42. BWInf Runde 2 - Aufgabe 1

# Aufgabenstellung

Die Aufgabe wird in zwei Richtungen erweitert:

* Alternativ zum fest vorgegebenen Zielfeld wird auch überprüft, welche Strategien und Ergebnisse möglich sind, wenn das Laub auf einem beliebigen Feld des Innenhofes (also kein Rand- oder Eckfeld) gesammelt wird.
* Anstatt eine Strategie heuristisch vorzudefinieren und damit fest vorzugeben, wird der Baum möglicher Nachfolgezustände **iterativ** durchsucht. Die Strategie ergibt sich dann aus der ersten Operation des jeweils besten Unterbaums. Die Aufgabe wird also als **Optimierungsproblem** interpretiert.

# Lösungsidee

Bei genauerer Betrachtung sieht man, dass die Zustände nicht als Baum, sondern als **Graph** geordnet sind. Das liegt daran, dass zwei verschiedene Zustände lediglich eine **Spiegelung** voneinander sein können. Es reicht dann aus, wenn nur einer der beiden weiter untersucht wird. Dies wird dadurch sichergestellt, dass solch **äquivalente** Zustände durch einen Lookup über ihren **normalisierten Repräsentanten** erkannt werden. Als Folge davon lässt sich je nach Schulhofform (quadratisch vs. rechteckig) und dem Zielfeld (spiegelsymmetrisch für alle Spiegelungen) die Zahl der einzigartigen Nachfolgezustände um den Faktor 8 (23) verringern. Dies ändert auch die Komplexitätsklasse für von *O(fn*) zu *O((f/8)n)*, wobei *f* die Zahl der möglichen Operationen pro Zustand und *n* die Zahl der erlaubten Schritte (also Blase-Operationen) sind. In der Implementierung wird von **Knoten** statt Zuständen gesprochen, um die Interpretation als Graphen zu verdeutlichen.

Die Komplexitätsklasse lässt sich weiter reduzieren, indem auch nur solche Operationen in Betracht gezogen werden, bei denen sich die Laubsituation verändern kann. Zum Beispiel gibt es auf einem 5∗5 Schulhof an sich 25∗4=100 Möglichkeiten zu blasen, da es 4 Richtungen gibt. Da aber das Blasen gegen den Rand aus der Entfernung 0 oder 1 zu keiner Änderung führt (Detaildefinitionen siehe im Unit Test und in der Beispielliste (siehe unten)), können für jede Richtung 2∗5 Blaseorte ignoriert werden. Damit gibt es nur (100−4∗2∗5)=60 **potenziell verändernde** Blase-Operationen, die zu berücksichtigen sind. Zusammen mit der Äquivalenzbetrachtung ergibt sich also für einen 5∗5 Schulhof eine Komplexität von *O(7.5n)* statt *O(100n)*.

Dies ist aber natürlich immer noch viel zu viel, um die möglichen Zugfolgen in die Tiefe untersuchen zu können. Entscheidend ist also die **Priorisierung** bei der Durchforstung des Unterbaums, um bei gegebenem Budget die beste Zugfolge möglichst auch zu finden. Dieser Ansatz ähnelt der **programmatischen Künstlichen Intelligenz (KI)** bei Spielen wie Schach (siehe [Deep Blue](https://de.wikipedia.org/wiki/Deep_Blue)). Für diese Priorisierung wird eine andersartige Strategie benötigt. Im Vergleich zur Aufgabenstellen legt sie nur fest, wie erfolgversprechend einzelne Zustände erscheinen. Ob sie das aber wirklich sind, zeigt nur die Optimierung und Simulation. Es ist also eine Strategie zur Priorisierung der Nachfolgersuche - die tatsächliche "Zugreihenfolge" ergibt sich aber durch den am Ende gefundenen optimalen Pfad. Im Gegensatz zu Spielen mit mehreren Spielern (wie Schach) beinhaltet diese Zugreihenfolge auch die komplette Strategie im Sinne der Aufgabe, da die Reihenfolge durch keine Züge eines Gegenspielers unterbrochen werden können.

# Umsetzung

Das Programm ist mit Java 11 geschrieben. Im Folgenden wird zunächst das Klassendiagramm als Übersicht gezeigt. Im Anschluss werden die Java-Klassen/Interfaces werden nach Paketen aufgeteilt genauer betrachtet. Dabei wird nur auf Aspekte eingegangen, die im Klassendiagramm nicht klar erkennbar sind.

## Klassendiagramm

Um die Abbildung zu vereinfachen, ist im UML Klassendiagramm Folgendes weggelassen:

* Hole-Methoden: durch die Attributauflistung nicht zwingend erforderlich
* Override von Standard-Methoden (equals, hashCode, toString): im Modell standardmäßig durchgeführt aber von der Optimierung nicht benötigt
* Die Unterklassen von Lauf sind nur schematisch angedeutet durch XYZ-Suche. Die einzelnen Läufe wie zum Beispiel Tiefensuche, Gierigsuche und Annealingsuche verfahren nach demselben Schema wie XYZ-Suche.
* Konstruktoren sind nicht aufgelistet, es sei denn, es gibt dabei Besonderheiten (bei Schulhof der Fall).

A screenshot of a computer

Description automatically generatedAbbildung 1: Klassendiagramm

## Modell

### Feld

Verwendung:

* Falls ein Zielfeld vorgegeben ist, wird dafür ein Objekt dieser Klasse benutzt.
* Dasselbe gilt auch für Rückgabeparameter der Spiegelungen (in Schulhof).

### Schulhof

Objektattribute:

* Kalkuliert für jedes neue Schulhof 4 Attribute (repraesentant => zur Normalisierung des Schulhofes, maxLaub, maxFeldX, maxFeldY => die Daten des Feldes, auf dem am meisten Laub vorhanden ist

Konstruktoren:

1. Schulhof mit der Breite, Höhe und dem Zielfeld (für alle zugänglich)
2. Schulhof wie 1 und einer Liste der Felder auf dem Schulhof, da diese für das eigene felder-Attribut geklont wird, damit keine Änderungen am Schulhof von außen möglich sind.
3. Schulhof mit Zielfeld und einer Matrix von Feldern => diese wird direkt übernommen, wegen der package visibility ist sichergestellt, dass kein Externer die Matrix so konstruieren und danach die Felder ändern kann.

Methoden:

* equals
* Gibt wahr zurück, wenn other äquivalent ist, d.h.
* dieselben Dimensionen und dasselbe Laubmuster haben
* (oder das für Spiegelungen (horizontal, vertikal, diagonal) zutrifft).
* Das ist genau dann der Fall, wenn die Repräsentanten gleich sind.
* hashCode
* Muss für äquivalente Schulhöfe dasselbe Ergebnis liefern. Das ist dann der Fall, wenn die Repräsentanten gleich sind. Das berücksichtigt auch das Zielfeld, wenn vorhanden.
* berechneAttribute
* wird immer am Ende des Konstruktors aufgerufen
* berechnet alle abhängigen Attribute, um spätere hole-Aufrufe effizient beantworten zu können
* berechneRepraesentant
* Alle Schulhöfe mit derselben Normalisierung werden äquivalent genannt (also sie ergeben sich aus einer Serie von Spiegelungen).
* Normalisierung bedeutet, einen Repräsentanten des Schulhofes zu bekommen, der sich, auch wenn der Schulhof gespiegelt wird, genau gleichbleiben würde.
* Wenn ein Zielfeld vorgegeben ist, wird es auch entsprechend gespiegelt und muss nach der Normalisierung gleich sein, damit Schulhöfe äquivalent sind.
* Alle Spiegelachsen in allen Kombinationen werden ausprobiert, sofern der Schulhof dies möglich macht (bei rechteckigem Schulhof ist keine diagonale Spiegelung möglich).
* Ruft die Methode spiegele (siehe unten) auf, um den Repräsentanten berechnen zu können.
* Spiegele
* Gibt ein Feld zurück – je nachdem, ob istUmgekehrt wahr/falsch ist
* Falls istUmgekehrt wahr ist, wird die „eigentliche“ Spiegelungsreihenfolge (vertikal, horizontal, diagonal) umgedreht, damit eine Schulhofsituation, bei der „nur“ gespiegelt wurde, der gleiche Repräsentant berechnet wird.

