TU Berlin, Fakultät IV
Blankertz/Röhr/Stahl
MAR 4-3 und MAR 5-1

Aufgabenblatt 11*

Effiziente Suche mit Hash Sets

Bonus Blatt

Alle 100 Punkte dieses Blattes sind Bonuspunkte. Die Regeln für diese Bonuspunkte sind dieselbe, wie für die Bonuspunkte, die bisher erlangt werden konnten. Es werden die Punkte (reguläre und Bonus-Punkte) aller Blätter von 0 bis 11 addiert, durch 2.000 geteilt und das Ergebnis auf maximal 50 begrenzt (d.h. man kann nicht mehr als 100% der regulären Punkte im Aufgabenteil bekommen). Dies sind die Modulpunkte aus den Aufgabenblättern. Dann werden die Modulpunkte aus der Klausur addiert und ganz am Ende wird gerundet.

Abgabe (bis 16.07.2019 19:59 Uhr)

Die folgenden Dateien werden für die Bepunktung berücksichtigt:

`Blatt11/src/Position.java`	Aufgabe 1.2
`Blatt11/src/Board.java`	Aufgabe 1.3
`Blatt11/src/PartialSolution.java`	sowie
`Blatt11/src/RicochetRobots.java`	Aufgabe 1.4
`Blatt11-Extra/src/*`	Aufgabe 1.5 (ohne Punkte)

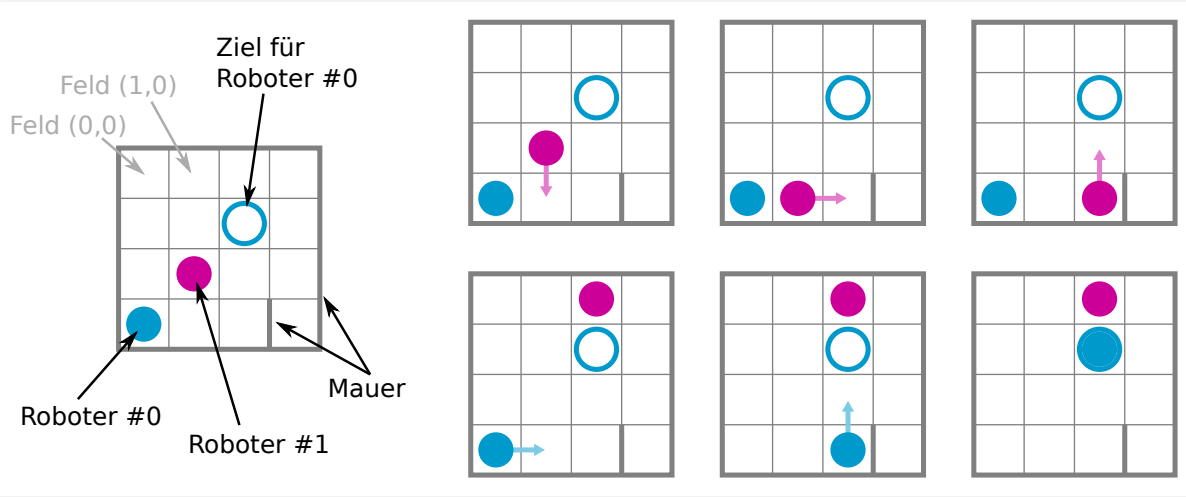
Als Abgabe wird jeweils nur die letzte Version im git gewertet.

Aufgabe 1: Lösungsprozedur für ein Knobelspiel (Hausaufgabe)

Rasende Roboter (*Ricochet Robots*) ist ein Brettspiel für mehrere Personen von Alex Randolph, siehe https://de.wikipedia.org/wiki/Rasende_Roboter. Es kann auch alleine als Knobelspiel gespielt werden, auch auf anderen Brettern, als die Brettspielvariante. Es gibt viele Apps für Mobilgeräte, die dieses Spiel entweder direkt implementieren, oder davon inspiriert sind. Um neue Level für ein solches Spiel zu erzeugen, benötigt man einen sehr schnellen *Solver*. Für dieses Aufgabenblatt ist die Grundfunktionalität des Spieles gegeben. Alle spezifischen Bestandteile für eine effiziente Lösungsprozedur (und die Prozedure selber) sind zu implementieren.

