Heuristische Algorithmen

am Beispiel des

A*-Algorithmus / 8-Puzzle

Eine Ausarbeitung für das Fach “Algorithmische Anwendungen”
im Studium der Allgemeinen Informatik WS05/06
der FH Köln, Campus Gummersbach

Sanjay Jena – Matrikelnr. 11037063
Nils Liebelt – Matrikelnr. 11036979

Inhaltsverzeichnis

Problemstellung

Das 8-Puzzle zählt zu den Standardproblemen, welche mit Hilfe von Algorithmen künstlicher Intelligenz gelöst werden können.

Das Feld besteht aus 8 Spielsteinen in einer 3*3-Matrix. Somit existiert ein leeres Feld, in welches nun benachbarte Spielsteine hineingeschoben werden.

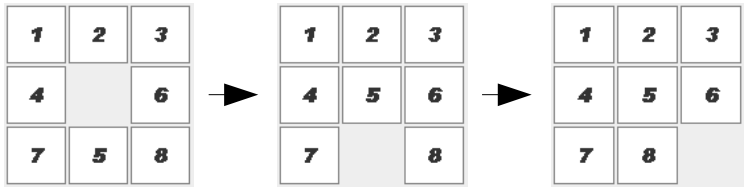


Abbildung 1: Lösungsschritte von einem gemischten Startzustand zum Zielzustand

Ausgehend von einem gemischten Startzustand müssen die Steine nun durch Verschieben in einen definierten Endzustand gebracht werden.

Zur Demonstration von Algorithmen, welche ein 8-Puzzle lösen, wird häufig auch eine höhere Anzahl von Steinen gewählt, wir z.B. beim 15-Puzzle in einer 4*4-Matrix.

Werkzeuge zur Lösung

Zur automatischen Lösung eines 8-Puzzles bedienen wir uns der Kürzeste-Wege-Algorithmen der Künstlichen Intelligenz. Eine Möglichkeit wäre, ausgehend vom Startzustand, alle möglichen Verschiebungen auszuprobieren, bis wir den Zielzustand erreicht haben. Dies würden Brute-Force-Algorithmen wie die Tiefensuche (Depth-search), Breitensuche (Breadth-search), limitierte Tiefensuche (Depth-limited-search) oder iterative Tiefensuche (iterative-depth-search) realisieren.

Um die Tauglichkeit solcher Algorithmen in der Praxis für unser Problem zu analysieren, setzen uns uns mit der asymptotische Laufzeit der Algorithmen auseinander: Lassen wir jeweils die Rückkehr zum vorherigen Zustand außer Acht, so besitzen wir in vier Positionen (jeweils die Ecken der Matrix) der Matrix genau eine weitere Verschiebe-Möglichkeit, in vier Positionen (die mittlere Position jeder Kante) zwei weitere Verschiebe-Möglichkeiten und in der mittleren Position drei weitere Möglichkeiten. Wir kommen also auf einen Branching Faktor von etwa 2.

Eine typischen Lösung besitzt etwa 20 Lösungsschritte. Eine Brute-Force-Suche würde also im schlimmsten Fall $3^{20} = 3.5 * 10^9$ Zustände traversieren. Begrenzen wir die Suche auf Zustände, welche noch nicht traversiert werden, so reduzieren wir die Anzahl der Zustände erheblich, erreichen jedoch immerhin noch 9! = 362.880 Zustände, die Anzahl aller Permutationen der Puzzle-Zustände.

Brute-Force-Algorithmen mögen also für kleine Problemgrößen brauchbar sein, für mittlere oder gar große Probleme sind sie jedoch vollkommen ungeeignet. Eine Mischung zweier Brute-Force-

Algorithmen in eine Bidirektionale Suche (bidirectional-search) erhöht die Erfolgschance, ist im Grunde aber immer noch ein Brute-Force-Algorithmus, welcher das Risiko lediglich halbiert.

Zur effizienten Lösung des Problems sind Algorithmen nötig, welche weniger Zustände traversieren und sich direkter auf den Zielzustand hinbewegen. Dies sind heuristische Algorithmen, welche durch bestimmte Kenntnisse über das Problem eine Schätzung vornehmen kann, welcher der folgende Möglichkeiten am erfolgversprechendsten ist. Ein solcher heuristischer Algorithmus ist A*, ein Dijkstra-Algorithmus, welcher um die Nutzung einer Heuristik erweitert wurde. Abhängig von der Güte seiner Schätzfunktion, der Heuristik, gelangt er schneller als all seine Mitstreiter zur optimalen Lösung.

Heuristiken

Historie

Der Begriff "Heuristik" wurde vom griechischen Verb "heuriskein" abgeleitet, welches die Bedeutung von "finden" oder "entdecken" hat. Die technische Bedeutung des Begriffs durchlebte im Laufe der Geschichte einige Änderungen. Im Jahre 1957 nutzte George Polya (Lit. [6]) den Begriff "Heuristik" ("heuristic") das erste mal in Bezug auf problemlösende Techniken und Suchmethodiken.

