

Aufgabenblatt 09

Heuristische Algorithmen – A-Star

Abgabe (bis 01.07.2024 23:59 Uhr)

Die folgenden Dateien werden für die Bepunktung berücksichtigt:

| | |
|----------------------------------|-------------|
| Blatt09/src/Board.java | Aufgabe 3.1 |
| Blatt09/src/PartialSolution.java | Aufgabe 3.2 |
| Blatt09/src/AStar15Puzzle.java | Aufgabe 3.3 |

Als Abgabe wird jeweils nur die letzte Version im main branch in git gewertet.

Aufgabe 1: Heuristische VS approximative Algorithmen

- 1.1 Was sind heuristische Algorithmen?
- 1.2 Was sind approximative Algorithmen? Was ist ein ρ -approximativer Algorithmus?
- 1.3 Wie unterscheiden sich approximative und heuristische Algorithmen?
- 1.4 Welche drei Kategorien von Problemen lassen sich in Bezug auf die Approximierbarkeit eines Problems aufstellen?

Aufgabe 2: A* Algorithmus: Heuristik

- 2.1 Wie funktioniert der A* Algorithmus? Nach welchem Kriterium wird der nächste Knoten ausgewählt? Welche Rolle spielt die Heuristik?
- 2.2 Gehen Sie die folgenden Beispiele für Heuristiken durch und diskutieren Sie, welchen Effekt diese auf den A* Algorithmus haben.

Abbildung 1A: Heuristiken

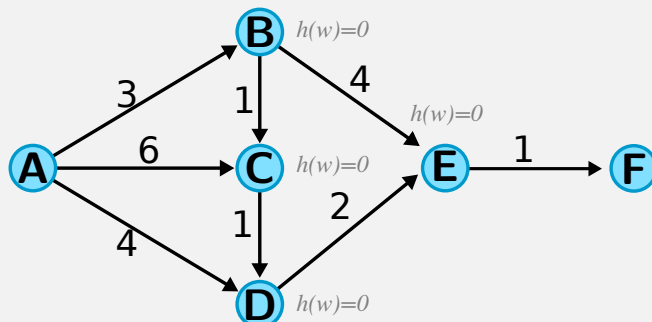


Abbildung 1B: Heuristiken

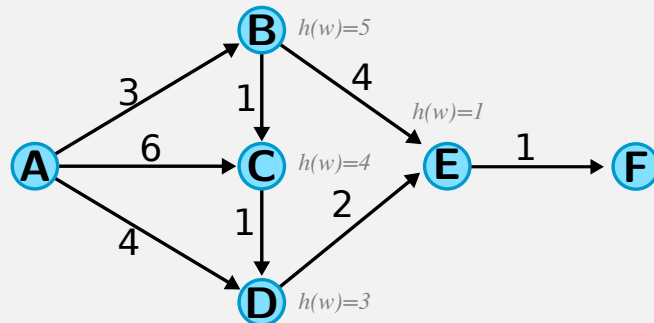
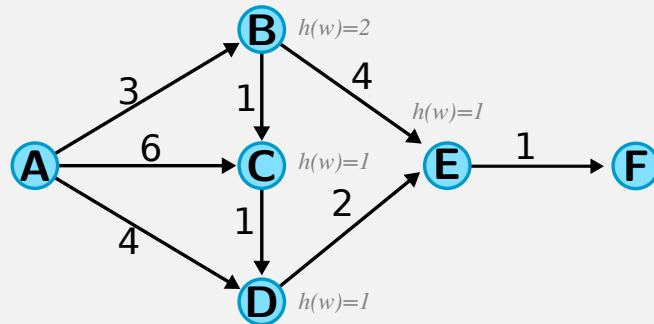
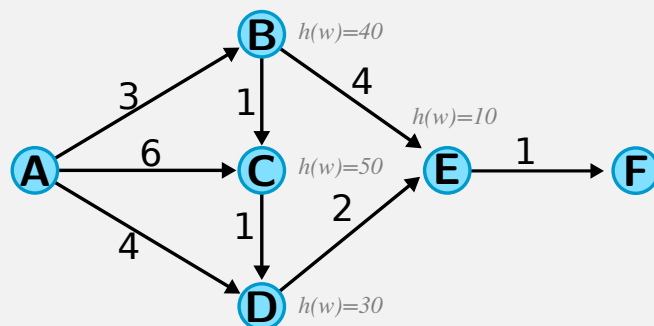
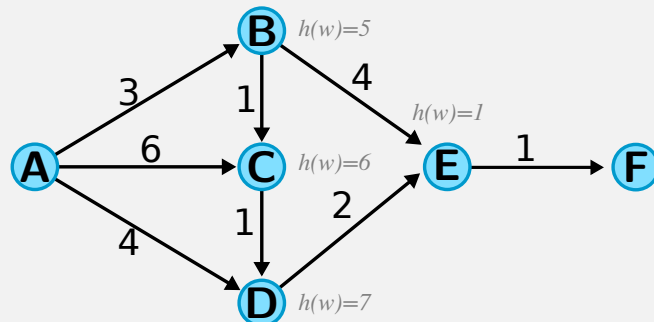