### BlaseOp

Objektattribute:

* Koordinaten x, y um die „Blase-Stelle“ und dx, dy um die Blasrichtung bestimmen zu können

Konstruktor:

* Ist nur public, um direkte Tests zu ermöglichen. Die anderen Klassen benutzten die statische Methode holeVeraenderndeOps (siehe unten), um die Blase Operationen zu erhalten.

Methoden:

* tue
* Setzt die Idee von Blasen um (vorgegebene & selbst hinzugefügte Regeln (siehe Lösungsidee)).
* Der Parameter ist ein Schulhof, von dem die Felder direkt kopiert werden, da der Schulhof später gebraucht wird, um zu überprüfen, ob bestimmte Felder existieren.
* Gibt ein neuer Schulhof mit den „neuen Feldern“ (sie sind nicht zwangsläufig anders als die im „alten“ Schulhof) und dem vorgegeben Zielfeld zurück.
* *(statisch)* holeVeraenderndeOps
* Gibt eine Liste von Blase Operationen zurück
* Dafür werden auf jedem Feld des Schulhofs alle 4 Blase Richtungen ausprobiert. Anschließend wird die Blase-Operation (BlaseOp) auf den selbsterstellten Schulhof (mit den Parametern Breite & Höhe) ausgeführt.
* Eine BlaseOp ist nur dann potentiell verändernd, wenn sie die Chance hat, die Laubverteilung zu verändern.
* Dafür ist notwendig, dass die Blase-Operation bei der initialen Verteilung eine Veränderung hervorruft. Daher wird die Liste anhand mehrerer Op-Simulationen erstellt.

## Optimierung

### (abstrakt) Optimierer

*(protected)* Objektattribute:

* Liste von den möglichen Blase-Operationen (nur die, die danach eine Veränderung auf dem Schulhof verursachen)
* einer Strategie (die, die beim Aufrufen mitgegeben wird)
* gesehen vom Typ Set<String>:
  + Gesehen hält Repräsentanten (String) und nicht Schulhof (mehrere Attribute inkl. Felder), um die Größe des Sets klein zu halten.
  + Nur die Repräsentanten untereinander nicht äquivalenter Schulhöfe werde hier abgelegt.
  + Das reicht aus und ist sogar wünschenswert, da äquivalente Schulhöfe nur eine bloße Spiegelung sind und nicht weiter betrachtet werden müssen.
* und weitere (siehe Klasse Optimierer)
* Diese sind alle *protected*, weil niemand außer der Unterklassen von Optimierer Zugriff auf diese haben soll.

Konstruktor:

* Der Parameter zielfeld hat den Wert null, wenn kein Zielfeld fest vorgegeben ist, sondern nur ein beliebiges Innenhoffeld den Maximalwert haben darf.
* Die „Budget“ Variablen sind optionale Beschränkungen, diese können später per setze-Methode noch angegeben werden. Im Konstruktor wird es zuerst auf *default* gesetzt, also keine Beschränkung -> unendlich (Integer.MAX\_VALUE)

Methoden:

* *(abstrakt)* tue
  + Diese Methode können die Unterklassen von Optimierer selbst implementieren, hier ist sie lediglich mit einer Signatur festgelegt.

### Knoten

Implementiert *Comparable* da compareTo diese Implementation braucht, um zwei Knoten vergleichen zu können.

Objektattribute:

* Ein Knoten enthält immer folgende drei Informationen: sein eigener Schulhof, sein Knoten-Vorgänger und die letzte Blase-Operation (BlaseOp), um zu seinem Schulhof zu gelangen.
* Außerdem auch den Schritt, also an der wievielten „Verzweigung“ sich der betrachtete Knoten befindet und die Priorität. Jeder Knoten besitzt eine Priorität und in Funktion auf diese wird er früher oder später aufgerufen (oder gar nicht).

Konstruktoren:

* Es gibt zwei Konstruktoren, der Erste, wenn es sich um den ersten Knoten handelt, dieser enthält also nur die Informationen seines Schulhofs, sein Knoten-Vorgänger existiert nicht, ist also null. Folglich hat der Knoten auch keine Information zu der vorherigen BlaseOp.
* Der Zweite ist für den anderen Fall, also wenn es sich nicht um den ersten Knoten handelt. Dann sind alle drei Werte gegeben.

Methoden:

* setzePrioritaet
* Wird nur einmal gesetzt und nicht bei jedem compareTo neu berechnet, damit:

1. Knoten die Strategie nicht kennen muss
2. der Vergleich effizient ist (manche Strategien könnten aufwändiger zu berechnen sein)

* diese Methode ist nur public, um darauf Tests machen zu können
* compareTo
  + Gibt die Differenz zwischen 2 Knoten zurück, um ausrechnen zu können, welcher Knoten eine höhere Priorität hat.
  + *Beispiel:* Soll Knoten1 (K1) vor Knoten2 (K2) betrachtet werden, so muss K1 < K2 sein. Das bedeutet für ein Knoten KP+ mit höherer Priorität im Vergleich zu einem Knoten KP- mit niedriger Priorität: KP+ < KP-
* Der Vergleich gibt also die gewünschte zeitliche Abfolge der Bearbeitung zurück und nicht den aufsteigenden Prioritäten.

### GierigOptimierer

* Ist eine Unterklasse von Optimierer

Methoden:

* Tue
* Speichert Werte für Statistiken (zum Beispiel wie oft ein Knoten per Zufall ausgewählt wurde) => siehe Output-Dateien
* Ruft tueNaechsten auf, solange die Prioritäts-Todo-Liste (prioTodo) nicht leer ist und das Budget der Blase-Operationen noch nicht erreicht wurde. In beiden Fällen macht es kein Sinn den nächsten Knoten zu bearbeiten, denn
  + wenn prioTodo leer ist und trotzdem Knoten bearbeitet werden, die von der Zufalls-Liste „kommen“, dann handelt es sich nicht um eine Optimierung mit einer Strategie, sondern um puren Zufall
  + das Budget der Blase-Operationen wird benutzt, um die Optimierungen vergleichen zu können (nach Leistung, also wie viel Laub auf einem möglichen Feld ist)
* tueNaechsten
* entfernt einen Knoten von prioTodo
* bereitet den nächsten Knoten vor
* überprüft, ob die budgetSchritttiefe, also wie „tief“/“weit“ die Knoten reichen dürfen, überschritten wurde
* wenn dies noch nicht der Fall ist, wird dem nächsten Knoten seine Priorität ausgerechnet (siehe Strategie) und zugewiesen
* erst dann wird der nächste Knoten der Todo-Liste hinzugefügt

### AnnealingOptimierer

* Ist eine Unterklasse von Optimierer
* Idee: Um direkten Indexzugriff auf Knoten der todo-Liste zu erhalten, werden zwei Listen parallel gehalten (Heap für Priorität und ArrayList für Indexzugriff). Außerdem gibt es einen Lookup für schon bearbeitete Knoten, um sie nicht aus der jeweilig anderen Liste löschen zu müssen (würde O(n) kosten).