Von einigen wurde Heuristiken sogar als das Gegenteil von Algorithmik bezeichnet. 1963 veröffentlichten Simon, Shaw und Newell (Lit. [7]) "A process that may solve a given problem, but offers no guarantees of doing so, is called a heuristic for that problem".

In Wirklichkeit hängt die Qualität der durch einen heuristischen Algorithmus gefundenen Lösung von der Qualität der Heuristik ab. Es gibt keine Garantie, wie lange ein Algorithmus läuft. In einigen Fällen gibt es ebenso wenig eine Garantie für die Güte der Lösung.

Aktuelle Heuristiken sollen in der Regel die Average-Case-Laufzeiten von problemlösenden Algorithmen verbessern, was jedoch nicht notwendigerweise die Worst-Case-Laufzeit verbessert.

Im speziellen Gebiet der Suchalgorithmen beziehen sich Heuristiken auf Lösungskosten schätzende Funktionen. Der Begriff heuristische Suche (heuristic search) kam erstmals in einer von Newell und Ernst 1965 veröffentlichten Publikation (Lit. [3]) über generalisierte Suchalgorithmen auf. Dort basierte eine heuristische Suche auf einer Schätzfunktion, welche den Abstand eines Zustand zum Zielstatus schätzt, und somit Aussage über die Güte eines möglichen Lösungschrittes macht. Durch solche Schätzungen kann ein Algorithmus Pfade mit einer geschätzt niedrigen Distanz zum Ziel im Gegensatz zu weniger zielnahen Pfaden priorisieren. Dieses Konzept wurde im selben Jahr von Lin bestätigt (Lit. [4]). Ein Jahr später führten Doran und Michie intensive Experimente zur Problemlösung mit Hilfe von heuristischen Algoritmen durch (Lit. [5]), insbesondere zum 8- und 15-Puzzle. (Quelle: [8])

Bedingungen an eine Heuristik

Eine Heuristik garantiert nur dann eine optimale Lösung, also eine Lösung mit minimalen

Pfadkosten, wenn sie die reellen Pfadkosten zum Ziel niemals überschätzt. Solche Heuristiken nennt man *Admissable Heuristics* (zulässige Heuristiken).

Sei *f(n) = geschätzte Kosten der kostengünstigsten Lösung durch Knoten n*

Eine Heuristik *h* ist *zulässig* (*admissable*), wenn *f(n)* niemals die aktuellen Kosten der kostengünstigsten Lösung durch *n* überschätzt.

Eine Heuristik *h* ist *monoton*, wenn die f-Kosten entlang ihres Pfades von der Wurzel aus niemals abnehmen. Dies kann durch negative Pfadkosten entstehen. In einem Graphen mit ausschließlich positiven Pfadkosten, ist eine Heuristik somit immer monoton.

Heuristiken zur Lösung des 8-Puzzle

Um eine optimale Lösung zu finden, muss eine Heuristik gefunden werden, welche die Lösungskosten niemals überschätzt. Im folgenden zwei zulässige (admissable) Heuristiken:

- h_1 = Anzahl aller Rechtecke, welche sich in der falschen Position befinden

- h_2 = Summer aller Distanzen der Rechtecke zu ihrer Zielposition. Da die Rechtecke nicht vertikal verschoben werden können, ist die Distanz die Summe der vertikalen und horizontalen Distanz zum Ziel. Diese Heuristik wird auch *Manhattan-Distanz* genannt.

Heuristik h_2 dominiert h_1, da $h_2 \geq h_1$ für alle *n* (und beide zulässig sind).
Allgemein gilt: für beliebige zulässige Schätzfunktionen (admissable heuristics) h_i, h_j kann eine dominante Schätzfunktion *h* konstruiert werden: $h(n) = max(h_i(n), h_j(n))$

Performance-Untersuchungen

Betrachten wir die beiden Heuristiken, so gehen wir davon aus, dass die Manhattan-Heuristik aufgrund ihres intelligenteren Vorgehens weitaus weniger Schritte zum Ziel benötigt. Hierzu stellen wir einige Experimente an, um die Laufzeit des A*-Algorithmus mit den beiden Heuristiken zu vergleichen. Hierbei stellt sich zunächst die Frage, welches Kriterium wir messen. Ein häufig verwendetes Kriterium ist die Laufzeit des Algorithmus in Millisekunden. Da dieses Mess-Kriterium jedoch die Konstante Nutzung der CPU voraussetzt, entscheiden wir uns für die Anzahl der Knoten, welche der Algorithmus vom Startzustand bis zum Zielzustand traversiert.

Wir betrachten ein 15-Puzzle und erzeugen jeweils 100 Stichproben von Problemen mit einer Lösungslänge von 1 bis 22. Eine solche Menge von Stichproben liefert einen zuverlässigen Mittelwert. Wir stellen die Ergebnisse in einem Powertest dar, indem die Achsen logarithmiert werden (Abbildung 2 und 3). Hierdurch haben wir die Möglichkeit, das asymptotische Laufzeitverhalten in beiden Fällen direkt miteinander vergleichen zu können.