Abbildung 1C: Heuristiken

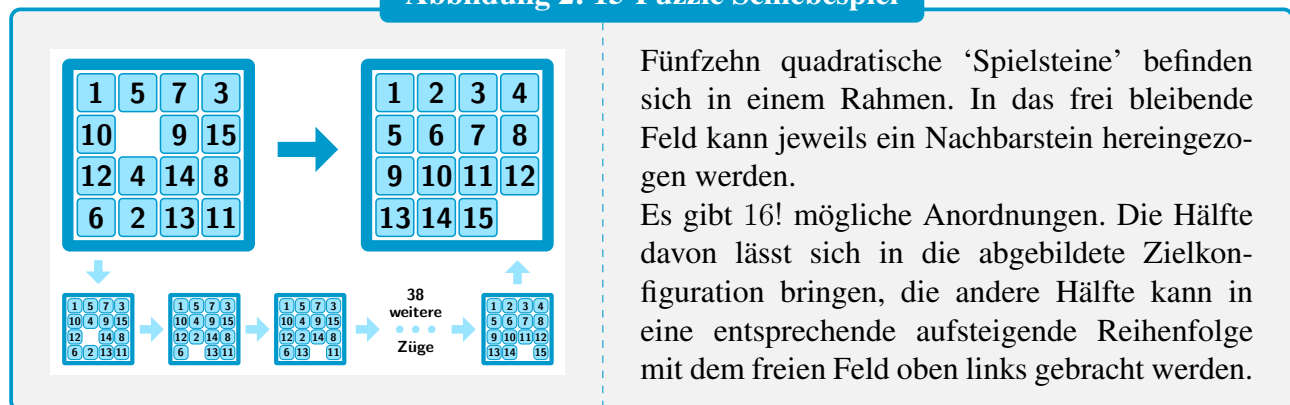


2.3 Was ist eine zulässige, was ist eine konsistente Heuristik?

Aufgabe 3: Lösungsprozedur für ein Schiebepuzzle (Hausaufgabe)

Das Schiebepuzzle auf Abb. 2 gibt es in unterschiedlichen Varianten. Üblicherweise handelt es sich um ein 4×4 Spielfeld mit 15 quadratischen Spielsteinen. Es bleibt also ein Feld frei. Die Spielsteine von den Nachbarfeldern können auf das freie Feld gezogen werden. Durch eine geeignete Zugfolge soll eine bestimmte Spielstellung erreicht werden.

Abbildung 2: 15-Puzzle Schiebispiel



In dieser Aufgabe soll eine kürzeste Zugfolge von einer gegebenen Spielposition zur Zielkonfiguration mit Hilfe des A*-Algorithmus bestimmt werden. Es wird im Baum der Teillösungen gesucht. Die Rückwärtskosten einer Teillösung ist die Anzahl der Züge von der Anfangsstellung dorthin, die Vorwärtskosten sind durch die in Aufgabe 3.2 definierte Heuristik definiert und die Gesamtkosten sind die Summe von Vorwärts- und Rückwärtskosten.