Objektattribute:

* + zufallProb und zufallFaktor – beide vom Typ double
* zufallProb ist die anfängliche Wahrscheinlichkeit, dass nächster zu bearbeitende Knoten zufällig gewählt wird
* zufallFaktor ist < 1

Konstruktor:

* Berechnet den zufallFaktor mithilfe des Parameters zufallHalbiertNach:
* Dieser gibt an, nach wie vielen Bearbeitungen von Knoten die Wahrscheinlichkeit, dass nächster zu bearbeitender Knoten zufällig gewählt wird, nur noch halb so groß ist.

Methoden:

* Benötigt kein Override für equals/hashCode, da nur die Informationen, ob der Knoten, in dem Set der schon bearbeiteten Knoten, existiert, relevant ist. Wichtig ist nur, ob der Knoten identisch mit einem im Set ist.
* Tue
* Siehe Tue von GierigOptimierer, die Tätigkeit von Tue ist dieselbe wie die von GierigOptimierer, jedoch dass bei AnnealingOptimierer mehr „vorbereitet“ wird (für den tueNaechsten)
* tueNaechsten
  + Siehe tueNaechsten von GierigOptimierer, die Todo-Liste funktioniert genauso auch hier, nur gibt es noch zusätzliche Variablen.
* Immer bevor ein Knoten bearbeitet wird, wird zufallProb mit zufallFaktor multipliziert, dieser ist < 1. Somit wird sichergestellt, dass die Wahrscheinlichkeit, den nächsten Knoten zufällig zu wählen immer kleiner wird (Annealing)
* Enthält zusätzlich zu der Queue und der Liste einen Set von Knoten als Parameter, in dem alle schon bearbeiteten Knoten gespeichert sind. Denn den Knoten, der von prioTodo geholt wurde, in der Zufallsliste zu entfernen kostet O(n), genauso umgekehrt. Daher ist es möglich, dass ein schon bearbeiteter Knoten nochmals auftaucht (in der Liste oder in der Queue). Deswegen wird bevor der Knoten bearbeitet wird, überprüft, ob dieser nicht schon in dem Set vorhanden ist.

### (interface) Strategie

Wie bereits anfangs erwähnt ist dies die Strategie, mit welcher Priorität ein Knoten auf Nachfolger untersucht wird

Methode:

* berechnePriorität
  + gibt als Wert eine Zahl (int), also die Priorität des gegebenen Knotens zurück
  + Eine höhere Priorität bedeutet, dass Nachfolgeknoten bevorzugt werden sollen. Ein Knoten mit Priorität 5 soll also vor einem Knoten mit Priorität 4 bearbeitet.

## Lauf

### (abstrakt) Lauf

Methode:

* *(statisch und protected)* outputErgebnis
  + Methode ist protected, damit nur die Unterklassen von Lauf auf outputErgebnis zugreifen können und kein Externer
  + IOException, damit es keine Probleme aufbereitet, wenn der Parameter name, also der Name der Datei, in der später der Output stehen soll, nicht existiert.

### Gierigsuche

* Ist eine Unterklasse von Lauf
* ausführbar
* Die Priorität der Knoten setzt sich aus dem Maximal Laub auf dem Schulhof des Knotens zusammen. Dies ist auch die Strategie, es wird versucht, so viel Laub wie möglich auf das Zielfeld zu blasen (oder wenn kein Feld vorgegeben ist, auf eines der Innenfelder).
* Ruft GierigOptimierer als Optimierer auf

### GierigHeuristikZentrum

* Ist eine Unterklasse von Lauf
* ausführbar
* Die Priorität berechnet sich aus dem Feld mit dem maximalen Laub subtrahiert mit den gegebenen Blasekosten, die mit dem jeweiligen Schritt des Knotens (siehe Knoten) multipliziert werden.
* Ruft GierigOptimierer als Optimierer auf

### Annealingsuche

* Ist eine Unterklasse von Lauf
* ausführbar
* Die Priorität ist dieselbe wie der der Gierigsuche
* Ruft AnnealingOptimierer als Optimierer auf

### AnnealingHeuristikZentrum

* Ist eine Unterklasse von Lauf
* ausführbar
* Die Priorität ist dieselbe wie der der GierigHeuristikzentrum, nur dass das Feld mit dem meisten Laub 4-fach gewertet und alle anderen Felder, auf denen der Traktor das Laub abholen könnte, dazu addiert werden.
* Ruft AnnealingOptimierer als Optimierer auf

### Tiefensuche

* Ist eine Unterklasse von Lauf
* ausführbar
* Die Priorität ist die Schritttiefe des Knotens (siehe Schritt bei Knoten).
* Ruft GierigOptimierer als Optimierer auf

# Evaluation

## Ergebnisübersicht

Jeder der Läufe ist so parametrisiert, dass er ca. 500.000 Blase-Operationen simuliert und auswertet. Das dauert ca. 10 Sekunden (Standard Desktop PC).

Für jede Fragestellung fassen zwei Tabellen die Ergebnisse zusammen:

* Die jeweils *beste gefundene Zugreihenfolge* pro Optimierungsansatz und Priorisierungsstrategie. Die Tabelle ist aufsteigend geordnet nach dem besten erzielten Wert.
* Die jeweiligen *Bestwerte* nach einer *Anzahl von Zügen* pro Optimierungsansatz und Priorisierungsstrategie. Der Bestwert ergibt sich aus dem Laub auf dem Zielfeld, wenn vorgegeben, oder ansonsten einem beliebigen Innenhof-Feld. Bricht die beste Blasereihenfolge wegen dieser Restriktion der Zahl der simulierten Blase-Operationen schon vor der gegebenen Anzahl der Züge ab, so ist das mit --- gekennzeichnet.