Unser Experiment zeigt ein interessantes, aber nicht unerwartetes Verhalten auf: Der A*-Algorithmus zeigt unabhängig von der Heuristik ein ähnliche asymptotische Laufzeit. Die Simpleheuristik traversiert hierbei jedoch ein Vielfaches an Knoten. Und zwar gilt: Umso grösser *n*,

also die Problemgrösse (Länge der Lösung), umso höher der Faktor, welchen die Simple-Heuristik länger braucht als die Manhattan-Heuristik.

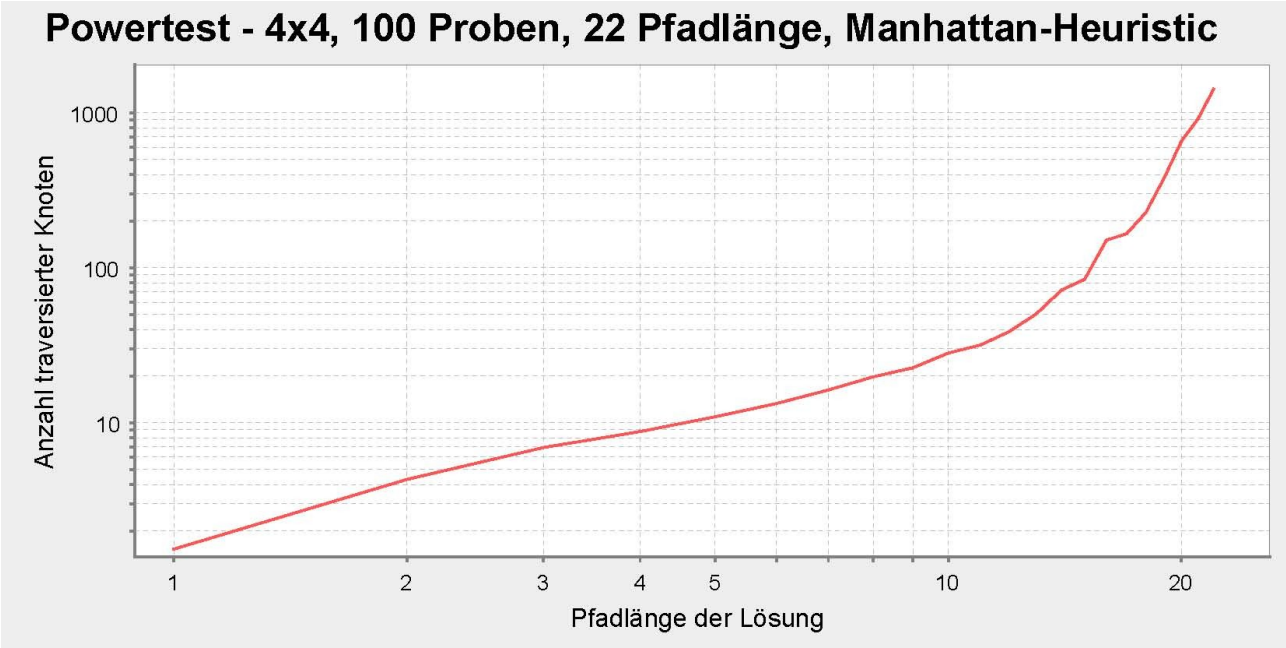


Abbildung 2: Powertest unter Verwendung der Manhattan-Heuristik

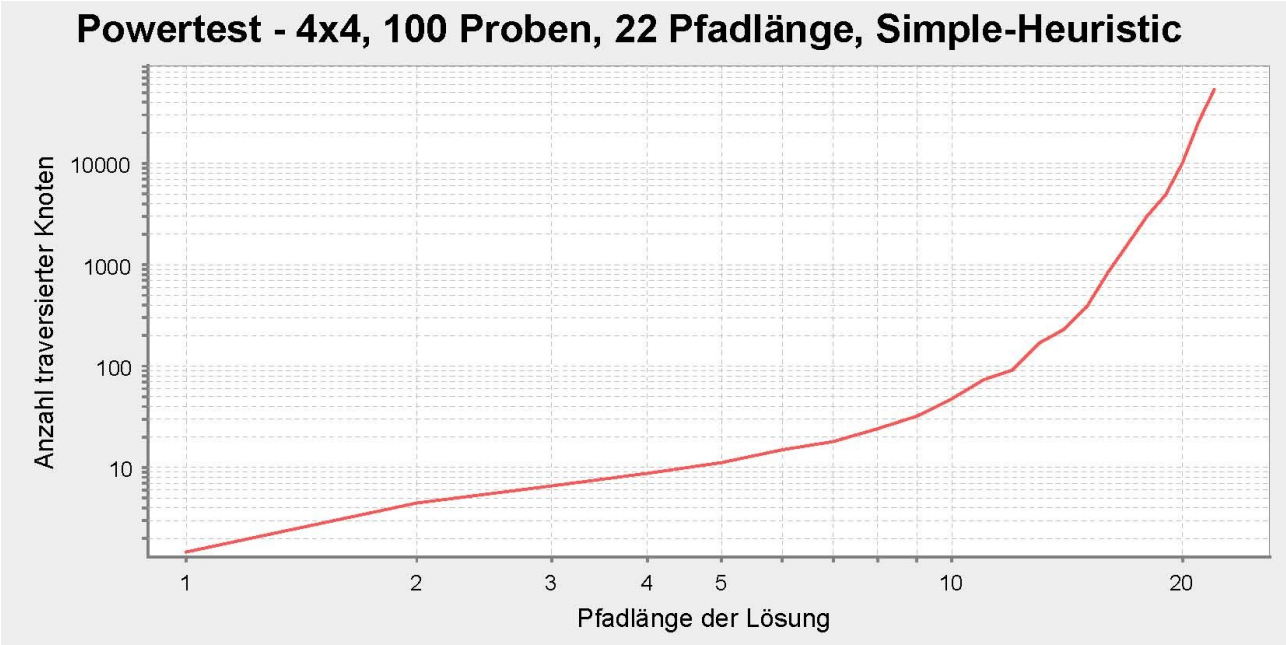


Abbildung 3: Ein Powertest unter Nutzung einer einfachen Heuristik zeigt das selbe Laufzeitverhalten wie bei der Manhattan-Heuristik, nur um einen Multiplikator c erhöht

A*-Algorithmus

Historie und aktueller Stand

Der A*-Algorithmus (gesprochen "A Star" bzw. "A Stern"), unter Einbeziehung der aktuellen Pfadlänge in eine heuristische Suche, wurde von Hart, Nilsson und Raphael entwickelt und1968 in "A Formal Basis for the Heuristic Determination of Minimum Cost Paths" (Lit. [1]) publiziert. Einige technische Unstimmigkeiten wurden später in einer weitere Veröffentlichung (Lit. [2]) korrigiert.

Seitdem ist der A*-Algorithmus bis heute immer wieder Gegenstand wissenschaftlicher Veröffentlichungen, welche sich sowohl mit der exakteren Beweisführung der Optimalität, Vollständigkeit und Komplexität des Algorithmus, als auch mit der Tauglichkeit des Algorithmus für Probleme aus der Praxis, auseinandersetzen. Für eine ausführliche Historie verweisen wir auf Lit. [8].

Heute ist der A*-Algorithmus einer der bekanntesten und, nicht zuletzt aufgrund seiner Einfachheit, einer der am häufigsten verwendeten heuristischen Algorithmen.

Funktionsweise

Die Vorgehensweise des Algorithmus basiert auf dem Shortest-Path-Algorithmus von Dijkstra. Laut Dijkstra werden ausgehend von einem Startknoten die Nachbarknoten verarbeitet. Ausgehend von den Nachbarknoten dann wiederum deren Nachbarknoten. Die Pfadkosten werden aufsummiert und es wird jeweils der Knoten zur Betrachtung der Nachbarn priorisiert, welcher die geringsten Pfadkosten besitzt. Somit ist auch sichergestellt, dass die erste gefundene Lösung auch eine optimale Lösung ist. Die Pfadkosten werden für jeden Knoten in Form einer Funktion abgebildet:

$f(n) = g(n)$ mit *g(n) = Summe aller Pfadkosten vom Startknoten zu n über den kostengünstigsten Pfad*

