicherverbrauch das Stacks kann auf dem PC mit dem zusätzlichen Flag `MEM` berechnet werden. Die Berechnung ergibt dann, dass auf dem Stack maximal 87 Byte gleichzeitig durch das Programm abgelegt werden.

Die generierte .ihx-Datei ist 15620 Byte groß und passt somit auch auf den Programmspeicher.

Wir können sehen, dass der Speicherplatz des AKSEN-Board ausreicht um unseren Algorithmus ausführen zu können.

4.2.3. Zeitmessung

Lösungslänge	Schrittweite	Zeit in ms	Gefundene Lösungslänge
8	1	27691	8
	2	14903	8
	3	19607	8
	5	28778	8
	10	26171	8
9	1	44348	9
	2	36811	9
	3	11705	9
	5	13896	9
	10	11288	9
10	1	126567	10
	2	65945	10
	3	156591	10
	5	36321	10
	10	32813	10
11	1	209477	11
	2	201049	11
	3	149167	11
	5	716196	11
12	1	346442	12
	2	190270	12
	3	141265	12
	5	653257	12
	10	339585	12
	15	588897	12

Tabelle 1: Zeitmessung der Version 1 Tiefensuche mit ID

Die Tabelle 1 zeigt uns in der ersten Spalte die optimale Lösungslänge von einem Initialzustand zu einem Finalzustand. Diese beiden Zustände bekommt der Algorithmus als Eingabe, für welche er die optimale Lösung finden soll. In der zweiten Spalte ist angegeben welchen **DEPTH_ITERATOR** er dafür nutzt. In der dritten Spalte ist angegeben wie lange er dafür gebraucht hat. Da das AKSEN-Board kein Betriebssystem besitzt müssen die Algorithmen nicht mehrfach ausgeführt werden, da die Zeiten immer gleich bleiben. Die letzte Spalte gibt an, welche Länge vom Algorithmus tatsächlich gefunden wurde.

Man kann sehen, dass der Algorithmus ab einer Lösungslänge von 10 sehr langsam wird. Um noch die optimalen Zeiten zu erreichen, müsste die Schrittgröße jedes mal per Hand an die optimale Lösung angepasst werden. Optimalerweise auf exakt die Lösungslänge.

Wir müssen also einen Algorithmus finden, welcher effizienter arbeitet.

```
1 while (agendaPointer >= 0 && !agendaPointerOverflow && !found)
```

Listing 4: Innere **while**-Schleife der Version 2

```
1 if (agenda[agendaPointer][DEPTH_INDEX] > 0) {  
2     actionList[(agenda[agendaPointer][DEPTH_INDEX] - 1)] = agenda[  
         agendaPointer][ACTION_INDEX];  
3 }
```

Listing 5: Eintrag wird aus der Agenda genommen Version 2 Tiefensuche mit ID

4.3. Version 2 Tiefensuche mit ID

4.3.1. Beschreibung

Die Grundidee der zweiten Variante ist eine globale Aktionsliste. Anstatt, dass die komplette Aktionsliste für jeden Puzzlezustand in der Agenda abgelegt werden, liegt jetzt nur noch die letzte Aktion in der Agenda. Das alleine reicht jedoch nicht aus. Zusätzlich benötigen wir noch die aktuelle Tiefe für die entsprechende Aktion, um diese zuordnen zu können.

Da der Speicherverbrauch unserer Agenda jetzt wesentlich geringer geworden und nicht mehr abhängig von der maximalen Suchtiefe ist, können wir auch noch etwas mehr Informationen in die Agenda legen. Zum einen legen wir den jeweils dazugehörigen Puzzlezustand in der Agenda mit ab. Dadurch brauchen wir diesen nicht jedes mal neu zu berechnen und sparen uns somit viel Rechenkapazität ein. Zum anderen legen wir auch noch die Position des leeren Feldes mit in die Agenda. Da diese für die Erstellung des neuen Puzzlezustandes schon berechnet werden muss, sparen wir uns den Aufwand der späteren erneuten Berechnung mit ein.

Um zu verstehen wie der Algorithmus mit der globalen Aktionsliste funktioniert, schauen wir uns diesen am besten in seiner Gesamtheit einmal an.

Der Anfang ist, wie in Version 1, der gleiche. Zuerst wird überprüft, ob eine Lösung möglich ist (Listing 1 S. 10). Dann wird wieder die äußere **while**-Schleife gestartet, welche die schrittweise Vertiefung implementiert (Listing 2 S. 10). In der inneren **while**-Schleife gibt es jedoch eine kleine Änderung, welche im Listing 4 dargestellt ist. Während wir in der ersten Version noch durch einen `agendaPointer < 0` vorzeitig aus der inneren **while**-Schleife ausbrechen konnten, ist das hier nun nicht mehr möglich. Das liegt daran, dass wir auch nachdem der Algorithmus beendet ist noch wissen müssen an welcher Stelle in der Agenda sich die Lösung befindet, da dort die Tiefe drinnen steht bis zu welcher die globale Aktionsliste ausgelesen werden muss. Deshalb wird zusätzlich abgefragt, ob eine Lösung bereits gefunden wurde.