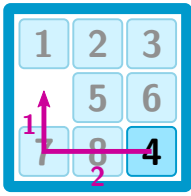
3.1 Klasse Board vervollständigen (15 Punkte)

Betrachten Sie die gegebenen Klassen `Position`, `Move` und die gegebenen Methoden der Klasse `Board`. Methoden für die üblichen Spieloperationen sind vorhanden. Es muss noch eine Methode implementiert werden, die für die Heuristik des A*-Algorithmus verwendet wird:

```
public int manhattan()
```


Die Methode berechnet als “Manhattan Heuristik” (basierend auf dem Manhattan Abstand, bzw. der *taxicab metric*) einen Abstand von der aktuellen Spielposition zu dem Zielzustand (siehe Abb. 2). Die Berechnung des Wertes der Heuristik wird in Abbildung 3 erklärt. Die dort gezeigten Beispiele sind als Dateien in der Vorgabe enthalten, siehe `main()` der Klasse `Board`.

Abbildung 3: Manhattan Heuristik



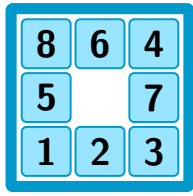
Manhattan
Heuristik = 3
4: $2+1 = 3$

board-manhattan0.txt



Manhattan
Heuristik = 6
3: $1+1 = 2$
4: $2+1 = 3$
5: $1+0 = 1$

board-manhattan1.txt



Manhattan
Heuristik = 18

board-manhattan2.txt

Der Manhattan Abstand zwischen zwei Punkten im zwei-dimensionalen Raum ist die Summe der Abstände in x - und in y -Richtung. Die **Manhattan Heuristik** für eine Position in dem Schiebepuzzle definiert man wie folgt. Für jeden Spielstein wird der Abstand in x und in y Richtung seiner Position zu seiner Zielposition addiert. Die Summe dieser Werte für alle Spielsteine ist der Wert der Manhattan-Metrik, siehe Abbildung. (Dabei wird der Abstand des freibleibenden Feldes zu seiner Zielposition nicht berücksichtigt.) Da jeder Zug die Manhattan Heuristik höchstens um 1 vermindern kann, ist die Manhattan Heuristik konsistent.

3.2 Implementation der Klasse `PartialSolution` (25 Punkte)

Die Klasse `PartialSolution` wird von der A*-Implementation benötigt. Als Teillösung wird der jeweilige Zustand des Spielfeldes, sowie die Zugfolge von der Ausgangssituation dorthin gespeichert. Damit A* die aussichtsreichste Teillösung aus der *priority queue* auswählen kann, muss `PartialSolution` die Schnittstelle `Comparable` basierend auf den Kosten einer Teillösung implementieren. Aus Effizienzgründen ist es sinnvoll, die Kosten in einer Objektvariable, z. B. `cost`, zu speichern. (Die Kosten wurden in der Einleitung definiert.)

Die Beschreibung der einzelnen Methoden können Sie der folgenden API und den Javadoc Kommentaren in der Klasse entnehmen.

| public class <code>PartialSolution</code> implements <code>Comparable<PartialSolution></code> | | |
|---|--|---|
| | <code>PartialSolution(Board board)</code> | Erzeugt eine leere Teillösung mit einer Kopie (<i>deep copy</i>) von <code>board</code> als Spielstellung und leerer Zugfolge. |
| | <code>PartialSolution(PartialSolution that)</code> | erzeugt eine unabhängige Kopie (<i>deep copy</i>). |
| | <code>doMove(Move move)</code> | führt <code>move</code> aus (und aktualisiert die Kosten) |
| boolean | <code>isSolution()</code> | Ist die Spielstellung die Zielstellung? |
| Iterable<Move> | <code>moveSequence()</code> | die Zugfolge von der Anfangsstellung zur aktuellen Stellung in chronologischer Reihenfolge von erstem zum letzten Zug |
| Iterable<Move> | <code>validMoves()</code> | alle möglichen Züge von der aktuellen Stellung, mit Ausnahme des Zuges, der den letzten Zug der Teillösung rückgängig machen würde (falls schon mindestens ein Zug ausgeführt wurde)* |
| int | <code>compareTo(PartialSolution that)</code> | vergleicht Teillösungen an Hand der Kosten |

Da die notwendige Funktionalität von der Klasse `Board` zur Verfügung gestellt wird, sind die Implementierungen dieser Klassen sehr kurz (maximal drei Zeilen). Diese Informationen dient zur Orientierung.

Hinweise:

- Beachten Sie, dass in dem Copy Konstruktor auch eine *unabhängige* Kopie der Zugsequenz

erzeugt werden muss.