A* erweitert diese Vorgehensweise um die Berücksichtigung einer Schätzfunktion *h(n)*. Diese beschreibt für jeden Knoten *n* die geschätzte Entfernung zum Ziel:

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

Als Summe der reellen Kosten vom Startknoten bis n und den geschätzten Kosten von *n* bis zum Ziel, beschreibt *f(n)* beschreibt also eine Schätzung der gesamten Pfadkosten vom Startknoten bis zum Ziel über den Knoten *n*.

Wie auch bei Dijkstra priorisiert A* nun den Knoten mit dem geringsten *f(n)*.

Im folgenden der Pseudocode des A*-Algorithmus:

```
1  struct node {
2      node *parent;
3      int x, y;
4      float f, g, h;
5  }

   // A*
6  initialize the open list
7  initialize the closed list
8  put the starting node on the open list (you can leave its f at zero)

9  while the open list is not empty
10     find the node with the least f on the open list, call it "q"
11     pop q off the open list
12     generate q's 8 successors and set their parents to q
13     for each successor
14         if successor is the goal, stop the search
15         successor.g = q.g + distance between successor and q
16         successor.h = distance from goal to successor
17         successor.f = successor.g + successor.h

18         if a node with the same position as successor is in the OPEN list \
19             which has a lower f than successor, skip this successor
20         if a node with the same position as successor is in the CLOSED list \
21             which has a lower f than successor, skip this successor
22         otherwise, add the node to the open list
23     end
24     push q on the closed list
25 end
```

Das *struct* zur Darstellung eines Knotens in den Zeilen 1-5 ist eine Standarddatenstruktur für Knoten. Zeile 6 und 7 erzeugen leere Listen für die Knoten, welche noch abgearbeitet werden müssen (*open list*), und solche welche schon abgearbeitet wurden und nicht weiter berücksicht werden (*closed list*).

Wir iterieren solange übre die Knoten der open list, bis diese leer ist; in diesem Fall haben wir alle relevanten Knoten, welche für eine optimale Lösung in Frage kommen, durchlaufen. Für die Darstellung der beiden Listen empfiehlt sich eine sortierte Datenstruktur, um in konstanter Zeit auf den Knoten mit dem kleinsten *f*-Wert zugreifen zu können.