Nun kommen wir zum wichtigen Teil. Ein Eintrag wird von der Agenda genommen (Listing 5 S. 14). Dabei wird wieder der Eintrag aus der Agenda gewählt, auf welchen der `agendaPointer` zeigt. Es wird überprüft ob die Tiefe größer ist als 0. Ist dies der Fall gibt es eine Aktion, welche in die Aktionsliste geschrieben werden kann. Der Index der Position, an welche die Aktion in der Aktionsliste geschrieben wird, ist dabei die Tiefe

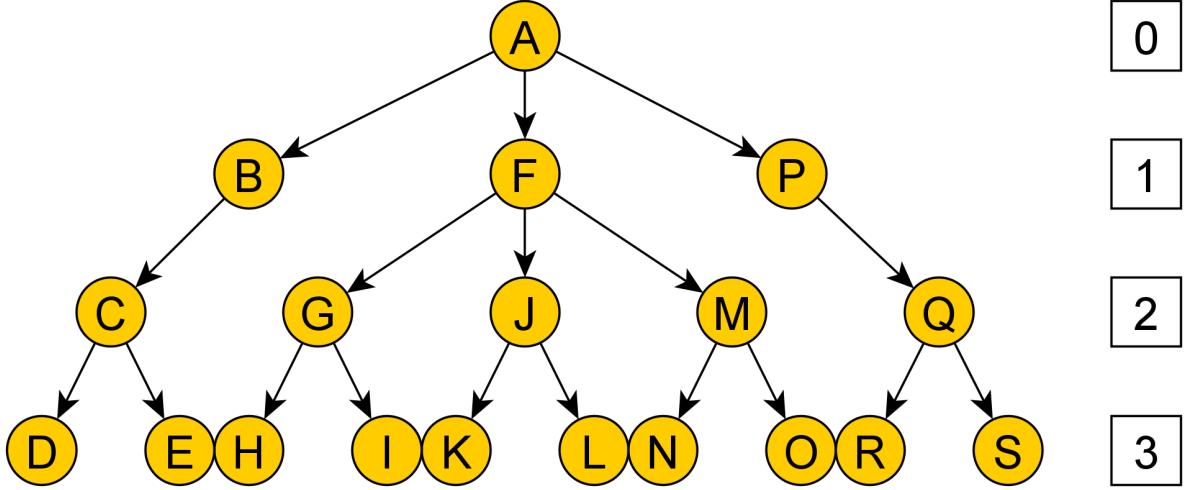


Abbildung 5: Suchbaum der Tiefensuche zur Erklärung der globalen Aktionsliste

der Aktion minus 1. Dadurch, dass in der Tiefensuche mit Backtracking gearbeitet wird, ist die Aktionsliste bis zur aktuellen Tiefe immer korrekt, da nur nach oben gesprungen und nie nach unten gesprungen wird.

Schauen wir uns dafür die Grafik in Abb. 5 an. Diese stellt einen Suchbaum dar, welcher so durch unseren Tiefensuchealgorithmus entstehen könnte. Das leere Feld beim Startknoten A ist ein Kantenfeld und kann somit in 3 Richtungen expandieren. Von den 3 expandierten Feldern sind zwei der leeren Felder Eckfelder und eins ist das mittlere Feld. Die Eckfelder haben nur eine Aktionsmöglichkeit zur Verfügung, das mittlere besitzt 3 Möglichkeiten. Jedes der expandierten Felder in Tiefe 2 ist nun ein Kantenfeld und hat somit genau 2 Aktionsmöglichkeiten.

A ist der erste Knoten welcher in der Agenda liegt und expandiert wird. Dabei würden die Knoten P, F und B auf die Agenda gelegt werden, in der Reihenfolge. Dann würde B expandiert werden und C würde auf den Stapel gelegt werden. Da C nun ganz oben liegt wird es expandiert und E und D werden auf den Stapel gelegt. Im nächsten Schritt liegt D ganz oben, da aber unsere maximale Tiefe von 3 erreicht ist, wird es nicht weiter expandiert.

In jedem Schritt wurde die Aktionsliste mit den jeweiligen Aktionen aus B, C und D modifiziert. Dabei sind wir immer um jeweils 1 weiter in die Tiefe vorgedrungen. Nun liegt E ganz oben auf der Agenda. Auch E ist ganz unten und wird nicht weiter expandiert. Jedoch hat E die Aktion von D in der Aktionsliste ersetzt. Die Aktionen, welche zum Puzzlezustand D geführt haben sind nun nicht mehr nachvollziehbar. Das ist jedoch kein Problem, da D nicht der Finalzustand war, ansonsten wäre der Algorithmus stehen geblieben. Die globale Aktionsliste hat sich von $\{B, C, D\}$ auf $\{B, C, E\}$ geändert.

Der nächste Knoten in der Agenda ist das F. Nun springen wir zum ersten mal in der Tiefe nach oben auf Tiefe 1. Das bedeutet, dass der Inhalt unserer Aktionsliste $\{F, C, E\}$ ist, da die Aktion von F an der Stelle $Tiefe(F)-1$ abgespeichert wurde. Die Aktionen von C und E ergeben nun keinen Sinn mehr. Wir wissen weder, zu welchem Puzzlezustand

```

1 if (pBlank % 3 != 2 && pAction != 'l') {
2     // Aktion right ist moeglich
3 }
4 if (pBlank % 3 != 0 && pAction != 'r') {
5     // Aktion left ist moeglich
6 }
7 if (pBlank <= 5 && pAction != 'u') {
8     // Aktion down ist moeglich
9 }
10 if (pBlank >= 3 && pAction != 'd') {
11    // Aktion up ist moeglich
12 }

```

Listing 6: Berechnung der Aktionsmöglichkeiten in Version 2 Tiefensuche mit ID

0	1	2
3	4	5
6	7	8

Abbildung 6: Puzzle mit angegebenen Indizes

$\{F, C, E\}$ führt, noch ob diese Kombination an Aktionen überhaupt gültig ist. Aber auch das ist kein Problem, denn die Aktionsliste ist immer nur bis zur aktuellen Tiefe des aktuellen Knotens gültig. In diesem Fall ist der aktuelle Knoten F und somit ist die Aktionsliste nur bis Index $Tiefe(F) - 1$, also 0 gültig.

Man kann die Grafik nach der alphabetischen Bezeichnung weiter durchgehen um sich den Vorgang noch besser verdeutlichen zu können.