Während das Original Brettspiel 16×16 Felder besitzt, ist bei uns die Größe des Spielfeldes variable und wird meist deutlich kleiner sein. Ebenso variiert die Anzahl der Roboter. Das Spiel wird in Abbildung 1 erkläutert.

Abbildung 1: Ricochet Robots



Das rechteckige Spielfeld besteht aus quadratischen Feldern, die durch Mauern getrennt sein können. Es gibt immer eine äußere Berandung durch Mauern. Bei einem Spielzug wird einer der Roboter in eine Richtung gezogen, und zwar soweit, bis er durch eine Mauer oder einen anderen Roboter gestoppt wird. Auch wenn dabei über mehrere Spielfelder hinweggezogen wird, zählt dies als *ein* Spielzug. Ein Bremsen auf freier Strecke ist nicht möglich.
Ein Feld ist als Zielfeld für einen der Roboter markiert. Ziel des Spieles ist es, eine möglichst kurze Zugfolge zu findet, durch die dieser Roboter auf dem Zielfeld zum Stehen kommt. Ein Roboter rutscht allerdings über das Zielfeld hinweg, falls er nicht durch eine Mauer oder einen anderen Roboter gestoppt wird. (Beispieldatei: `rrBoard-sample01.txt`)

In dieser Aufgabe soll eine effizient Lösungsprozedur für Ausgangssituationen von Ricochet Robots mit der Hilfe von *Hash Sets* implementiert werden. Anders als bei dem 15-Puzzle des letzten Aufgabenblattes ist bei Ricochet Robots nicht klar, wie eine konsistente Heuristik formuliert werden könnte. Es gibt nicht einmal einen Ansatz für eine (sinnvolle) zulässige Heuristik, die als Schranke für ein *Branch-and-Bound* Verfahren dienen könnte. Somit gibt es zunächst nur den Ansatz der Breitensuche. Bei diesem Spiel ergibt sich dabei das Problem, dass man durch viele unterschiedliche Spielfolgen zu derselben Spielsituation gelangt. Wenn dies nicht berücksichtigt wird, wächst der Suchbaum der Breitensuche (und in der Implementation konkret die Warteschlange) so schnell an, dass schon relativ einfache Problem nicht in akzeptabler Rechenzeit gelöst werden können.

Daher prüft man nach der Ausführung eines Zuges, ob die neue Spielsituation schon vorher erreicht wurde und verwirft sie in diesem Fall. Um die Vergleich einer Spielsituation mit allen früheren Spielsituationen effizient zu gestalten, werden *Hash Sets* für die Klasse `Board` verwendet.

1.1 Methoden für Spiellogik studieren (0 Punkte)

Betrachten Sie die gegebenen Klassen `Position`, `Move`, `Robot` und `Board`, die die Spiellogik von Ricochet Robots implementieren. Die `main()` von `Board` enthält Beispielcode, der direkt ausgeführt werden kann.

Die variablen Elemente bei einem einzelnen Spiel sind nur die Positionen der Roboter. Daher wird bei diesem Spiel die Spielsituation nicht über ein zwei-dimensionales Array mit den Inhalten jedes Spielfeldes gespeichert, sondern als eine Liste der Positionen der Roboter.

Um einen Hashcode für die Klasse `Board` zu implementieren, benötigt man also einen Hashcode für die Klasse `Position`. Fangen wird damit an:

1.2 Klasse `Position` vervollständigen (10 Punkte)

Implementieren Sie die Methoden

```
@Override
public boolean equals(Object o)
@Override
public int hashCode()
```

in der Klasse `Position`. Die Hashcodes von `Position` Objekten sollten für alle Positionen auf einem 20×20 Brett unterschiedlich sein.

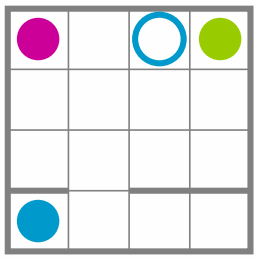
1.3 Klasse `Board` vervollständigen (15 Punkte)

Implementieren Sie die Methoden

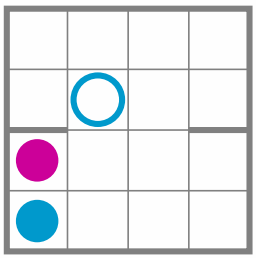
```
@Override
public boolean equals(Object o)
@Override
public int hashCode()
```

in der Klasse `Board`. Sie brauchen in beiden Methoden nur diejenige Information zu berücksichtigen, die sich während eines Spielverlaufes (bzw. während der Lösung einer Ausgangsstellung) ändert.