Der Knoten *q* mit dem kleinsten *f*-Wert wird der Liste entnommen und dessen Nachbarn überprüft, ob sie schon der Zielknoten sind. In diesem Fall kann die Suche beendet werden. Im anderen Fall wird die Entfernung zum Ziel geschätzt. Der Knoten wird nun in die open list übernommen, wenn

- der Knoten noch nicht in der *open list* vorhanden ist, zumindest nicht mit einem geringeren Schätzwert als aktuell, und
- der Knoten noch nicht in der *closed list* mit einem geringeren Schätzwert vorhanden ist.

Somit wird garantiert, dass jeder Knoten *n* jeweils mit dem kostengünstigsten Pfad (vom Start bis zu *n*) gesichert wird. Zu Ende der Schleife wird *q* nun in die *closed list* verschoben.

Optimalität

Der A*-Algoritmus ist optimal, wenn seine verwendete Heuristik *zulässig* und *monoton* ist. Im folgenden nun die Beweisführung (Quelle: Lit. [9]):

Zu zeigen: Der A*-Algorithmus findet immer einen kürzesten Pfad

Annahme: Der A*-Algorithmus findet eine suboptimale Lösung G2, wobei die optimale Lösung G1 Kosten von C1 hat.

Da für alle Zielknoten gilt dass die geschätzte Entfernung vom Zielknoten bis zum Zielknoten 0 ist, gilt insbesondere:

$h(G2) = 0$

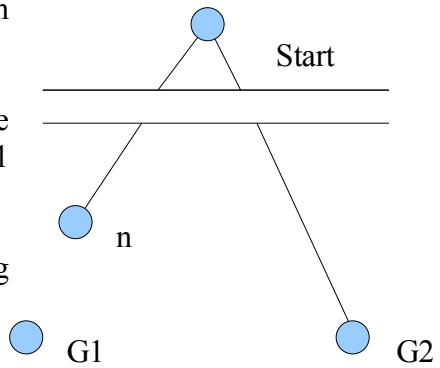


Abbildung 4: G2 ist suboptimales Ziel. n ist Knoten auf dem optimalen Pfad zu optimalem Ziel G1

Da G2 nach Annahme eine suboptimale Lösung ist, gilt nun aber folgende Gleichung:

$f(G2) = g(G2) + h(G2) = g(G2) > C1$

Betrachtet man nun einen beliebigen Knoten n auf dem kürzesten Pfad zum optimalen Ziel G1 und die Tatsache dass die Heuristik die tatsächlichen Kosten niemals überschätzt, gilt:

$f(n) = g(n) + h(n) \leq C1$

Fasst man nun die beiden Gleichungen zusammen, so erhält man:

$f(n) \leq C1 < f(G2)$

Dies bedeutet aber nun dass der A*-Algorithmus den Knoten G2 niemals erkundet bevor er die optimale Lösung (G1) findet. Somit findet der Algorithmus zuerst die Lösung G1, und berechnet damit tatsächlich einen kürzesten Pfad.

Eine Beweisführung, welche auf Widerspruch basiert, kann in anschaulicher Weise dem AI-Standardwerk "Artificial Intelligence – A Modern Approach" (Lit. [8]) von Stuart J. Russel und Peter Norvig entnommen werden.

Vollständigkeit

A*expandiert die Knoten in aufsteigender Reihenfolge von f. Letztendlich muss der Algorithmus also auch einen Zielknoten travessieren. Dies ist gegeben, solange keine unendliche Anzahl von Knoten mit $f(n) < f^*$ vorhanden ist. Dies wäre der Fall, wenn der *branching factor* (Anzahl der Söhne eines Knotens) eines bestimmten Knotens unendlich wäre, oder ein Pfad mit endlichen Kosten, jedoch mit einer unendlichen Anzahl von Knoten exisitiert.

A* ist somit vollständig für lokal finitive Graphen (*locally finite graphs*, Graphen mit einer endlichen Anzahl von Söhnen), bei welchen die Pfadkosten zwischen zwei beliebigen Knoten immer positiv und grösser 0 ist.

Laufzeitkomplexität

Die Laufzeitkomplextität des Algorithmus ist stets abhängig von der Qualität der Heuristik. Es gibt perfekte Heuristiken, welche zur Folge haben, dass der Algorithmus nur die Knoten traversiert, welche auch auf dem Lösungspfad liegen. Solche Heuristiken machen in der Praxis jedoch wenig Sinn, da der Aufwand für ihre Berechnung unproportional hoch ist. Als Gegenteil existieren Heuristiken, welche kaum Rechenaufwand benötigen (vgl. erste Heuristik im Beispiel des 8-Puzzles dieser Ausarbeitung), bei welchen der Algorithmus jedoch kaum schneller ist als der Dijkstra. Eine gute Heuristik erreicht den Mittelweg zwischen perfekter und schlechter Performanz.