Wenn Q unseren Finalzustand darstellt, würde die Aktionsliste, am Ende des Algorithmus, $\{P, Q, O\}$ sein und bis Index 1 ausgelesen werden.

Die Aktion des obersten Eintrags in der Agenda wurde nun in die globale Aktionsliste geschrieben. Der Algorithmus überprüft jetzt, ob es sich um den Finalzustand handelt oder nicht. Ist dies der Fall stoppt der Algorithmus und die globale Aktionsliste kann bis zur $Tiefe(Knoten) - 1$ des aktuellen Knotens ausgelesen werden. Sollte es jedoch nicht der Finalzustand sein, wird der Puzzlezustand gegebenenfalls erweitert.

Die zweite Variante hat ein Upgrade gegenüber der ersten bekommen, was die Berechnung der möglichen Aktionen angeht. Während bei Version 1 aus dem Index des leeren Feldes xy-Koordinaten berechnet wurden, wird bei Version 2 darauf verzichtet. Die Überprüfung dafür, welche Aktionen möglich sind, wird in Listing 6 dargestellt.

Zum besseren Verständnis ziehen wir die Abbildung 6 hinzu. Jedes der Felder ist mit seinem Index versehen. Wollen wir ein Feld nach oben bewegen, ist dies nur möglich, wenn es sich in den unteren beiden Zeilen befindet. Das kann man leicht berechnen, da der Index dafür ≥ 3 sein muss (Z. 10). Ebenso einfach ist es zu bestimmen, ob ein Feld nach unten bewegt werden kann. Dafür muss der Index ≤ 5 sein (Z. 7).



Abbildung 7: Das Ergebnis der Durchführung von Version 2 des Tiefensucheargorithmus mit ID

Nicht mehr ganz so leicht, aber immer noch nicht schwer, ist die Überprüfung, ob eine Bewegung nach links oder rechts möglich ist. Ein Feld kann sich nicht nach links bewegen, wenn es in der ersten Spalte ist. Für alle Indizes in der ersten Spalte gilt: $index \bmod 3 = 0$. Wenn die Berechnung nicht 0 ergibt ist eine Bewegung nach links möglich (Z. 4).

Eine Bewegung nach rechts ist nicht möglich, wenn sich das entsprechende Feld in der letzten Spalte befindet. Für alle Indizes in der letzten Spalte gilt: $index \bmod 3 = 2$. Wenn die Berechnung nicht 2 ergibt ist eine Bewegung nach rechts möglich (Z. 1).

Diese einfachen Überprüfungen ersparen uns die Berechnung der xy-Koordinaten und erhöhen somit die Geschwindigkeit des Algorithmus.

Der weitere Ablauf des Algorithmus unterscheidet sich nicht ausschlaggebend von der vorherigen Version und wird somit nicht weiter beschrieben.

Wenn das Beispiel, im Ordner `Version 2 Tiefensuche mit ID`, auf dem AKSEN-Board durchgeführt wird erhalten wir das Ergebnis, welches in Abb. 7 dargestellt ist. Zu sehen ist die Aktionsliste welche auf den Initialzustand `{g, a, e, f, d, b, o, c, h}` angewendet werden muss um zum Finalzustand `{d, o, e, f, g, b, c, a, h}` zu gelangen. Auch Version 2 kommt auf das gleiche Ergebnis wie Version 1.

4.3.2. Speicherverbrauch

Für den globalen Speicher mit einer maximalen Suchtiefe von 30:

- 3 mal `bool` = 3 Byte
- 1 mal Agenda bestehend aus `char` = $33 \cdot 12 = 396$ Byte
- 4 mal `char` = 4 Byte
- 2 Puzzle bestehend aus jeweils 9 `char` = 18 Byte
- 1 globale Aktionsliste bestehend aus `char` = 30 Byte

- Gesamt = 451 Byte

Daraus können wir folgende Funktion zur Berechnung des globalen Speichers erstellen:

$$f(x) = 13x + 63, \text{ für } x = \text{maximale Suchtiefe}$$

Wenn wir für $f(x)$ nun unsere maximal verfügbaren 7116 Byte im globalen Speicher einsetzen, erhalten wir rund 542,5 für x . Also kann dieser Algorithmus spechertechnisch bis maximal Tiefe 542 suchen. Für die Funktion wurden zwei der genutzten **char**-Variablen durch **int**-Variablen getauscht, da bei so einer Tiefe die Größe eines **char** nicht mehr ausreichen würde.

Der maximale Speicherverbrauch das Stacks kann auf dem PC mit dem zusätzlichen Flag **MEM** berechnet werden. Die Berechnung ergibt dann, dass auf dem Stack maximal 43 Byte gleichzeitig durch das Programm abgelegt werden.

Die generierte .ihx-Datei ist 7977 Byte groß und passt somit auf den Programmspeicher.

Wir können sehen, dass der Speicherplatz des AKSEN-Board ausreicht um unseren Algorithmus ausführen zu können. Des weiteren sehen wir, dass die zweite Version der Tiefensuche mit ID der ersten Version spechertechnisch in allen Punkten überlegen ist.

4.3.3. Zeitmessung

Lösungslänge	Schrittweite	Zeit in ms	Gefundene Lösungslänge
8	1	1100	8
	2	583	8
	3	696	8
	5	901	8
	10	784	8
9	1	1650	9
	2	1276	9
	3	456	9
	5	473	9
	10	357	9
10	1	4276	10
	2	2201	10
	3	4498	10
	5	1141	10
	10	986	10
11	1	6503	11
	2	5804	11
	3	4227	11
	5	16616	11
	10	44485	19
	15	14433	11
12	1	10467	12
	2	5801	12
	3	4324	12
	5	16968	12
	10	84325	20
	15	15019	12
18	1	384025	18
	2	237011	18
	3	194603	18
	5	111304	20
	10	78383	20

Tabelle 2: Zeitmessung der Version 2 Tiefensuche mit ID

Die Geschwindigkeit des Algorithmus ist extrem gestiegen und es sind auch längere Lösungen in annehmbarer Zeit möglich. Jedoch braucht der Algorithmus für eine Lösung der Länge 18 immer noch sehr lange. Wir sehen auch, dass ab einer größeren Schrittgröße die gefundene Lösungslänge nicht unbedingt optimal ist.