### Ohne fest vorgegebenes Zielfeld

| **Lauf/Strategie** | **Bester Schulhof: Laub auf Innenfeldern** | **Bester Schulhof: Zahl der benötigten Blase-Operationen** | **Einzigartige gesehene Schulhöfe** |
| --- | --- | --- | --- |
| Tiefensuche\_4 | 400 | 4 | 284.240 |
| GierigsucheMitBlasekosten\_80 | 696 | 9 | 398.373 |
| GierigsucheMitBlasekosten\_40 | 701 | 13 | 288.035 |
| GierigsucheMitBlasekosten\_20 | 765 | 21 | 296.051 |
| GierigsucheMitBlasekosten\_10 | 878 | 51 | 305.924 |
| Annealingsuche | 1.347 | 708 | 142.927 |
| Gierigsuche | 1.395 | 703 | 92.734 |
| GierigsucheHeuristikZentrum | 1.954 | 1.307 | 205.300 |

Tabelle 1: Bestes Ergebnis je Optimierungsansatz und Priorisierungsstrategie (ohne fest vorgegebenes Zielfeld)

| **Lauf/Strategie** | **4** | **9** | **13** | **21** | **51** | **703** | **1.307** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Tiefensuche\_4 | 400 | --- | --- | --- | --- | --- | --- |
| GierigsucheMitBlasekosten\_80 | 365 | 696 | --- | --- | --- | --- | --- |
| GierigsucheMitBlasekosten\_40 | 357 | 655 | 701 | --- | --- | --- | --- |
| GierigsucheMitBlasekosten\_20 | 357 | 576 | 621 | 765 | --- | --- | --- |
| GierigsucheMitBlasekosten\_10 | 290 | 374 | 448 | 525 | 878 | --- | --- |
| Annealingsuche | 261 | 360 | 412 | 427 | 505 | 1.345 | --- |
| Gierigsuche | 261 | 360 | 412 | 427 | 505 | 1.395 | --- |
| GierigsucheHeuristikZentrum | 395 | 481 | 555 | 689 | 851 | 1.771 | 1.954 |

Tabelle 2: Bestwerte nach Anzahl von Zügen je Optimierungsansatz und Priorisierungsstrategie (ohne fest vorgegebenes Zielfeld)

### Mit fest vorgegebenem Zielfeld

Das Zielfeld ist den Läufen auf (2,1) gesetzt.

| **Lauf/Strategie** | **Bester Schulhof: Laub auf Innenfeldern** | **Bester Schulhof: Zahl der benötigten Blase-Operationen** | **Einzigartige gesehene Schulhöfe** |
| --- | --- | --- | --- |
| Tiefensuche\_4 | 389 | 4 | 1.132.722 |
| GierigsucheMitBlasekosten\_80 | 569 | 8 | 336.123 |
| GierigsucheMitBlasekosten\_40 | 698 | 14 | 273.589 |
| GierigsucheMitBlasekosten\_20 | 679 | 22 | 310.427 |
| GierigsucheMitBlasekosten\_10 | 947 | 49 | 262.043 |
| GierigsucheHeuristikZentrum | 1.419 | 374 | 252.368 |
| Gierigsuche | 1.488 | 1.105 | 151.480 |
| Annealingsuche | 1.492 | 1.066 | 158.201 |

Tabelle 3: Bestes Ergebnis je Optimierungsansatz und Priorisierungsstrategie (mit fest vorgegebenem Zielfeld)

| **Lauf/Strategie** | **4** | **8** | **14** | **22** | **49** | **374** | **1.066** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Tiefensuche\_4 | 389 | --- | --- | --- | --- | --- | --- |
| GierigsucheMitBlasekosten\_80 | 320 | 569 | --- | --- | --- | --- | --- |
| GierigsucheMitBlasekosten\_40 | 381 | 510 | 698 | --- | --- | --- | --- |
| GierigsucheMitBlasekosten\_20 | 324 | 412 | 565 | 679 | --- | --- | --- |
| GierigsucheMitBlasekosten\_10 | 324 | 394 | 527 | 623 | 947 | --- | --- |
| GierigsucheHeuristikZentrum | 324 | 385 | 501 | 592 | 794 | 1.419 | --- |
| Annealingsuche | 324 | 394 | 463 | 487 | 544 | 1.027 | 1.492 |
| Gierigsuche | 324 | 394 | 463 | 487 | 544 | 1.027 | 1.488 |

Tabelle 4: Bestwerte nach Anzahl von Zügen je Optimierungsansatz und Priorisierungsstrategie (mit fest vorgegebenem Zielfeld)

## Interpretation

### Ohne fest vorgegebenes Zielfeld

Die **Tiefensuche** mit limitierter Tiefe liefert die Basis, was das beste Ergebnis nach einer gewissen (kleinen) Anzahl von Schritten überhaupt möglich ist. Dass dabei viele unvorteilhafte Knoten weiterverfolgt werden, führt zu hoher Laufzeit bei schlechtem Ergebnis.

Die **Gierigsuche** verfolgt den jeweils besten Schulhof weiter und führt nur zu einem lokalen Maximum. Da durch weitere Blase-Operationen das Ergebnis immer weiter (aber auch immer weniger) verbessert werden kann, werden lange Ketten von Blase-Operationen untersucht. Dass das Ergebnis nach so vielen Blase-Operationen stark verbessert ist, ist nicht verwunderlich. Auffällig ist, dass sehr viel weniger einzigartige Schulhöfe im Lauf gesehen werden - nur 92.734 der 500.000 insgesamt Möglichen. Das liegt wohl daran, dass gezielt Permutationen derselben Blase-Operationen ausprobiert werden (und als Duplikate verworfen werden).

Die **Gierigsuche mit Blasekosten** berücksichtigt die Zahl der benötigten Blase-Operationen in der Priorität, nach der Knoten zur weiteren Verfolgung ausgewählt werden. Die Zielfunktion (maximales Laub auf einem der Innenhoffelder) bleibt aber gemäß der Aufgabe dieselbe und berücksichtigt die Zahl der benötigten Blase-Operationen nicht. Wie erwartet nähert sich die *Gierigsuche* mit steigendenen *Blasekosten* der Tiefensuche. Besonders interessant ist der Zielkonflikt zwischen Zahl der Schritte und dem erzielten Ergebnis. Während in den ersten Schritten noch knapp 100 Steigerung pro Schritt möglich sind, fällt dieser Wert auf ca. 10 pro Schritt zwischen dem 10. und 20. Schritt. Am Ende liegt er nur noch bei 3 oder 4. Auch ist bemerkenswert, dass die Läufe mit geringeren Blasekosten längerfristig angelegt sind und nach derselben Zahl von Schritten ein vergleichsweise schlechtes Ergebnis haben. Der Schulhof wird also erst "vorbereitet", um danach effizienter zu sein.

Die **Gierigsuche mit Zentrum-Heuristik** geht in eine ganz andere Richtung. In die Priorität der gierigen Abarbeitungsreihenfolge geht hier nicht nur der Wert des maximalen Laubs ein. Dazu kommt noch die Summe des Laubes auf allen Innenhoffelder. Diese *Heuristik* basiert auf der Annahme, dass das Laub der Innenhoffelder relativ einfach noch auf das Maximalfeld geblasen werden kann. Das Laub auf ihnen ist also zum guten Teil ein Indikator für zukünftige Erfolge. Das herausragende Ergebnis (ein zur *Gierigsuche* über 50% gesteigerte Zielwert) bestätigt diese Annahme. Außerdem ist interessant, dass hier die verfolgten Ketten noch tiefer sind. Durch die ständige Steigerung des Zielwertes kommen Alternativen für frühere Knoten nicht zur Berücksichtigung.

Als Alternative ist auch die **Annealingsuche** umgesetzt, die das *simulated annealing* implementiert. Mit einer exponentiell fallenden Wahrscheinlichkeit wird zufällig ein Knoten unabhängig von seiner Priorität auf mögliche Nachfolger untersucht. Diese Wahrscheinlichkeit und ihr Verfall sind so eingestellt, dass im gesamten Lauf ca. 25% die Untersuchungen auf diesem Zufallsprinzip beruhen. Das Ergebnis lässt den Schluss zu, dass diese Randomisierung hier keinen Vorteil bringt. Das lässt sich so interpretieren, dass durch lokale Maxima durch die iterativen Blase-Operationen wenig oder kaum ausgeprägt sind. Durch die Verfolgung zufälliger Knoten ist das Ergebnis ein bisschen schlechter als beim *gierigen Ansatz*.