Unabhängig von der Heuristik (solange sie *zulässig* ist), besitzt A* in jedem Fall eine bessere Laufzeit als der Dijkstra-Algorithmus, welcher eine Komplexität von $O(n^2)$ besitzt. Unter Verwendung von Fibonacci-Heaps kann auch eine Laufzeit von $O(n*\log(n))$ erreicht werden.

Die Betrachtung der Powertests in Abbildung 2 und 3 bestätigt diese Aussage. Der Graph lässt sich aufgrund seines ungeraden Verlaufs keinem exponentiellen Wachstum im Schema $t(x)=b*x^c$ zuordnen. Aufgrund des geringeren Laufzeit bei kleinen *n* ergibt sich zunächst eine sehr geringe Steigung, welche dann später durch grössere *n* deutlich verstärkt wird; ein analoges Verhalten wie bei *n*log(n)*.

Implementierung

Architektur

Ziel der Implementierung war es mit möglichst geringen Aufwand eine Software zu erstellen die robust, leicht erweiterbar und vor allem gut lesbar sein sollte. Unsere algorithmische Fragestellung sollte die zentrale Rolle der Applikation bekommen und auch im Quellcode möglichst keine Abhängigkeiten von der genutzten Infrastruktur haben. Ein weiterer wichtiger Aspekt war die Testbarkeit der Applikation. Vor allem in Verbindung mit Performancefragen war es wichtig einzelne Klassen und Pakete isoliert untersuchen zu können.

Um diese Anforderungen auch unter dem Gesichtspunkt des Aufwands zu realisieren haben wir uns für den Einsatz des bekannten Application Frameworks Spring entschieden. Lit[10,11]
Das Spring Framework basiert auf Rod Johnson's Ideen, welche erstmals in seinem Buch „Expert One-On-One J2EE Programming“ 2002 vorgestellt wurden. Der Kern des Frameworks besteht aus einer Umsetzung der bekannten Entwurfsmuster „Inversion of Control“ (IoC) bzw. „Dependency Injection“ durch einen einfachen XML-Kontext. Wir möchten an dieser Stelle nicht weiter auf die Programmiermethodiken eingehen jedoch einen kurzen pragmatischen Einblick in den Kontext unserer Applikation geben:

```
<bean id="puzzle" class="de.fhkoeln.fb10.ala.puzzle.domain.PuzzleImpl">
  <constructor-arg type="int">
    <value>3</value>
  </constructor-arg>
```

```
  <property name="puzzleObserver">
    <list>
      <ref bean="puzzlePane"/>
    </list>
  </property>
</bean>

<bean id="puzzlePane" class="de.fhkoeln.fb10.ala.puzzle.ui.PuzzlePane"
    init-method="initialize">
  <property name="puzzleState">
      <ref bean="puzzle"/>
  </property>
  <property name="spacing" value="4"/>
  <property name="padding" value="28"/>
  <property name="title" value="Puzzle Panel"/>
  <property name="movementTime" value="300"/>
</bean>
```

Der Spring IoC-Container erzeugt aus obigem XML zwei Objekte:

- Ein Model Objekt des Puzzles
- UI Panel welches das Puzzle graphisch repräsentiert

Daneben werden in diese Objekte Attribute und Abhängigkeiten injiziert. So bekommt das „puzzle“ Objekt eine Instanz des Puzzle-UI-Panel um ggf. bei Änderungen des Zustandes informieren zu können, dabei kennt das „puzzle“ jedoch nur den Typ:

```
public interface PuzzleListener {
          public void puzzleUpdate(Puzzle puzzle, Object event);
}

public class PuzzleImpl implements Puzzle {

          ...

          private List<PuzzleListener> puzzleObserver;
          ....
}
```

Die Anwendung des Containers macht es so möglich Abhängigkeiten zwischen den Objekten herzustellen die nur über Schnittstellen arbeiten und die implementierenden Klassen völlig voneinander Kapseln.

Spring findet heutzutage hauptsächlich in Verbindung mit der J2EE Technologie Verwendung und bietet zu fast allen bedeutsamen JAVA Technologien Schnittstellen. Was sich dort bereits für Anwendungen mit sehr großen Problemdomänen erfolgreich etabliert hat, funktionierte auch bei uns hervorragend. Bereits mit geringen Kenntnissen des Frameworks ist es uns gelungen das Projekt sehr viel klarer zu strukturieren was auch dem fachlichen Verständnis beigetragen hat.