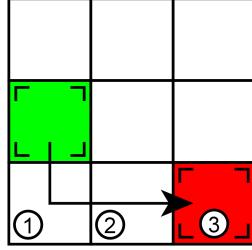


Abbildung 8: Manhattan-Distanz von grün nach rot

4.4. A*

4.4.1. Beschreibung

A* bietet einen weiteren Ansatz das Schiebepuzzleproblem zu lösen. Um diesen durchführen zu können benötigen wir eine Kostenfunktion $g()$ und eine Heuristikfunktion $h()$. Die Kostenfunktion ist schnell gefunden und beträgt $g() = \text{Tiefe des Knotens}$. Für die Heuristik gibt es ebenfalls eine einfache Lösung, die Manhattan-Distanz. Es wird für jedes Teil im aktuellen Puzzlezustand die Manhattan-Distanz zu seiner Position im Finalzustand berechnet und die Werte werden zusammenaddiert. Wie die Manhattan-Distanz berechnet wird ist in Abb. 8 zu sehen. Der Abstand vom grünen zum roten Feld wird Schachbrettartig gemessen und beträgt 3.

Wenn sowohl $g()$, als auch $h()$ berechnet wurden, werden diese zusammenaddiert und ergeben den Wert des Knotens. Der Algorithmus zur Berechnung der Heuristik ist im Listing 7 beschrieben. Wie wir sehen, wird für jedes Teil seine entsprechende xy-Position berechnet. Die jeweiligen x- und y-Werte können dann voneinander abgezogen, der Betrag davon gebildet und abschließend dem Heuristikwert hinzu addiert werden. Die vier Zeilen 6,7,10 und 11 können an sich gespart werden, da man das Zwischenspeichern der xy-Werte nicht benötigt, es macht den Code jedoch deutlich lesbarer und der Arbeitsspeicherverbrauch steigt nicht signifikant an.

Der Knoten wird dann anhand seines Wertes in die Agenda eingesortiert von klein nach groß, denn je kleiner der Wert, desto besser. Das Einsortieren stellt jedoch schon ein Problem dar. Denn zum einen muss der Algorithmus jedes mal durch die Agenda durch um den neuen Eintrag mit den anderen vergleichen zu können. Zum anderen muss er dann alle Einträge, welche vor dem neuen Eintrag sitzen um eins verschieben. Das ist ein großer Rechenaufwand.

Wir führen für unsere Agenda eine neue Datenstruktur, die Linked List, ein. Diese erspart uns das Verschieben aller Einträge, da die Einträge der Agenda nicht physisch sondern nur logisch sortiert sind. Das heißt, dass Eintrag A und B, welche in der Liste nebeneinander liegen nicht zwingend im Arbeitsspeicher nebeneinander liegen müssen, wie in Abb. 9 dargestellt. Eintrag A hat nun einen Link, welcher auf den Arbeitsspeicherplatz verweist, an welchem sich B befindet. B wiederum hat einen Link zu C und C hat einen Link zu D. Dieser ist das letzte Element der Liste und hat somit keinen Nachfolger und auch keinen Link. Wenn wir immer wissen wo im Arbeitsspeicher sich

```

1 unsigned char calcHeuristic(unsigned char *puzzle, unsigned char *fPuzzle)
2 {
3     unsigned char i, j, x_i, y_i, x_j, y_j;
4     unsigned char heuristic = 0;
5     for (i = 0; i < PUZZLE_LENGTH; i++) {
6         x_i = i % 3;
7         y_i = i / 3;
8         for (j = 0; j < PUZZLE_LENGTH; j++) {
9             if (*puzzle + i) == *(fPuzzle + j)) { // die Teile sind
10                gleich
11                x_j = j % 3;
12                y_j = j / 3;
13                heuristic += abs(x_i - x_j) + abs(y_i - y_j);
14            }
15        }
16    }
17    return heuristic;
}

```

Listing 7: Berechnung Heuristik

das erste Element befindet, können wir anhand der Links die komplette Liste verfolgen.

Wenn wir zwischen B und C das N einfügen wollen, brauchen wir nur einen freien Arbeitsspeicherplatz, setzen den Link in N zu dem Link, der in B steht, also nach C, und setzen den Link in B nach N. So haben wir ohne eine Verschiebung einen Eintrag in die Liste eingesortiert. Das Ergebnis ist in Abb. 10 dargestellt.

Um eine Linked List in unserem Code umzusetzen benötigen wir eine zusätzliche Liste, welche alle freien Positionen im Arbeitsspeicher enthält. Diese wird als Stack organisiert und gibt immer den obersten freien Platz zurück. Sobald ein Platz in der Agenda nicht mehr benötigt wird, wird dieser auf diesen Stack wieder zurückgelegt.