### Mit fest vorgegebenem Zielfeld

Abweichend von obiger Budgetierung auf ca. 500.000 Blase-Operationen benötigt hier die Tiefensuche *viermal mehr* Blase-Operationen, um exhaustiv bis Schritttiefe 4 zu durchsuchen. Dies liegt daran, dass das Zielfeld die vertikale und diagonale Spiegelachse außer Kraft setzt und Äquivalenzklassen nur noch aus **2** statt bestehen statt der **8** für dann, wenn kein Zielfeld vorgegeben ist.

In den Ergebnissen zeigen sich keine großen Unterschiede bezüglich des jeweils erreichten Bestwerts. Das Vorabfestlegen des Zielfelds bringt also diesbezüglich keine Nachteile. Die einzige Ausnahme ist die **Gierigsuche mit Zentrum-Heuristik**, die deutlich schlechter abschneidet als ohne Zielfeld und sogar noch ein wenig schlechter als die *Gierigsuche* ist. Das liegt zum einen daran, dass das Zielfeld am Rand des Innenhofes liegt und manche angrenzende Felder bei der Heuristik nicht berücksichtigt werden. Zum anderen profitiert diese Heuristik ohne vorgegebenem Zielfeld sehr stark davon, dass alle Innenhoffelder Maximalfelder werden können. Dies entspricht genau der Berücksichtigung aller Innenhoffelder durch die Heuristik. Ihr volles Potential kann die Heuristik also nur ohne vorgegebenes Zielfeld entfalten.

# Beispiele

In diesem Kapitel werden zunächst Beispiele für die Schlüsselkonzepte der Lösungsidee gegeben. Weitere Beispiele finden sich in den Unit-Tests der entsprechenden Klassen.

Anschließend werden Auszüge aus den Optimierungsergebnissen gezeigt. Das log-Verzeichnis beinhaltet hier die kompletten Informationen.

## Äquivalenz & Repräsentanten

### Quadratisch ohne Zielfeld

Die folgenden Schulhöfe sind äquivalent, da sie sich aus einer Sequenz von Spiegelungen ergeben. Ihr Repräsentant ist der String 9,2,7,1,4,6,5,3,8@3,3, der sich aus dem Maximum des lexikografischen Vergleichs ergibt. Andere Vergleiche (Minimum, Einzelfeldvergleich in int) wären auch möglich. Entscheidend ist lediglich, dass diese Berechnung immer auf dieselben Art und Weise erfolgt und für alle äquivalenten Schulhöfe zum selben Ergebnis kommt.

Schulhof, aus dem sich der Repräsentant ergibt:

-------

|9|2|7|

-------

|1|4|6|

-------

|5|3|8|

-------

Schulhof, der durch **Diagonalspiegelung** zum Repräsentanten gelangt:

-------

|9|1|5|

-------

|2|4|3|

-------

|7|6|8|

-------

Schulhof, der durch **Horizontalspiegelung** zum Repräsentanten gelangt:

-------

|7|2|9|

-------

|6|4|1|

-------

|8|3|5|

-------

Schulhof, der durch **Vertikalspiegelung** zum Repräsentanten gelangt:

-------

|5|3|8|

-------

|1|4|6|

-------

|9|2|7|

-------

Schulhof, der durch **Horizontal- und Vertikalspiegelung** zum Repräsentanten gelangt:

-------

|8|3|5|

-------

|6|4|1|

-------

|7|2|9|

-------

Schulhof, der durch **Vertikal und Diagonalspiegelung** zum Repräsentanten gelangt:

-------

|7|6|8|

-------

|2|4|3|

-------

|9|1|5|

-------

Schulhof, der durch **Horizontal- und Diagonalspiegelung** zum Repräsentanten gelangt:

-------

|5|1|9|

-------

|3|4|2|

-------

|8|6|7|

-------

Schulhof, der durch **Horizontal-, Vertikal- und Diagonalspiegelung** zum Repräsentanten gelangt:

-------

|8|6|7|

-------

|3|4|2|

-------

|5|1|9|

-------

### Rechteck ohne Zielfeld

Bei nicht quadratischen Schulhöfen entfällt die Möglichkeit der Diagonalspiegelungen. In einer Äquivalenzklasse sind dann jeweils vier Schulhöfe.

Schulhof, aus dem sich der Repräsentant 9,2,7,0,1,4,6,5,5,3,8,2@4,3 ergibt:

---------

|9|2|7|0|

---------

|1|4|6|5|

---------

|5|3|8|2|

---------

Schulhof, der durch **Horizontalspiegelung** zum Repräsentanten gelangt:

---------

|0|7|2|9|

---------

|5|6|4|1|

---------

|2|8|3|5|

---------

Schulhof, der durch **Vertikalspiegelung** zum Repräsentanten gelangt:

---------

|5|3|8|2|

---------

|1|4|6|5|

---------

|9|2|7|0|

---------

Schulhof, der durch **Horizontal- und Vertikalspiegelung** zum Repräsentanten gelangt:

---------

|2|8|3|5|

---------

|5|6|4|1|

---------

|0|7|2|9|

---------

### Mit Zielfeld

Wenn ein Zielfeld fest vorgegeben ist, darf es sich durch die Spiegelungen nicht verändern. Dies ist nicht nur gegeben, wenn es auf den angewendeten Spiegelachsen liegt. Zum Beispiel sind beide folgenden Schulhöfe äquivalent, da der erste sich durch Horizontal-, Vertikal- und Diagonalspiegelung aus dem zweiten ergibt.

--------- ---------

|9|2|7|0| |2|4|5|0|

--------- ---------

|1|4‖6‖5| |8|9‖6‖7|

--------- ---------

|3|8|9|4| |3|8|4|2|

--------- ---------

|5|3|8|2| |5|3|1|9|

--------- ---------

## Blase-Operationen

### Vervollständigung der Aufgabenstellung

Mehrere Randfälle sind in der Aufgabenstellung nicht abgedeckt. Dieser Abschnitt zeigt ihre Definition. Dabei wird davon ausgegangen, dass

* der Schulhof in alle Richtungen umzäunt ist,
* kein Rückstoß aus dem Blasen gegen den Zaun entsteht.

Beim Blasen entlang des Rands gibt es am Zaun keinen Seitenverlust:

----------------- -----------------

| > | 40| 50| 60| | > | 0| 81| 65|

----------------- ==> -----------------

| 0| 10| 20| 30| | 0| 10| 24| 30|

----------------- -----------------

Beim Blasen zwei Felder vor dem Zaun gibt es keinen "Überlauf" nach hinten:

----------------- -----------------

| 0| > | 50| 60| | 0| > | 0|105|

----------------- ==> -----------------

| 0| 10| 20| 30| | 0| 10| 20| 35|

----------------- -----------------

Das Blasen ein Feld vor dem Zaun hat keine Wirkung:

----------------- -----------------

| 0| 40| 50| 60| | 0| 40| 50| 60|

----------------- ==> -----------------

| 0| ^ | 20| 30| | 0| ^ | 20| 30|

----------------- -----------------

Ebenso wenig das Blasen direkt vor dem Zaun:

----------------- -----------------

| 0| 40| 50| 60| | 0| 40| 50| 60|

----------------- ==> -----------------

| 0| v | 20| 30| | 0| v | 20| 30|

----------------- -----------------

### Potenziell verändernde Blase-Operationen

Für einen 5∗5 Schulhof sind alle 60 Blaseoperationen gezeigt, die das Potential haben, die Laubsitutation auf dem Schulhof zu verändern. Es wird nur von Potential gesprochen, da eine tatsächliche Veränderung natürlich von der Laubsituation abhängig ist (liegt zum Beispiel überhaupt kein Laub, wird keine Operation zu einer Veränderung führen).

