42. BWInf Runde 2 - Aufgabe 1

Inhaltsverzeichnis

[Aufgabenstellung 2](#_Toc162540580)

[Lösungsidee 2](#_Toc162540581)

[Umsetzung 3](#_Toc162540582)

[Klassendiagramm 3](#_Toc162540583)

[Modell 4](#_Toc162540584)

[Optimierung 6](#_Toc162540585)

[Lauf 9](#_Toc162540586)

[Evaluation 10](#_Toc162540587)

[Ergebnisübersicht 10](#_Toc162540588)

[Interpretation 13](#_Toc162540589)

[Beispiele 14](#_Toc162540590)

[Äquivalenz & Repräsentanten 14](#_Toc162540591)

[Blase-Operationen 18](#_Toc162540592)

[Optimierte Zugfolgen 20](#_Toc162540593)

[Quellcode 37](#_Toc162540594)

[Schulhof: berechneRepraesentant 37](#_Toc162540595)

[BlaseOp: tue 39](#_Toc162540596)

[GierigOptimierer: tueNaechsten 41](#_Toc162540597)

[AnnealingOptimierer: tueNaechsten 42](#_Toc162540598)

# 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:

* Werden für jeden Schulhof initial am Ende des Konstruktors kalkuliert.
* repraesentant zur Normalisierung des Schulhofes
* maxLaub, maxFeldX, maxFeldY als die Daten des Feldes, auf dem am meisten Laub vorhanden ist.

Konstruktoren:

* Basierend auf Breite, Höhe und dem Zielfeld, wenn vorgegeben
* Dasselbe und zusätzlich eine Liste der Laubwerte der Schulhoffelder. Diese werden für das eigene felder-Attribut geklont wird, damit keine Änderungen am Schulhof von außen möglich sind. Die Liste ist eindimensional und geht zeilenweise durch den Schulhof.
* Lediglich basierend auf einer Matrix der Laubwerte der Schulhoffelder und einem optionalen Zielfeld. Mit dieser Matrix sind die Breite und Höhe implizit gegeben. Die Matrix wird direkt als Attribut übernommen. Wegen der package visibility ist sichergestellt, dass kein externer Verwender den Schulhof so konstruieren und danach die Felder ändern kann. Dieser Konstruktor wird für die effiziente Durchführung der Blase-Operation benötigt.

Methoden:

* 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.
* Der Quellcode findet sich am Ende der Ausarbeitung. Durch die drei Spiegelachsen und zweidimensionale Achse werden hierfür fünf for-Schleifen ineinander verschachtelt. Hier ist es also besonders wichtig, dass die Berechnung nur einmal initial durchgeführt wird.
* Spiegele
* Spiegelt die x-y-Koordinaten eines Felds und gibt das Resultat als neues Feld zurück.
* Unterstützt die Spiegelung in Vor- und Rückrichtung abhängig vom Parameter istUmgekehrt. Dies ist wichtig, da berechneRepraesentant das Zielfeld in Vorrichtung spiegelt, die einzelnen Laubwerte der Felder aber in Rückrichtung durchgeht.
* equals
* Gibt wahr zurück, wenn other äquivalent ist, d.h.
* sie dieselben Dimensionen, dasselbe Laubmuster und dasselbe Zielfeld (wenn vorgegeben) haben,
  + oder das trifft für Spiegelungen (horizontal, vertikal, diagonal) zu.
* Das ist genau dann der Fall, wenn die Repräsentanten gleich sind. Daher benutzt die Berechnung String.equals auf ihnen.
* hashCode
* Muss für äquivalente Schulhöfe dasselbe Ergebnis liefern. Das ist dann der Fall, wenn die Hash-Codes der Repräsentanten verwendet werden. Daher benutzt die Berechnung String.hashCode auf ihnen.

### BlaseOp

Objektattribute:

* Koordinaten x, y für den Ort der „Blase-Stelle“ und dx, dy für die Definition der Blaserichtung.

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:

* *(statisch)* holeVeraenderndeOps
* Gibt eine Liste von potenziell verändernde Blase-Operationen zurück
  + Siehe schematische Darstellung im Kapitel „Beispiele“.
* Dafür werden auf jedem Feld des Schulhofs alle 4 Blaserichtungen ausprobiert. Anschließend wird die Blase-Operation (BlaseOp) auf einem Schulhof ausgeführt, der mit den Parametern Breite und Höhe und einer initialen Laub-Situation erstellt wurde.
* Eine BlaseOp ist nur dann potenziell verändernd, wenn sie die Chance hat, die Laubverteilung zu verändern.
* Dafür ist notwendig, dass die BlaseOp bei der initialen Verteilung eine Veränderung hervorruft. Daher wird die Liste anhand mehrerer Operation-Simulationen erstellt.
* tue
* Setzt die Idee von Blasen um. Dabei kommen die vorgegebenen und selbst hinzugefügten Regeln zur Anwendung.
  + Siehe schematische Darstellung im Kapitel „Beispiele“.
* Als Ausgangspunkt dient der Parameter-Schulhof. Von ihm werden die Felder für den Ergebnis-Schulhof geklont.
* Auf diesen Felder wird die Blaseoperation durchgeführt.
* Konstruiert am Ende den Ergebnis-Schulhof anhand dieser Felder und dem eventuell vorgegebenen Zielfeld des Parameter-Schulhofs.
* Der Quellcode findet sich am Ende der Ausarbeitung.