- Beachten Sie die Funktionalität von `Board.validMoves(Move move)` bei der Implementation von `PartialSolution.validMoves()`.
- Das Vermeiden der inversen Züge in `PartialSolution.validMoves()` ist nicht notwendig für das eigentliche Funktionieren der A*-Suche. Aber ohne diese Maßnahme wird der Suchraum bei Beispielen mit langen Zugfolgen so groß, dass die Suche nicht effizient durchgeführt werden kann.
- Die `main()` Methode der Klasse `PartialSolution` lädt das Beispiel `board-3x3-twosteps.txt` und führt die Lösungsschritte durch. Die erwartete Ausgabe ist in der Datei `board-3x3-twosteps-ExpectedOutput.txt` gespeichert. Der Code kann erweitert werden, um die Methoden `isSolution()` und `validMoves()` zu testen.

3.3 Implementation der Klasse `AStar15Puzzle` (60 Punkte)

Implementieren Sie die Klasse `AStar15Puzzle`, die die Lösung eines beliebigen $nx \times ny$ Schiebepuzzles mit Hilfe des A*-Algorithmus findet. Dabei soll als Heuristik der *Manhattan*-Abstand zur Zielstellung verwendet werden.

Konkret soll in der Klasse die Methode

```
public static PartialSolution solveByAStar(Board board)
```

implementiert werden. Für lösbare Spielstellungen `board` soll ein `PartialSolution` Objekt zurückgegeben werden, das eine kürzeste Zugfolge beinhaltet und mit der Methode `moveSequence()` abgerufen werden kann. (Für unlösbare Spielstellungen muss das Programm nicht terminieren. Eine effiziente Methode mit der dies verhindert werden kann, ist Bestandteil des nächsten Aufgabenblattes.)

Eine Implementierung (im normalen Programmierstandard), die den in der Vorlesung präsentierten Ansatz umsetzt, kommt mit 15 Zeilen aus. Dies ist keine Aufforderung dazu, eine möglichst kurze Methode zu schreiben, sondern soll nur als Orientierung dienen. Falls Sie eine Idee haben, die sehr viel mehr Code und Hilfsfunktionen erfordert, schauen Sie nochmal in die Vorlesung und überlegen Sie, wie Sie die Aufgabe anders lösen können.

In dem Ordner `samples` liegen drei 3x3 Ausgangssituationen zum Testen bereit (`board-3x3-*.txt`). In der `main()` Methode der Klasse `AStar15Puzzle` ist entsprechender Code zur Ausführung. Lösungen zu den drei Beispielen sind in den Dateien `*-ExpectedOutput.txt`. Da die kürzeste Zugfolge nicht unbedingt eindeutig ist, kann es auch andere korrekte Lösungen geben. Die Länge der Zugfolge ist eindeutig.

Tipps, Hinweise und Bemerkungen:

- Nachdem aus dem übergebenen `board` eine 'leere' Teillösung als Anfang der A*-Suche erstellt wurde, braucht hier nicht mehr direkt mit `Board` Objekten gearbeitet zu werden. Es können die entsprechenden Methoden der Klasse `PartialSolution` genutzt werden.
- Wenn Sie ein `PartialSolution` Objekt verändern und in die *priority queue* einfügen, beachten Sie, dass Sie zunächst eine *deep copy* erzeugen. Sonst verändern Sie ggf. andere Objekte in der Queue. Aus diesem Grund gibt es in der Klasse `PartialSolution` einen Copy Konstruktor.

- Die schwierigsten Stellungen im 3×3 Spiel erfordern 31 Züge. Diese Lösung kann mit dem A*-Algorithmus sehr schnell gefunden werden.
- Die schwierigsten Stellungen im 4×4 Spiel erfordern 80 Züge. Sie erfordern mit dieser einfachen A* Implementierung zu viel Rechenaufwand.
- Für das 5×5 Spiel ist noch nicht bekannt, wie viele Züge die schwierigsten Stellungen erfordern.

Was Sie nach diesem Blatt wissen sollten:

- Was sind heuristische Algorithmen? Was sind Heuristiken?
- Was ist eine zulässige Heuristik? Was ist eine konsistente Heuristik?
- Wie funktioniert der A* Algorithmus? Wie wird dabei relaxiert?
- Was sind approximative Algorithmen? Was ist ein ρ - approximativer Algorithmus?