---------------------

| | | | | |

| >| >|< >|< |< |

| v | v | v | v | v |

---------------------

| | | | | |

| >| >|< >|< |< |

| v | v | v | v | v |

---------------------

| ^ | ^ | ^ | ^ | ^ |

| >| >|< >|< |< |

| v | v | v | v | v |

---------------------

| ^ | ^ | ^ | ^ | ^ |

| >| >|< >|< |< |

| | | | | |

---------------------

| ^ | ^ | ^ | ^ | ^ |

| >| >|< >|< |< |

| | | | | |

---------------------

## Optimierte Zugfolgen

Im Folgenden gibt es ausgewählte Auszüge aus dem Ergebnis der Optimierung.

### Tiefensuche (exhaustive Suche) mit Tiefe 4 und Zielfeld (2,1)

Optimierungsdauer: 22.000 secs

Gesehene einzigartige Schulhoefe: 1132722

Durchgefuehrte Blaseoperationen: 2214540

Laengste Schrittfolge: 4

==================================================

Bester Schulhof

  Maximal-Laub: 389

  Schritte: 4

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100‖ 100‖ 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 1: (4, 2) <

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100‖ 110‖ 100| 100|

--------------------------

| 100| 110| 170|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 2: (2, 3) ^

--------------------------

| 100| 100| 111| 100| 100|

--------------------------

| 100| 117‖ 235‖ 117| 100|

--------------------------

| 100| 110|   0|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 3: (0, 1) >

--------------------------

| 100| 100| 122| 100| 100|

--------------------------

| 100|   0‖ 307‖ 140| 100|

--------------------------

| 100| 110|  11|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 4: (4, 1) <

--------------------------

| 100| 100| 136| 100| 100|

--------------------------

| 100|  30‖ 389‖   0| 100|

--------------------------

| 100| 110|  25|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

### Gierigsuche und Zielfeld (2,1)

Optimierungsdauer: 5.268 secs

Gesehene einzigartige Schulhoefe: 151480

Durchgefuehrte Blaseoperationen: 500000

Laengste Schrittfolge: 1113

==================================================

Bester Schulhof

  Maximal-Laub: 1488

  Schritte: 1105

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100‖ 100‖ 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 1: (4, 1) <

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 110‖ 170‖   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 2: (2, 3) ^

--------------------------

| 100| 100| 127| 100| 100|

--------------------------

| 100| 121‖ 241‖  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 3: (0, 1) >

--------------------------

| 100| 100| 139| 100| 100|

--------------------------

| 100|   0‖ 314‖  35| 100|

--------------------------

| 100| 100|  12| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 4: (4, 0) <

--------------------------

| 100| 113| 216|   0| 100|

--------------------------

| 100|   0‖ 324‖  35| 100|

--------------------------

| 100| 100|  12| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

…

Op 9: (0, 1) >

--------------------------

| 100|   0| 340|  21| 100|

--------------------------

| 110|   0‖ 427‖  83| 100|

--------------------------

| 100|   0|   9|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

…

Op 21: (4, 0) <

--------------------------

| 100|  30| 307|   0| 100|

--------------------------

| 110|  42‖ 484‖   0| 100|

--------------------------

| 100|   0|  17|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

…

Op 52: (0, 0) >

--------------------------

| 100|   0| 247|  24| 100|

--------------------------

| 110|  42‖ 550‖   0| 100|

--------------------------

| 100|   0|  17|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

…

Op 374: (0, 0) >

--------------------------

| 206|   0|  17|   1| 110|

--------------------------

|   0|  20‖1027‖   0| 100|

--------------------------

| 100|   0|   9|   0| 100|

--------------------------

| 100| 100|   0|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

…

Op 1105: (0, 0) >

--------------------------

| 402|   0|   9|   0| 329|

--------------------------

|   5|   0‖1488‖   0|   0|

--------------------------

|   0|   0|   0|   0|   1|

--------------------------

|   1|   0|   0|   0|   0|

--------------------------

| 165|   0|   0|   0| 100|

--------------------------

### Gierigsuche mit Kosten 20 und Zielfeld (2,1)

Optimierungsdauer: 6.172 secs

Gesehene einzigartige Schulhoefe: 310427

Durchgefuehrte Blaseoperationen: 500000

Laengste Schrittfolge: 22

==================================================

Bester Schulhof

  Maximal-Laub: 679

  Schritte: 22

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100‖ 100‖ 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 1: (4, 1) <

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 110‖ 170‖   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 2: (2, 3) ^

--------------------------

| 100| 100| 127| 100| 100|

--------------------------

| 100| 121‖ 241‖  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 3: (0, 1) >

--------------------------

| 100| 100| 139| 100| 100|

--------------------------

| 100|   0‖ 314‖  35| 100|

--------------------------

| 100| 100|  12| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 4: (4, 2) <

--------------------------

| 100| 100| 139| 100| 100|

--------------------------

| 100|   0‖ 324‖  35| 100|

--------------------------

| 100| 101|  91|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 5: (2, 3) ^

--------------------------

| 100| 100| 171| 100| 100|

--------------------------

| 100|   9‖ 365‖  44| 100|

--------------------------

| 100| 101|   0|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 6: (2, 4) ^

--------------------------

| 100| 100| 171| 100| 100|

--------------------------

| 100|   9‖ 365‖  44| 100|

--------------------------

| 100| 112|  88|  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 7: (2, 3) ^

--------------------------

| 100| 100| 207| 100| 100|

--------------------------

| 100|  17‖ 401‖  52| 100|

--------------------------

| 100| 112|   0|  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 8: (1, 3) ^

--------------------------

| 100| 101| 207| 100| 100|

--------------------------

| 111| 106‖ 412‖  52| 100|

--------------------------

| 100|   0|   0|  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 9: (0, 1) >

--------------------------

| 100| 101| 217| 100| 100|

--------------------------

| 111|   0‖ 457‖  93| 100|

--------------------------

| 100|   0|  10|  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 10: (1, 0) >

--------------------------

| 100| 101|   0| 286| 110|

--------------------------

| 111|   0‖ 457‖ 114| 100|

--------------------------

| 100|   0|  10|  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 11: (4, 1) <

--------------------------

| 100| 101|  11| 286| 110|

--------------------------

| 111|  45‖ 504‖   0| 100|

--------------------------

| 100|   0|  21|  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 12: (4, 0) <

--------------------------

| 100| 102| 268|   0| 110|

--------------------------

| 111|  45‖ 532‖   0| 100|

--------------------------

| 100|   0|  21|  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 13: (3, 0) <