Die Links sind zusätzliche **chars** in einem Agendaeintrag. Das bringt jedoch ein Problem mit sich. Dadurch sind wir auf 255 Plätze beschränkt, da ein **char** nur 256 Werte

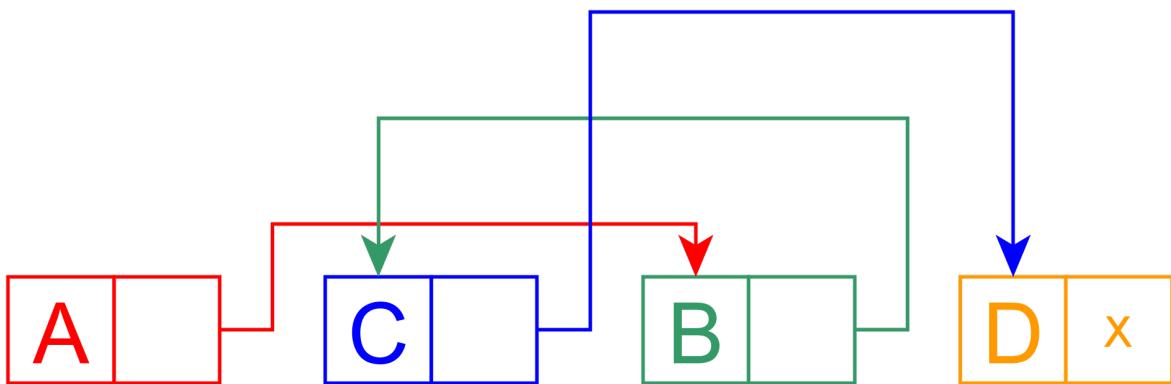


Abbildung 9: Linked List

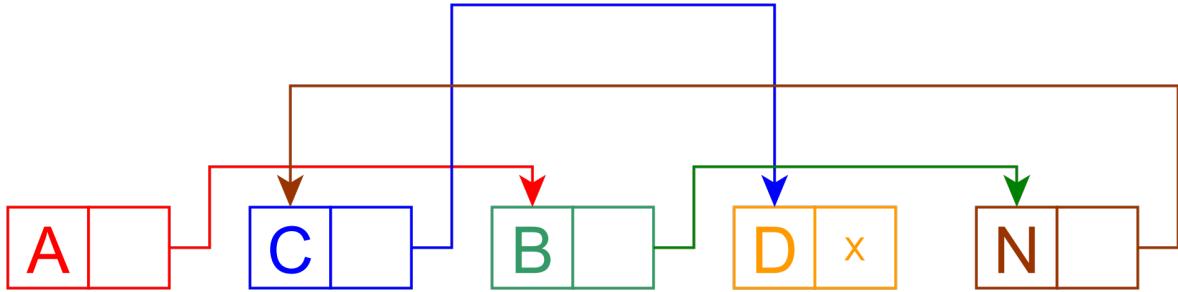


Abbildung 10: Linked List mit neuem Eintrag

fassen kann und wir noch einen benötigen um das Ende der Liste zu markieren. Wenn wir mehr Plätze zur Verfügung haben wollen, müssen wir eine zusätzliche Link-Liste implementieren, welche die Links für die einzelnen Agendaeinträge hält. Da diese jedoch aus Integern bestehen müsste, wäre der Arbeitsspeicherverbrauch auch doppelt so hoch. Dieses Problem wird in Kapitel 4.4.2 genauer beschrieben und ist auch der Grund, weshalb 255 Agendaplätze momentan das Maximum bilden.

Im Anhang A befindet sich das Listing 9, welches das Einfügen eines neuen Agendaeintrages darstellt. In Zeile 5 und 6 werden `postLink` und `prePointer` mit 255 initialisiert. Dabei steht `postLink` für den Link auf welchen der neue Eintrag verlinken soll und `prePointer` für den Agendaeintrag, welcher auf den neuen Eintrag verlinken soll. Dann wird eine Position aus dem `memoryPositions`-Array geholt (Z. 7). Nun müssen wir durch die ganze Agenda durchlaufen und die Werte der Einträge mit dem Wert des neuen Eintrages vergleichen (Z. 10-21). Wenn der Wert eines Eintrages in der Agenda kleiner ist, als der Wert des neuen Eintrages `v`, dann muss der neue Eintrag hinter diesem eingesertiert werden. Der entsprechende Eintrag wird als Vorgänger gemerkt und sein Link wird der neue `pointer`. Wenn ein Eintrag gefunden wurde, dessen Wert größer oder gleich `v` ist, wird dessen Link in `postLink` gespeichert und die `while`-Schleife wird abgebrochen. Dieser Vorgang läuft solange bis es keinen weiteren Eintrag in der Agenda mehr gibt oder ein Eintrag gefunden wurde, dessen Wert größer oder gleich `v` ist.

Nun wird der neue Eintrag an die Stelle im Arbeitsspeicher geschrieben, welche durch `memoryPositions` zur Verfügung gestellt wurde. Der Link wird entsprechend auf den Wert von `postLink` gesetzt (Z. 30). Wenn der `prePointer == 255` ist, gibt es keinen Vorgänger. D.h. der neue Eintrag ist das erste Element in der Agenda und der `agendaPointer` muss auf den neuen Eintrag zeigen (Z. 36). Sollte es jedoch einen Vorgänger geben so wird der Link des Vorgängers auf die Position des neuen Eintrages gesetzt (Z. 40).

Kommen wir zu einem weiteren Problem. Wie wir in Kapitel 4.3.1 gesehen haben, ist die Nutzung einer globalen Aktionsliste sehr praktisch. Diese fällt nun in A* wieder weg, da in A* nicht garantiert werden kann, dass in der Tiefe nur nach oben gesprungen wird. In A* kann man genauso gut nach unten in der Tiefe springen und würde somit falsche Ergebnisse in der globalen Aktionsliste bekommen. Also muss die Aktionsliste wieder in den Agendaeintrag verschoben werden und für jeden Puzzlezustand einzeln mitgeführt

werden. Jedoch wollen wir auch auf das eigentliche Puzzle in der Agenda nicht verzichten, da wir gemerkt haben, dass die erneute Berechnung viel zu hohen Rechenaufwand hat.