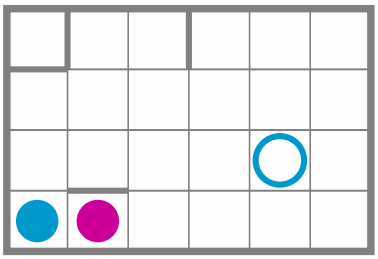
Abbildung 2: Drei Knobelaufgaben



`rrBoard-sample01.txt`



`rrBoard-sample02.txt`



`rrBoard-sample03.txt`

Hier sind drei weitere *Ricochet Robots* Aufgaben. Die Lösungen für die ersten beiden sind weiter unten abgebildet. Die dritte bleibt zum Knobeln. Oder wartet auf Ihre Implementierung des Solvers!
Alle drei Aufgaben sind in den Vorgaben als Dateien im `samples` Verzeichnis gegeben. Zu den ersten beiden (sowie dem Spiel aus Abb. 1) sind die Lösungen des Solvers ebenfalls mit dabei. Die `main()` Methode der Klasse `RicochetRobots` enthält Code zum Laden der Beispiele, Aufruf des Solvers und Ausgeben der Lösung.

1.4 Implementation der Klasse `PartialSolution` und `RicochetRobots` (75 Punkte)

Implementieren Sie in der Klasse `PartialSolution` die notwendige Funktionalität für die Lösungssuche bereitstellt. Die Implementierung ist eng verknüpft, mit der Implementierung der Lösungssuche in der Klasse `RicochetRobots` (siehe nächster Aufgabenteil). Sie sollten also das Konzept für beiden Klassen gemeinsam planen.

Hier ist die Aufgabe freier gestaltet, als auf dem letztem Aufgabenblatt. Sie können also selbst entscheiden, wieviel Funktionalität Sie in die Klassen `PartialSolution` verlegen möchten. Im Minimum muss `ParitialSolution` ein Spielstellung `Board` und die Zugsequenz dorthin speichern. Andere Funktionalität wie *isFinal* oder *doMove* kann direkt über `Board` in der Methode für die Breitensuche abgerufen werden, oder wie auf dem letzten Aufgabenblatt in `PartialSolution` integriert werden.

Die einzige Vorgabe dabei ist, dass die Methode

```
public LinkedList<Move> moveSequence()
```

implementiert sein muss. In den Tests wird nur geprüft, ob die Länge des Rückgabewertes mit der minimalen Anzahl von Spielzüge zur Lösung übereinstimmt, für das Objekt, das die als nächstes beschriebene Methode `bfsWithHashing()` zurückgibt.

Implementieren Sie in der Klasse `RicochetRobots` die Methode

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

um eine kürzeste Lösung (geringste Anzahl von Spielzügen) einer beliebigen Spielstellung (Brettgrößen bis 20×20 und bis zu 10 Robotern) zu bestimmen. Um die geforderte Effizienz zu erreichen soll bei der Lösungssuche mittels eines *Hash Sets* verhindert werden, dass dieselbe Spielsituation mehrfach untersucht wird. Ohne diese Optimierung explodiert der Lösungsraum und es wird fast kein Test ohne *timeout* bestanden.

Es ist sinnvoll zu Testen, ob das das Aussondern doppelter Spielstellungen mittels Hashing greift.

Hinweis:

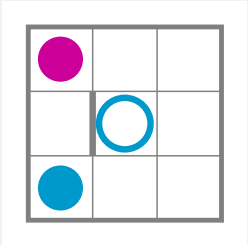
- Die größere Freiheit bei der Wahl der Implementierung bringt es mit sich, dass keine einzelnen Aspekte in unseren Tests überprüft werden können. Sie beruhen also im Wesentlichen auf einer Überprüfung der Gesamtfunktionalität. Es gibt also keine speziellen Tests für `PartialSolution`.
- Eine Implementierung von `bfsWithHashing()` kommt mit 20 Zeilen (im normalen Programmierstandard) aus, selbst wenn `PartialSolution` nur die Minimalfunktionalität zur Verfügung stellt. Dies ist keine Aufforderung dazu, eine möglichste 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 die die Aufgabe anders lösen können.