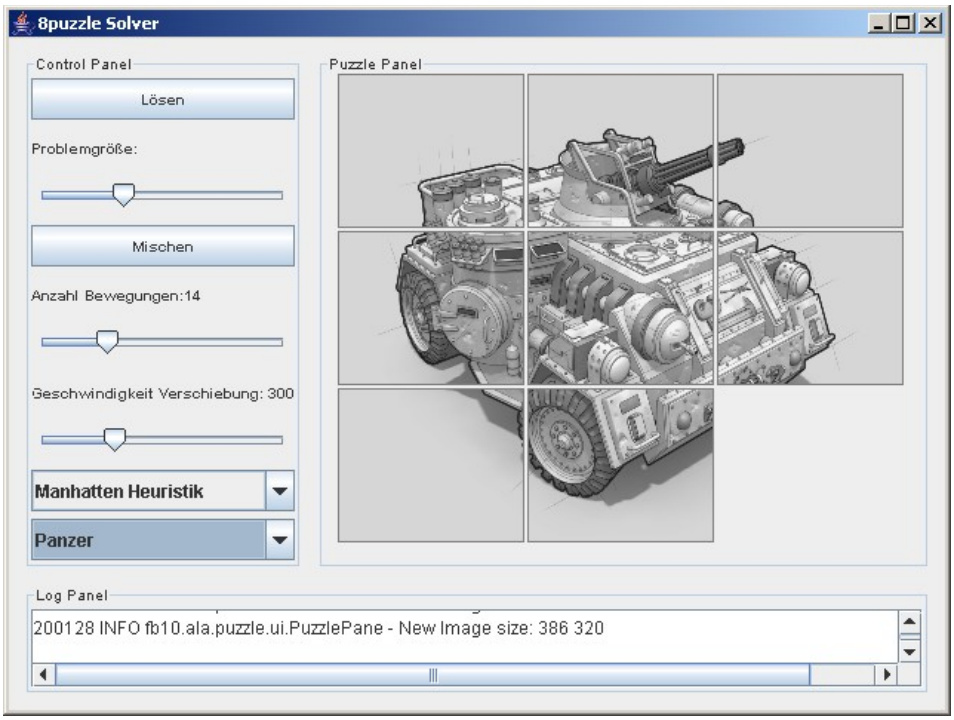


Abbildung 5: Die GUI bietet die Möglichkeit die Problemgrösse, die zu verwendende Heuristik und ein Bild für die Textur auszuwählen.

A*-Algorithmus

Die Implementierung des A* stellt eine Funktion solveProblem zur Verfügung, um sich die Lösung eines beliebigen Zustands berechnen zu lassen. Hierfür werden Start- und Zielzustand als Parameter übergeben, ebenso ein Comparator, welcher für den korrekten Vergleich der Goal-Distance verantwortlich ist.

Als Zustände werden Klassen genutzt, welche einer bestimmten Schnittstelle Node genügen müssen. Diese schreibt die Implementierung von Funktionen vor, welche Information über Goaldistance, Pfadkosten, Vater- und Sohnknoten geben.

Durch die Verwendung dieser Schnittstelle wird gewährleistet, dass sich der Algorithmus auch für andere Probleme als dem 8-Puzzle wiederverwenden lässt.