Dieses Problem beschränkt uns stark in unserer maximalen Suchtiefe, welche momentan auf 12 gesetzt ist. Eine Möglichkeit die Größe der Aktionsliste zu minimieren, ist nicht nur eine Aktion durch ein Byte zu kodieren, sondern vier. Eine genauere Beschreibung befindet sich auch wieder in Kapitel 4.4.2. Diese Möglichkeit wurde vorerst nicht weiter implementiert und getestet. Somit hat unser Algorithmus 255 Plätze in der Agenda und kann eine Tiefe von 12 erreichen.

Eine weitere Möglichkeit weiter in die Tiefe vordringen zu können ist das wiederholte Ausführen des Algorithmus. Da alle unsere Puzzlezustände nun einen Wert haben, können wir auch den Zustand mit dem besten Wert nehmen und diesen als neuen Initialzustand setzen und den Algorithmus von vorne laufen lassen. Dafür brauchen wir uns nur die 12 Aktionen zu merken, welche zu dem neuen Initialzustand geführt haben. Das Ganze können wir auch sehr oft wiederholen und müssen dabei nur beachten, dass die Historie, welche unsere vorangegangenen Aktionen speichert, mit jeder Iteration immer mehr Speicher verbraucht.

Diese Variante wurde auch prototypmäßig implementiert und getestet. Für die Historie wurde 252 Byte Speicher reserviert, sodass eine Lösungslänge von 264 möglich war. Es hat sich jedoch herausgestellt, dass die Lösungen, welche gefunden wurden, teilweise das 10-fache an Länge hatten, als die optimale Lösung oder teilweise gar keine Lösung innerhalb von 264 Schritten gefunden wurde, für Probleme mit einer optimalen Lösungslänge von > 12 . Aus diesem Grund wurde der Prototyp nicht weiterentwickelt. Jedoch sollte dieser Ansatz nicht aus den Augen verloren werden, da er vielversprechend zu sein scheint. Eine mögliche Optimierung kann in der Auswahl der neuen Initialzustände liegen.

Kommen wir nun zu der Beschreibung des A* wie er in unserem Projekt ist. Grundsätzlich unterscheidet er sich in seiner Arbeitsweise auch kaum von den beiden vorhergehenden Algorithmen, bis auf die Unterschiede, welche bereits besprochen wurden.

Zuerst wird überprüft, ob eine Lösung möglich ist. Ist dies der Fall so beginnt der eigentliche Algorithmus. Dafür wird die lokale maximale Tiefe um den entsprechenden **DEPTH_ITERATOR** erhöht und der erste Knoten wird aus der Agenda genommen. Es wird überprüft, ob der Knoten weiter expandiert werden darf. Wenn ja, dann werden die neuen Puzzlezustände gebildet, deren Werte berechnet und sie werden entsprechend ihrer Werte in die Agenda eingesortiert. Wenn der Knoten nicht weiter expandiert werden darf wird der Speicherplatz wieder freigegeben und auf die **memoryPositions** zurückgelegt. Dieser Algorithmus läuft solange bis eine Lösung gefunden wurde, oder die Agenda leer ist.

Wenn das Beispiel, im Ordner **Astar**, auf dem AKSEN-Board durchgeführt wird erhalten wir das Ergebnis, welches in Abb. 7 dargestellt ist. Zu sehen ist die Aktionsliste welche auf den Initialzustand $\{g, a, e, f, d, b, o, c, h\}$ angewendet werden muss um zum Finalzustand $\{d, o, e, f, g, b, c, a, h\}$ zu gelangen. A* hat das gleiche Ergebnis wie seine zwei Vorgänger.

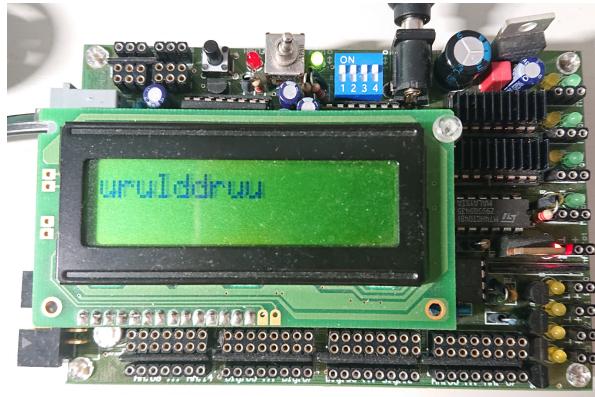


Abbildung 11: Das Ergebnis der Durchführung von A*

4.4.2. Speicherverbrauch

Für den globalen Speicher mit einer maximalen Suchtiefe von 12:

- 2 mal `bool` = 2 Byte
- 2 mal `int` = 4 Byte
- 1 mal `char` = 1 Byte
- 2 Puzzle bestehend aus jeweils 9 `char` = 18 Byte
- 1 mal Agenda bestehend aus `char` = $255 \cdot 26 = 6630$ Byte
- 1 mal Liste mit allen freien Plätzen der Agenda bestehend aus `char` = 255 Byte
- Gesamt = 6910 Byte

Daraus können wir folgende Funktion zur Berechnung des globalen Speichers erstellen:

$$f(x, y) = y(15 + x) + 25, \text{ für } x = \text{maximale Suchtiefe und } y = \text{Agendalänge}$$

