**8-Puzzel Problem**

**Wie in der Aufgabenstellung der Aufgabe 1 und 2** **geschrieben** =>

Als erstes werden ein Start-Zustand und ein Ziel-Zustand definiert, und sind im Prinzip ein Array der Länge 9, die 3 Nummern am Anfang entsprechen die erste Zeile das Puzzle, die 3 Nummern in die Mitte entsprechen die 2 Zeile und die letzte 3 Nummern entsprechen die letzte Zeile.

Dann folgt eine Menge von Operatoren, und zwar für die Verschiebung der leeren Kachel, da es einfacher ist die leere Kachel anstatt einer anderen Kachel zur verschieben (Tipp von Prof. Dr. Krechel).

Hier (bei der Menge von Operatoren) wird der Zustand (state-Array, length = 9) nach 0 (leere Kachel) gesucht und überprüft ob, die Verschiebung möglich ist, denn z.B. bei der Methode **move\_up( state )** wenn der Index der leeren Kachel 1, 2 oder 3 ist also ganz oben, dann können wir dieser Kachel nicht nach oben schieben.

Am Ende wird eine neue Array **new\_state** zurückgegeben mit der neuen Position der leeren Kachel nämlich, wenn wir der Index der leeren Kachel mit 3 abstrahieren, so wird der neue Index in der richtigen Position geschrieben.

Analog zu **move\_up** gibt es die Methode **move\_down** hier wird auch überprüft ob die leere Kachel ganz unten steht (Index 6, 7 oder 8) dann können wir den nicht verschieben, ansonsten verschieben wir sie und ändern wir die Position im **state-Array** (Position der Kachel + 3) und geben wir sie zurück.

Sehr ähnlich sind die letzten Operatoren, und zwar move\_left und move\_right.

Im ( **class Node:** ) wird ein spezieller Datentyp erstellt (bzw. Klasse und Konstruktor), hier werden die folgenden Eigenschaften der Node implementiert, und zwar:

**1) state:** ist die aktuelle state-Array, die sich im aktuellen Node befindet.

**2) parent:** der vorherige Node.

**3) operator:** entsprechende Verschiebung => oben, unten, links oder rechts.

**4) depth:** die Tiefe des Baums.

**5) cost:** die Kosten. Im Verschiebungspuzzel sind die Kosten immer gleich, aber wenn wir routenplaner haben, dann ändern sie sich und sind dann sehr wichtig für die Berechnung.

**getState, getDepth, getParent, getMoves, getCost** sind Getter-Methoden.

**pathFromStart:** Diese Methode gibt an, der Lösungsweg von Wurzel (Root), da wir die Methode reverse() haben, ansonsten zeigt der Methode der Lösungsweg von aktuellen Node (die Lösung) bis zur Wurzel (Root).

**create\_node:** erstellt ein Node (bzw. ein Objekt der Klasse Node) (Also der neue Datentyp).

**expand\_node:** expandiert der aktuelle Node

**def search( start, goal, strategy ):**

Diese Methode ist der Hauptteil des Programms, hier werden die Such-Algorithmen (BFS, DFS und A\*) implementiert.

Die 3-Such-Algorithmen sind von der Implementation fast alle gleich außer ein paar Unterschiede.

Als erstes möchten wir erklären, wie der BFS-Algorithmus funktioniert (besser gesagt den Code erklären und ist fast analog zu dem Algorithmus (Pseudo-Code), die in der Vorlesungsfolien geschrieben ist), und dann werden wir die entsprechenden Unterschiede der 3-Algorithmen nennen.

**if strategy =="bfs" or strategy =="dfs" or strategy =="a\_star":**

Hier wird eine der 3-Algorithmen von der Main ausgewählt und an diese Methode übergeben.

**Nodes: Das Array**, die alle Nodes enthält,

**nodes.append( create\_node( start, None, None, 0, 0 ) )**

Hier wird als erstes die Wurzel-Node gespeichert.

**Count:** ist der Anzahl der besuchten Nodes (wichtig für die Aufwandschätzung)

explored-Array (in der Vorlesungsfolien heißt reached) enthält die Zustände, die schon besucht (bzw. überprüft) worden.

Dann wird durch das Nodes-Array mithilfe von While-schleife, die Nodes überprüft, ob deren Zustand das entsprechende Ziel erreicht.

if node.state==goal:

Falls wir das erreicht haben, dann stoppen wir und folgen wir mithilfe der node.pathFromStart() der entsprechende Lösungsweg ansonsten:

expanded\_nodes = expand\_node( node )

expandiert sich die Node und wird auch durch die while-Schleife weiter durchsucht und so weiter und sofort bis wir die Lösung (bzw. der Ziel-Zustand) finden.

Und nun kommen wir zu den Unterschieden zwischen der 3-Such-Algorithmen,

Bei der **BFS** und **A\* Algorithmen** werden die expandierte Nodes in nodes-Array **Ans Ende** hinzugefügt, Also wie ein Queue, so werden die Nodes Tiefen-weise durchsucht.

Bei der **DFS Algorithmus hingegen** werden die expandierte Nodes in nodes-Array **An den Anfang hinzugefügt,** Also wie ein Stack, so werden alle Nodes von links nach rechts durchsucht, Also wenn alle linke Nodes fertig durchsucht, dann die rechten etc.

Und bei der A\* wird **ein Sortierverfahren** auch implementiert.

**Board\_State**

Temp ist ein leeres Array ([0,0,0]), das die Länge von 3 hat (3 Zeilen pro Zeile 3 Werte), danach werden 2 for Schleifen iterieren, die innere für die Spalten und die äußere für die Zeilen,

jede durch lauf wird in temp in der richtigen Position die aktuelle Zahl gespeichert . Am Ende wird ein Array, das die aktuellen Zahlen in ihren richtigen Positionen, zurückgegeben.

**getFinalPosition**

temp ist ein leeres Array ([0,0]) ,das die Länge von 3\*3 hat, danach kommt for schleife die 9 mal läuft, zuerst wird der Index vom Zahl gesucht ,d.h. in welcher Stelle sich die Zahl in Array befindet, danach wird Index überprüft und anschließend die passende Koordinate bzw. Zahlen in temp gespeichert (Zeile, Spalte). am Ende wir ein Array, das die richtige Koordinate in richtige rheinfolge zurückgegeben

**H1** rechnet wie viel falsche Kacheln von aktuellem Zustand im Vergleich zum End-Zustand liegen, am Ende wir eine Zahl<=8 zurückgegeben.

**H2** rechnet die Entfernung von Kacheln bis Zielposition, indem jede aktuelle Zahl seine Position in (X,Y) mit dem Ziel-Zahl Position subtrahieret so bekommt man den Abstand. am Ende wir eine Zahl zurückgegeben.