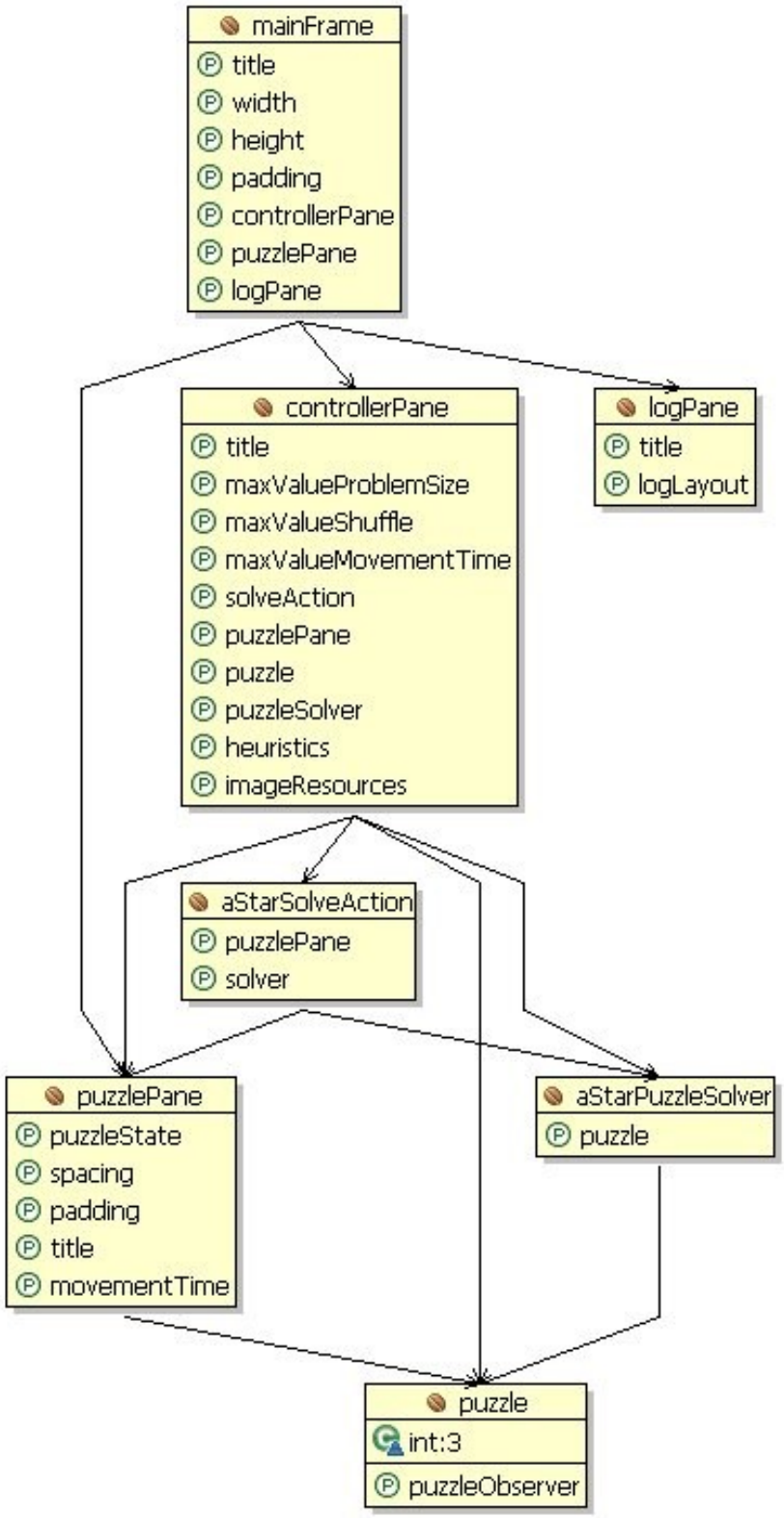
Heuristiken

Die Heuristiken werden den einzelnen Zuständen injeziert, um auf ihrer Basis die Goaldistance für jeden Knoten zu errechnen. Jede Heuristik genügt der Schnittstelle Heuristic, welche eine Funktion zur Errechnung der Goaldistance anbietet.

In unserem Fall implementieren wir zwei Heuristiken: Die Manhattan-Heuristik und eine einfache Heuristik. Eine fachlische Beschreibung kann dem Kapitel "Heuristiken zum Lösen des 8-Puzzle" entnommen werden.

Klassendiagramm der GUI

Das Diagramm zeigt die einzelnen Komponenten der GUI und Ihre Abhängigkeiten untereinander. Anhand der Attribute lässt sich auch erkennen wie sich die Applikation von außen konfigurieren lässt.



Ausblick auf andere Anwendungsfelder

Lösen eines Rubik-Würfels

Problemstellung: Es existieren zwölf verschiedene Verzweigungsmöglichkeiten, dass heißt in jedem Schritt zwölf verschiedene Möglichkeiten Änderungen am Rubik vorzunehmen.

Algorithmische Herausforderung: Reine Brute-Force-Algorithmen, wie z.B. ein Backtracking-Algorithmus sind in diesem Fall nicht einsetzbar. Besser in diesem Fall sind Algorithmen, welche sich der Lösung aufgrund von vorherigen Schätzungen dem Ziel schneller annähern, sprich heuristische Algorithmen.

Abbildung 6: Ein Rubik-Würfel

In Rubik-Wettbewerben kennen die Teilnehmer in der Regel etwa 50 verschiedene Vorgehensweisen und Algorithmen, um gewisse Farbsituationen herbeizuführen, welche dem Ziel näher sind. Dies zeigt die Komplexität der Problemstellung des Rubikwürfels auf. Der bisherige Weltrekord im Lösen eines Rubikwürfels durch einen Menschen beträgt 12 Sekunden. Herausforderung ist somit zum einen die algorithmische Lösung des Problems aus jedem Ausgangssituation heraus in einer Zeit, welche diesem in Relation steht, sprich maximal einer Sekunde.

Eine weitere Herausforderung ist es, eine passende "akzeptable Heuristik" (admissable heuristic) zu entwerfen, welche der Algorithmus zur Laufzeit errechnen kann und die Lösung des Problems in entsprechend wenig Schritten herbeiführt.

Literaturverzeichnis

[1] P. E. Hart, N. J. Nilsson, B. Raphael: *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Transactions on Systems Science and Cybernetics SSC4 (2), pp. 100-107, 1968.

[2] P. E. Hart, N. J. Nilsson, B. Raphael: *Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"*, SIGART Newsletter, 37, pp. 28-29, 1972.
Nilsson, Nils J.: *Problem-solving methods in artificial intelligence*, 1971

[3] Newell, A. and Ernst, C.: *The search for generality*. Proc. IFIP Congress 65, 1965

[4] S. Lin: *Computer solutions for the travelling salesman problem.* In: Bell Systems Tech. J. 44/1965

[5] J. Doran and D. Michie: *Experiments with the graph traverser program*. Proceedings of the Royal Society of London, 1966

[6] G. Polya: *How to Solve It*, Princeton University Press, 1957

[7] Allen Newell and Herbert A. Simon. GPS: *a program that simulates human thought*. In Edward A. Feigenbaum and Julian Feldman, editors, Computers and Thought, McGraw-Hill, New York, 1963

[8] Artificial Intelligence – A Modern Approach, Stuart J. Russel, Peter Norvig, Prentince Hall, 12/2002

[9] Wikipedia.de (http://de.wikipedia.org/wiki/A*-Algorithmus)

[10] Springframework – http://www.springframework.org

[11] Java Develoment with the Spring Framework – Johnson, Hoeller, Arendsen, Risberg, Sampaleanu Wiley Publishing 2005