Wenn wir für $f(x, y)$ 7116 einsetzen und $x = 12$ setzen, können wir ausrechnen, dass wir sogar Speicherplatz für bis zu 262 Agendaeinträge hätten. Jedoch würde uns dann ein `char` als Pointer nicht mehr reichen. Dies würde bedeuten, dass die Liste mit freien Plätzen aus Integern bestehen und auch die Links in den Agendaeinträgen in eine extra Liste aus Integern ausgelagert werden müsste. Wenn man dies tut ändert sich die Funktion folgendermaßen:

$$g(x, y) = y(17 + x) + 25$$

Wenn wir $x = 12$ und $y = 255$ setzen, kommen wir auf 7420. Das ist mehr, als wir Arbeitsspeicher zur Verfügung haben, bei gleicher Leistung. Die höchstmögliche Anzahl an Plätzen in der Agenda bei einer Tiefe von 12 beträgt 244. So kommen wir also nicht auf mehr Agendaplätze. Es gibt jedoch noch eine weitere Möglichkeit den Speicherverbrauch zu verringern. Um eine Aktion zu kodieren benötigt man eigentlich nur 2 Bit, da es lediglich vier verschiedene Aktionen gibt. Bisher verwenden wir jedoch ein ganzes Byte. Man könnte also auch 4 Aktionen mit einem Byte kodieren und hat somit nur noch ein Viertel an Speicherverbrauch für die Aktionsliste. Die neue Funktion sähe dann wie folgt aus:

$$h(x, y) = y(17 + \left\lceil \frac{x}{4} \right\rceil) + 25$$

Bei gleicher maximaler Tiefe und Agendalänge kommen wir nun auf 5125 Byte Speicherverbrauch. Bei einer Tiefe von 12 stehen sogar bis zu 354 Agendaplätze zur Verfügung. Nun können wir unsere maximale Tiefe bis 32 erhöhen, um an die vorhergehenden Algorithmen heranzukommen, und haben dann 283 Plätze zur Verfügung.

Das klingt sehr gut, jedoch gibt es ein Problem. Die Anzahl der Knoten, welche sich zeitgleich in der Agenda befinden, wird immer größer. Bereits bei einer Tiefe von 12 befinden sich ca. 200 Knoten gleichzeitig in der Agenda. 283 Plätze werden dann für eine Tiefe von 32 kaum reichen. Es ist auch nicht geklärt wie stark die Operationen zur Manipulation der Aktionsliste die Geschwindigkeit beeinflussen.

Die beiden genannten Gründe waren ausschlaggebend dafür, dass erst einmal nicht weiter auf diese Idee eingegangen wurde, da sie nicht so leicht zu implementieren und wenig erfolgversprechend ist.

Der maximale Speicherverbrauch das Stacks kann auf dem PC mit dem zusätzlichen Flag `MEM` berechnet werden. Die Berechnung ergibt dann, dass auf dem Stack maximal 83 Byte gleichzeitig durch das Programm abgelegt werden.

Die generierte `.ihx`-Datei ist 13172 Byte groß und passt somit auf den Programmspeicher.

4.4.3. Zeitmessung

Lösungslänge	Schrittweite	Zeit in ms	Gefundene Lösungslänge
8	1	8052	8
	2	3542	8
	3	3060	8
	4	1624	8
	6	2706	8
	12	1021	8
9	1	14736	9
	2	10187	9
	3	3119	9
	4	8258	9
	6	2767	9
	12	1072	9
10	1	32962	10
	2	12035	10
	3	18541	10
	4	9269	10
	6	3300	10
	12	843	10
11	1	53325	11
	2	35798	11
	3	16746	11
	4	8619	11
	6	3483	19
	12	1649	11
12	1	86176	12
	2	28997	12
	3	13003	12
	4	7240	12
	6	2616	20
	12	1016	12

Tabelle 3: Zeitmessung A*

Deutlich zu erkennen ist, dass die Schrittweite von 1 nicht mehr für A* zu bevorzugen ist. Da wir mit A* nur bis Tiefe 12 kommen, spielt das *iterative Deepening* keine große Rolle und kann auch vollständig weggelassen werden. In unserem Code ist das ID, dennoch enthalten und lediglich die Schrittweite wurde auf 12 gestellt, um das gleiche Verhalten zu erzeugen. Weiterhin sehen wir, dass A* mit maximaler Schrittweite seinen Vorgängern klar überlegen ist. Während Version 1 für eine Lösungslänge von 12 ein Minimum von 140 Sekunden und Version 2 ein Minimum von 4 Sekunden brauchte, braucht A* nur

noch 1 Sekunde. Aber auch für alle anderen Lösungslängen gilt, dass sie in ungefähr 1 bis 2 Sekunden lösbar sind.

5. Fazit

Wenn einem wichtig ist lange Lösungswege zu lösen, sollte man auf die Version 2 als Algorithmus zurückgreifen, da dieser Lösungslängen bis 18 in annähernd annehmbarer Zeit lösen kann. Sollte man jedoch eine Begrenzung bis Lösungslänge 12 haben, ist der A* definitiv zu bevorzugen, da dieser alle Probleme der maximalen Länge 12 in 1 bis 2 Sekunden lösen kann. Generell sollte man sein Hauptaugenmerk auf die Weiterentwicklung des A* legen, da dieser durchaus erfolgversprechende Ansätze bietet. Gerade die wiederholte Durchführung des A*, vielleicht sogar kombiniert mit der verkürzten Aktionsliste, kann zum Erfolg führen.

Eine weitere denkbare Möglichkeit zur Lösung des Problems ist ein neuronales Netz. Dieses müsste vorher auf einem PC trainiert und anschließend auf das AKSEN-Board übertragen werden. Dann könnte das neuronale Netz für jeden Schritt eine Vorhersage treffen und somit das Problem schrittweise lösen. Die hier entwickelten Algorithmen können dabei als Lösungsvorlage dienen um randomisierte Trainingsdaten zu erzeugen und somit den Trainingsprozess zu automatisieren.

Abschließend ist zu sagen, dass eine Lösung des Achterpuzzle-Problems auf dem AKSEN-Board möglich ist, solange die Lösungslängen nicht zu lang werden. Jedoch sind Ansätze des A* vielversprechend und auch das neuronale Netz kann gute Lösungen hervorbringen.