1.5 Lust auf mehr? (0 Punkte, aber ...)

Wer eine größere Herausforderung sucht, kann sich an der folgenden Erweiterung versuchen: Finden Sie unter den kürzesten Lösungen diejenige, bei der es am wenigsten Wechsel zwischen

Robotern in der Zugfolge gibt, siehe Abbildung 3. Eine (effiziente) Implementierung ist eine größere Herausforderung, als es auf den ersten Blick scheinen mag. Sie ist aber mit den Techniken aus der Vorlesung zu bewältigen und nicht viel länger als die Aufgabe ohne den Zusatz.

Abbildung 3: Lösungen mit möglichst wenig Wechseln



Es gibt oft unterschiedliche Lösungen mit minimaler Zugzahl. Hier benötigen die Lösungen `R0: E, R1: E, R0: NW` und `R1: E, R0: ENW` beide vier Züge. Die erste hat 2 die zweite 1 Wechsel des ziehenden Roboters. Bei der Zusatzaufgabe geht es darum unter den kürzesten Lösungen diejenige zu finden, die die wenigsten Wechsel hat.

Die Laufzeit wird durch die zusätzliche Anforderung etwas schlechter. Um die Punkte für Aufgabe 1.4 nicht zu gefährden, sollte die Zusatzaufgabe am besten getrennt in einem separater Verzeichnis (z. B. '`Blatt11-Extra`') gespeichert werden. Die Zusatzaufgabe wird per Hand durchgesehen, daher Abgaben bitte per E-Mail an ›`benjamin.blankertz@tu-berlin.de`‹ melden und einen Link auf das Verzeichnis in Gitlab senden (bis 23.07.). Nur in dem Fall, dass es überraschend viele Abgaben für die Zusatzaufgabe gibt, werden automatische Tests eingesetzt.

Für die Bewältigung dieser Herausforderung gibt es keine weiteren Punkte, aber einen kleinen Preis.

Aufgabe 2: Hashing

2.1 Wie funktioniert Hashing und was sind die Vorteile? Was sind Kriterien für eine gute Hashfunktion? Wo wird Hashing angewandt?

2.2 Sei k die Stelle eines Buchstaben im Alphabet. Betrachten Sie nun Hashing mit folgender Hashfunktion $h(k) = 11k \mod 7$. Fügen Sie damit das Wort ALGODAT in die Hashtabelle ein. Verwenden Sie dabei zur Kollisionsauflösung lineares Sondieren mit Inkrement 1 (d.h. $s(n) = n$).

0	1	2	3	4	5	6

Aufgabe 3: Wiederholung

3.1 Geben Sie möglichst genaue Wachstumsordnung für den folgenden Code an. Begründen Sie Ihre Herleitung der Laufzeit am Code. Gehen Sie davon aus, dass der Graph N Knoten enthält und E Kanten. solved ist hierbei ein `boolean` Array der Länge N.

```
public void mystery(Graph G, int v, boolean[] solved) {
    solved[v] = true;
    for (int w : G.adj(v)) {
        if (!solved[w]) {
            mystery(G, w, solved);
        }
    }
}
```

Was macht der Code? Kennen Sie diesen Algorithmus?

3.2 Erklären Sie an der Karte in Abbildung 4 (https://www.visitberlin.de/de/blog/berlin-bade-karte), welche unterscheidlichen Konzepte von Graphen Sie kennengelernt haben, welche Algorithmen auf diesen Graphen Sie kennen und welche Arten von Problemen diese lösen.

Abbildung 4: Berlin Bade-Karte

3.3 **(Optional)** Erstellen sie für folgende Kleidungsstücke Knoten und Kanten in einem gerichteten Graphen, wobei eine Kante bedeutet, dass der Startknoten vor dem Endknoten angezogen werden muss: Unterhemd (0), Socken (1), Krawatte (2), Gürtel (3), Schuhe (4), Hose (5), Sakko (6), Unterhose (7).