--------------------------

| 110| 334|   0|   0| 110|

--------------------------

| 111|  71‖ 532‖   0| 100|

--------------------------

| 100|   0|  21|  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 14: (0, 0) >

--------------------------

| 110|   0| 301|   0| 110|

--------------------------

| 111|  71‖ 565‖   0| 100|

--------------------------

| 100|   0|  21|  11| 100|

--------------------------

| 100| 100|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 15: (1, 4) ^

--------------------------

| 110|   0| 301|   0| 110|

--------------------------

| 111|  71‖ 565‖   0| 100|

--------------------------

| 110|  80|  31|  11| 100|

--------------------------

| 100|   0|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 16: (1, 3) ^

--------------------------

| 110|   7| 301|   0| 110|

--------------------------

| 119| 128‖ 573‖   0| 100|

--------------------------

| 110|   0|  31|  11| 100|

--------------------------

| 100|   0|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 17: (0, 1) >

--------------------------

| 110|   7| 313|   0| 110|

--------------------------

| 119|   0‖ 620‖  57| 100|

--------------------------

| 110|   0|  43|  11| 100|

--------------------------

| 100|   0|   0| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 18: (3, 4) ^

--------------------------

| 110|   7| 313|   0| 110|

--------------------------

| 119|   0‖ 620‖  58| 100|

--------------------------

| 110|   0|  53|  90| 110|

--------------------------

| 100|   0|   0|   0| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 19: (3, 3) ^

--------------------------

| 110|   7| 313|   5| 110|

--------------------------

| 119|   0‖ 629‖ 125| 109|

--------------------------

| 110|   0|  53|   0| 110|

--------------------------

| 100|   0|   0|   0| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 20: (4, 1) <

--------------------------

| 110|   7| 325|   5| 110|

--------------------------

| 119|  62‖ 668‖   0| 109|

--------------------------

| 110|   0|  65|   0| 110|

--------------------------

| 100|   0|   0|   0| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 21: (2, 4) ^

--------------------------

| 110|   7| 325|   5| 110|

--------------------------

| 119|  62‖ 674‖   0| 109|

--------------------------

| 110|   0|  59|   0| 110|

--------------------------

| 100|   0|   0|   0| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

Op 22: (2, 4) ^

--------------------------

| 110|   7| 325|   5| 110|

--------------------------

| 119|  62‖ 679‖   0| 109|

--------------------------

| 110|   0|  54|   0| 110|

--------------------------

| 100|   0|   0|   0| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

### Gierigsuche mit Zentrumsheuristik ohne fest vorgegebenem Zielfeld

Optimierungsdauer: 9.082 secs

Gesehene einzigartige Schulhoefe: 205300

Durchgefuehrte Blaseoperationen: 500000

Laengste Schrittfolge: 1313

Zufaellige / prioritaetsbasierte Bearbeitungen: 2775 / 8334

==================================================

Bester Schulhof

  Maximal-Laub: 1954

  Schritte: 1307

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

| 100| 100| 100| 100| 100|

--------------------------

…

Op 10: (2, 4) ^

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

| 100| 101|  98|   0| 100|

--------------------------

| 100|  69| 504|   8| 100|

--------------------------

| 100| 100|   0|   0| 100|

--------------------------

| 100| 100| 110| 100| 100|

--------------------------

…

Op 50: (1, 4) ^

--------------------------

| 120| 103|   0| 100| 100|

--------------------------

| 111|  12|   0|   0| 100|

--------------------------

|  10|  59| 851|  93| 100|

--------------------------

| 100|   0| 105|   0| 100|

--------------------------

| 110| 126|   0| 100| 100|

--------------------------

…

Op 100: (0, 0) v

--------------------------

| 129|   0|  97|   9| 110|

--------------------------

|   0|  36| 110|   0|  98|

--------------------------

| 162|  82|1109| 133|  90|

--------------------------

|   0|  11|   0|   0|   0|

--------------------------

| 110| 105|   0|   9| 100|

--------------------------

…

Op 200: (4, 3) ^

--------------------------

| 129|  37|   0|   9| 119|

--------------------------

|  33|  11|   0|   3|  35|

--------------------------

|   0|  38|1491| 181|   0|

--------------------------

|   0|  12| 150|   0|   0|

--------------------------

| 110|  33|   0|   9| 100|

--------------------------

…

Op 400: (0, 3) ^

--------------------------

| 134|  17|   0|   9| 119|

--------------------------

|  11|   4|  30|   5|   0|

--------------------------

|   0| 101|1632|   7|   9|

--------------------------

|   1|   9| 174|   0|   0|

--------------------------

| 110|  19|   0|   9| 100|

--------------------------

…

Op 800: (0, 3) ^

--------------------------

| 134|   9|   0|   9| 119|

--------------------------

|  11|  10|  30|   0|   0|

--------------------------

|   0|  97|1815|   7|   9|

--------------------------

|   0|   0|  31|   0|   0|

--------------------------

| 110|   9|   0|   0| 100|

--------------------------

…

Op 1307: (1, 0) v

--------------------------

| 134|   9|   0|   0| 119|

--------------------------

|   9|   0|  13|   0|   9|

--------------------------

|   1|  31|1954|   0|   0|

--------------------------

|   0|   2|   0|   0|   0|

--------------------------

| 110|   9|   0|   0| 100|

--------------------------

# Quellcode

Da der komplette Quellcode sehr umfangreich ist, werden im Folgenden nur Auszüge aus den Schlüsselpassagen der Lösung gegeben.

## Schulhof: berechneRepraesentant

    /\*\*

     \* Alle Schulhoefe mit derselben Normalisierung werden aequivalent genannt (also

     \* sie ergeben sich aus einer Serie von Spiegelungen).

     \*

     \* Normalisierung bedeutet, einen Repraesentanten des Schulhofes zu bekommen,

     \* der sich, auch wenn der Schulhof gespiegelt wird, genau gleich bleiben wuerde.

     \*

     \* Wenn ein Zielfeld vorgegeben ist, wird es auch entsprechend gespiegelt und

     \* muss nach der Normalisierung gleich sein, damit Schulhoefe aequivalent sind.

     \*/

    private void berechneRepraesentant() {

        String besterKandidat = "";

        // diag/horiz/vert sind wie boolean Variablen

        // -> Spiegelungsachse ist aktiv, wenn sie 1 sind

        for (int diag = 0; diag < ((holeBreite() == holeHoehe()) ? 2 : 1); diag++) {

            for (int horiz = 0; horiz < 2; horiz++) {

                for (int vert = 0; vert < 2; vert++) {

                    String kandidat;

                    if (zielfeld == null) {

                        kandidat = "";

                    } else {

                        Feld zielfeldGespiegelt = spiegele(false, zielfeld.holeX(), zielfeld.holeY(),

                            diag, horiz, vert);

                        kandidat = zielfeldGespiegelt + "@";

                    }

                    // Aufbau des Kandidaten Zeile fuer Zeile

                    for (int y = 0; y < holeHoehe(); y++) {

                        for (int x = 0; x < holeBreite(); x++) {

                            // x/y aus Sicht des Kandidaten, Transformation zu this Feldern notwendig

                            // => umgekehrte Spiegelungsreihenfolge

                            Feld gespiegelt = spiegele(true, x, y, diag, horiz, vert);

                            if (x + y > 0) {

                                // nicht erstes Feld

                                kandidat += ",";

                            }

                            kandidat += holeLaub(gespiegelt);

                        }

                    }

                    // der lexikografisch groesste Kandidat wird genommen

                    // eine andere Wahl waere auch moeglich - entscheidend ist:

                    // jeden aequivalenten Schulhof bekommt dieselbe Normalisierung

                    if (kandidat.compareTo(besterKandidat) > 0) {

                        besterKandidat = kandidat;

                    }

                }

            }

        }

        repraesentant = besterKandidat + "@" + holeBreite() + "," + holeHoehe();

    }

    /\*\*

     \* Benoetigt fuer die Repraesentantenberechnung.