A. Listings

```
1  bool calcChildrenAndAddToAgenda () {
2      int xy[2];
3      unsigned char i;
4      unsigned char j;
5      int lastIndex = 0;
6      char lastMove = 'x';
7      int *coordinates;
8      for (i = 0; i < PUZZLE_LENGTH; i++) {
9          if (currentPuzzle[i] == 'o') {
10              coordinates = getXYFromIndex(i); // Berechnung der xy-
11                  Koordinate fuer das leere Feld
12              for (j = 0; j < 2; j++) {
13                  xy[j] = *(coordinates + j);
14              }
15          }
16      // Bestimmung der letzten Aktion
17      lastIndex = getLengthOfActionList() - 1;
18      if (lastIndex >= 0) {
19          lastMove = currentActionList[lastIndex];
20      }
21      if (xy[0] < 2 && lastMove != 'l') {
22          // Bewegung nach rechts ist moeglich
23          if (!createChildAndAddToAgenda('r')) {
24              return false;
25          }
26      }
27      if (xy[0] > 0 && lastMove != 'r') {
28          // Bewegung nach links ist moeglich
29          if (!createChildAndAddToAgenda('l')) {
30              return false;
31          }
32      }
33      if (xy[1] < 2 && lastMove != 'u') {
34          // Bewegung nach unten ist moeglich
35          if (!createChildAndAddToAgenda('d')) {
36              return false;
37          }
38      }
39      if (xy[1] > 0 && lastMove != 'd') {
40          // Bewegung nach oben ist moeglich
41          if (!createChildAndAddToAgenda('u')) {
42              return false;
43          }
44      }
45      return true; // Funktion wurde erfolgreich durchgefuehrt
46 }
```

Listing 8: Funktion calcChildrenAndAddToAgenda Version 1 Tiefensuche mit ID

```

1 void putInAgenda(unsigned char *puzzle, unsigned char d, unsigned char b,
2   unsigned char a, unsigned char v, unsigned char *al)
3 {
4   unsigned char i;
5   int pointer = agendaPointer;
6   int postLink = 255; // der Nachfolger des neuen Eintrags
7   int prePointer = 255; // der Vorgaenger des neuen Eintrags
8   unsigned char pos = memoryPositions[memoryPositionPointer]; // 
9     Position, an welcher der neue Eintrag im Arbeitsspeicher eingefuegt
10    wird
11   memoryPositionPointer--;
12   // Position zum Einfuegen in die Liste finden
13   while (pointer != 255) {
14     if (agenda[pointer][VALUE_INDEX] >= v)
15       { // der neue Eintrag muss direkt vor dem pointer-Eintrag
16         einsortiert werden
17         postLink = pointer;
18         break;
19       }
20     else
21     { // merke dir den pointer als Vorgaenger und gehe zum naechsten
22       Link
23       prePointer = pointer;
24       pointer = agenda[pointer][LINK_INDEX];
25     }
26   }
27   // Einfuegen des neuen Eintrags
28   for (i = 0; i < PUZZLE_LENGTH; i++) {
29     agenda[pos][i] = *(puzzle + i);
30   }
31   agenda[pos][DEPTH_INDEX] = d;
32   agenda[pos][BLANK_INDEX] = b;
33   agenda[pos][ACTION_INDEX] = a;
34   agenda[pos][VALUE_INDEX] = v;
35   agenda[pos][LINK_INDEX] = (unsigned char)postLink; // Link zum
36   Nachfolger wird gesetzt
37   for (i = 0; i < d; i++) {
38     agenda[pos][AL_INDEX + i] = *(al + i);
39   }
40   if (prePointer == 255)
41   { // es gibt keinen Vorgaenger; Der neue Eintrag ist das erste Element
42     der Agenda
43     agendaPointer = pos;
44   }
45   else
46   { // Der Link des Vorgaengers wird auf den neuen Eintrag gesetzt
47     agenda[prePointer][LINK_INDEX] = pos;
48   }
49 }

```

Listing 9: Das Hinzufügen eines neuen Agendaeintrages mit einer Linked List

B. Verschiedenes

B.1. Ablauf der Programmausführung auf dem AKSEN-Board

- Aufrufen des cygwin-Terminals
- navigieren zum Projektordner
- make im Terminal eingeben; eine .ihx wurde erzeugt
- AKSEN-Board mit Strom versorgen und mit einem Kabel verbinden
- Schalter nach rechts legen; die rote Lampe leuchtet
- Reset Knopf kurz gedrückt halten
- öffnen des Flashers
- die erstellte .ihx auswählen
- den richtigen Port wählen
- flashen
- Schalter des AKSEN-Board umlegen; die grüne Lampe leuchtet
- Reset Knopf kurz gedrückt halten

B.2. Ablauf der Programmausführung auf dem PC (Windows)

Um das Programm auf dem PC auszuführen wird `gcc` benötigt. Es kann auch ein anderer C-Compiler funktionieren, es wurde aber nur mit `gcc` getestet.

- Aufrufen der cmd
- Navigation zum Projektordner
- `gcc -o ps puzzlesolver.c -D PC` in der cmd eingeben
- `ps` in der cmd eingeben

Literatur

- [Kö00] Köller, J.: Fünfzehnerspiel, 14 - 15 - Spiel, Boss Puzzle, 2000, URL: <http://www.mathematische-basteleien.de/15erspiel.htm>.
- [RN02] Russell, S.; Norvig, P.: Artificial intelligence: a modern approach./, 2002.
- [05] Nutzerhandbuch AKSEN-Board. 2005.
- [Ge19] GeeksForGeeks: How to check if an instance of 8 puzzle is solvable?, 2019, URL: <https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/>.
- [He20] Heinsohn, J.: Suchverfahren, 2020.