Geben sie anschließend eine Topologische Sortierung und eine Beispielreihenfolge für das Anziehen an, indem sie den folgenden Code per Hand simulieren. Wie muss dafür die Methode `public void mystery` angepasst werden? Für welche Graphen funktioniert der Algorithmus?

```
    public void topologicalSortClothing(Graph g){
        Stack<Integer> stack = new Stack<Integer>();
        boolean visited[] = new boolean[G.V()];
        for (int i = 0; i < G.V(); i++){
            visited[i] = false;
        }
        for (int i = 0; i < V; i++){
            if (!visited[i]){
                mystery(G, i, visited, stack);
            }
        }
        while (!stack.isEmpty()){
            System.out.print(stack.pop() + " ");
        }
    }
```

3.4 **(Optional)** Gehen Sie die Handsimulation der Alpha-Beta-Suche in Abbildung 5 schrittweise durch. Markieren Sie ggf. α- und β- Cutoffs.

Abbildung 5: Alpha-Beta Suche- Handsimulation

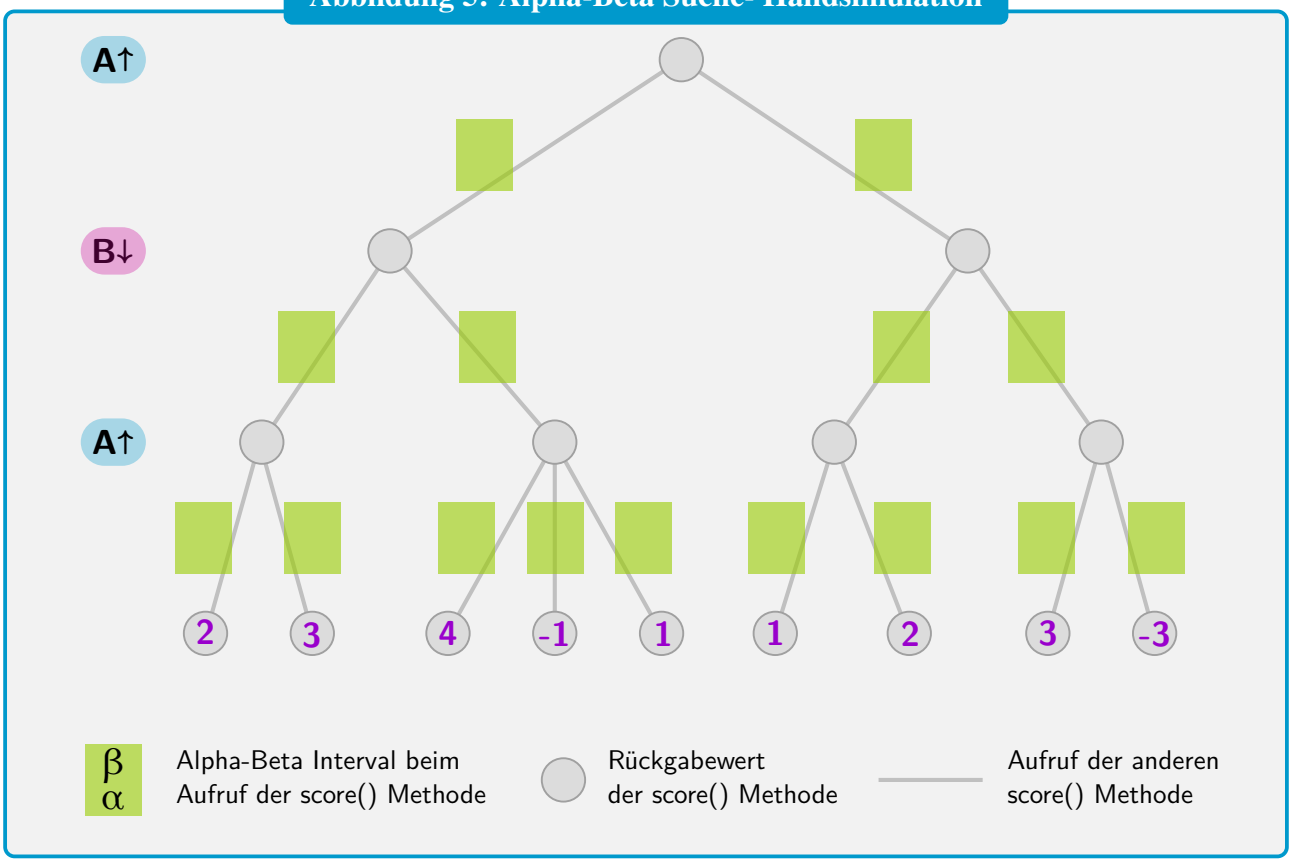


Abbildung 6: Lösungen der Aufgaben aus Abb. 2