     \*

     \* @param istUmgekehrt bedeutet, dass die Reihenfolge umgedreht ist. Also ist

     \* spiegele(true, spiegele(false, feld)) dasselbe wie feld (vereinfacht dargestellt).

     \*

     \* Ein Wert von "1" fuer diag / horiz / vert bedeutet, dass die Spiegelung

     \* in dieser Achse aktiv ist. Sonst ist der Wert "0".

     \*/

    private Feld spiegele(boolean istUmgekehrt, int x, int y, int diag, int horiz, int vert) {

        if (!istUmgekehrt) {

            if (vert == 1) {

                y = holeHoehe() - 1 - y;

            }

            if (horiz == 1) {

                x = holeBreite() - 1 - x;

            }

        }

        if (diag == 1) {

            int t = x;

            x = y;

            y = t;

        }

        if (istUmgekehrt) {

            if (horiz == 1) {

                x = holeBreite() - 1 - x;

            }

            if (vert == 1) {

                y = holeHoehe() - 1 - y;

            }

        }

        return new Feld(x, y);

    }

## BlaseOp: tue

    public Schulhof tue(Schulhof davor) {

        int hoehe = davor.holeHoehe();

        int breite = davor.holeBreite();

        int[][] neueFelder = new int[hoehe][breite];

        for (int x = 0; x < breite; x++) {

            for (int y = 0; y < hoehe; y++) {

                neueFelder[y][x] = davor.holeLaub(x, y);

            }

        }

        // A und B beziehen sich auf die Felder wie im Beispiel der Aufgabenstellung

        // L und R sind aus Blasrichtung gesehen die Felder oben und unten

        //   (bzw. links und rechts) von B

        // C ist das Feld hinter B

        int ax = this.x + dx;

        int ay = this.y + dy;

        if (davor.existiertFeld(this.x, this.y) && davor.existiertFeld(ax, ay)) {

            int bx = ax + dx;

            int by = ay + dy;

            int cx = bx + dx;

            int cy = by + dy;

            if (davor.existiertFeld(cx, cy)) {

                int bNachCLaub = Math.round(neueFelder[by][bx] / 10);

                neueFelder[cy][cx] += bNachCLaub;

                neueFelder[by][bx] -= bNachCLaub;

            }

            int lx = bx - Math.abs(dy);

            int ly = by - Math.abs(dx);

            int rx = bx + Math.abs(dy);

            int ry = by + Math.abs(dx);

            int aLaubZuVerteilen = neueFelder[ay][ax];

            neueFelder[ay][ax] = 0;

            int seitenLaub = Math.round(aLaubZuVerteilen / 10);

            if (davor.existiertFeld(lx, ly)) {

                neueFelder[ly][lx] += seitenLaub;

                aLaubZuVerteilen -= seitenLaub;

            }

            if (davor.existiertFeld(rx, ry)) {

                neueFelder[ry][rx] += seitenLaub;

                aLaubZuVerteilen -= seitenLaub;

            }

            if (davor.existiertFeld(bx, by)) {

                neueFelder[by][bx] += aLaubZuVerteilen;

                aLaubZuVerteilen = 0;

            }

            if (aLaubZuVerteilen != 0) {

                neueFelder[ay][ax] = aLaubZuVerteilen;

            }

        }

        return new Schulhof(neueFelder, davor.holeZielfeld());

    }

## GierigOptimierer: tueNaechsten

    private void tueNaechsten(PriorityQueue<Knoten> todo) {

        Knoten aktueller = todo.poll();

        for (BlaseOp blaseOp : blaseOps) {

            Schulhof naechsterSchulhof = blaseOp.tue(aktueller.holeSchulhof());

            zahlBlaseOps++;

            String repraesentant = naechsterSchulhof.holeRepraesentant();

            if (!gesehen.contains(repraesentant)) {

                gesehen.add(repraesentant);

                Knoten naechster = new Knoten(naechsterSchulhof, aktueller, blaseOp);

                maxSchritttiefe = Math.max(maxSchritttiefe, naechster.holeSchritt());

                if (naechsterSchulhof.holeMaxLaub() > besterKnoten.holeSchulhof().holeMaxLaub()) {

                    besterKnoten = naechster;

                }

                if (naechster.holeSchritt() < budgetSchritttiefe) {

                    naechster.setzePrioritaet(strategie.berechnePrioritaet(naechster));

                    todo.add(naechster);

                }

            }

            if (zahlBlaseOps >= budgetBlaseOp) {

                return;

            }

        }

    }

## AnnealingOptimierer: tueNaechsten

    private void tueNaechsten(PriorityQueue<Knoten> prioTodo, List<Knoten> zufallTodo,

            Set<Knoten> bearbeiteteKnoten) {

        Knoten aktueller;

        if (Math.random() <= zufallProb) {

            zahlBearbeitungZufall++;

            int indexListe = (int) Math.random() \* zufallTodo.size();

            aktueller = zufallTodo.get(indexListe);

            // keine Loesung aus prioTodo, da Kosten O(n) sein wuerden

            // keine Loesung aus prioZufall, da Kosten O(n) sein wuerden (Linksschieben)

        } else {

            zahlBearbeitungPrio++;

            aktueller = prioTodo.poll();

            // keine Loesung aus prioZufall, da Kosten O(n) sein wuerden

        }

        // Als Ausgleich fuer fehlendes Wegloeschen, werden bearbeitete

        // (=nach Nachfolgern durchgesuchte) Knoten gemerkt

        if (bearbeiteteKnoten.contains(aktueller)) {

            return;

        }

        bearbeiteteKnoten.add(aktueller);

        // annealing schreitet nur fuer neuen Knoten voran

        zufallProb \*= zufallFaktor;

        for (BlaseOp blaseOp : blaseOps) {

            Schulhof naechsterSchulhof = blaseOp.tue(aktueller.holeSchulhof());

            zahlBlaseOps++;

            String repraesentant = naechsterSchulhof.holeRepraesentant();

            if (!gesehen.contains(repraesentant)) {

                gesehen.add(repraesentant);

                Knoten naechster = new Knoten(naechsterSchulhof, aktueller, blaseOp);

                maxSchritttiefe = Math.max(maxSchritttiefe, naechster.holeSchritt());

                if (naechsterSchulhof.holeMaxLaub() > besterKnoten.holeSchulhof().holeMaxLaub()) {

                    besterKnoten = naechster;

                }

                if (naechster.holeSchritt() < budgetSchritttiefe) {

                    naechster.setzePrioritaet(strategie.berechnePrioritaet(naechster));

                    prioTodo.add(naechster);

                    zufallTodo.add(naechster);

                }

            }

            if (zahlBlaseOps >= budgetBlaseOp) {

                return;

            }

        }

    }