## Optimierung

### Knoten

Entsprechend der Lösungsidee repräsentiert die Klasse einen Knoten im Graphen der Zugfolgen.

Objektattribute:

* Der Schulhof mit seiner Laubsituation nach der bisherigen Zugfolge.
* Information zur Zugfolge, um zu ihm zu gelangen. Das ist also der Vorgängerknoten und die Blase-Operation, die auf dem Vorgängerknoten ausgeführt wurde, um zu diesem Knoten zu kommen.
* Zudem wird (redundant) auch der Schritt als die Tiefe der Zugfolge gehalten, damit die Optimierungsstrategie effizient darauf zugreifen kann.
* Außerdem wird eine Priorität gehalten, die von außen (vom Optimierer basierend auf seiner Strategie) gesetzt wird. Basierend auf dieser wird er früher oder später auf Nachfolger untersucht.

Methoden:

* setzePrioritaet
* Wird nur einmal gesetzt und nicht bei jedem compareTo neu berechnet, damit
* der Knoten die (Optimierungs-)Strategie nicht kennen muss,
* der Vergleich effizient ist (manche Strategien könnten aufwändiger zu berechnen sein).
* Diese Methode ist nur public, um darauf Tests machen zu können.
* compareTo
  + Implementierung von Comparable, um Knoten in der PriorityQueue des Optimierers entsprechend ihrer Priorität einordnen zu können.
  + Der Vergleich gibt die gewünschte zeitliche Abfolge der Bearbeitung zurück. Da die PriorityQueue aufsteigend sortiert, müssen also Knoten in umgekehrte Priorität verglichen werden.
* *Beispiel:* Soll ein Knoten K1 der Priorität P1 vor Knoten K2 der Priorität P2 betrachtet werden, so muss K1 < K2 sein, obwohl P1 > P2. Das bedeutet, dass das Ergebnis der Methode durch eine einfache Differenz P2 – P1 ist.

### Strategie

Wie bereits in der Lösungsidee erläutert, ist dies die Optimierungsstrategie, d.h. die Strategie, mit welcher Priorität ein Knoten auf seinen Nachfolger untersucht wird. Ein Suchlauf muss dieses Interface also implementieren, um die Untersuchungsreihenfolge festzulegen.

Methode:

* berechnePriorität
  + Berechnet die Priorität eines gegebenen Knotens bezüglich der Untersuchung auf Nachfolger und gibt sie zurück.

### Optimierer

Diese Klasse dient als Schablone für verschiedene Ansätze der Optimierung (siehe Unterklassen).

*(protected)* Objektattribute:

* Liste der potenziell verändernden Blase-Operationen
* Optimierungsstrategie
* gesehen als Lookup 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 werden hier abgelegt.
  + Das reicht aus und ist sogar wünschenswert, da äquivalente Schulhöfe nur eine bloße Spiegelung sind und nicht weiter betrachtet werden müssen.
* Die „Budget“-Variablen sind optionale Beschränkungen für die Optimierung. Diese können zwischen dem Konstruktor und der tue Methode per setze-Methoden angegeben werden.
* Die restlichen Variablen halten Statistiken des Optimierungslaufes fest und werden in den Implementierungen der tue Methode gesetzt. Sie können im Anschluss von außen abgerufen werden.

Konstruktor:

* Die „Budget“-Variablen zunächst auf unendlich gesetzt (Integer.MAX\_VALUE).

### GierigOptimierer

* Idee: Neben einer Prioritätsliste mit Knoten wird ein Set für schon bearbeitete Repräsentanten-Schulhöfe gehalten. Damit wird verhindert, dass äquivalente Schulhöfe mehrmals auf Nachfolger untersucht werden.

Methoden:

* Tue
* Bereitet die nötigen Informationen für tueNaechsten vor.
* Erstellt eine Prioritäts-Todo-Liste. Die Knoten-Implementierung von compareTo stellt sicher, dass der Knoten der größten Priorität immer als erstes zurückgegeben wird.
* Ruft tueNaechsten auf, solange die Prioritätsliste noch etwas enthält und das Budget der Blase-Operationen noch nicht aufgebraucht wurde.
* tueNaechsten
* Die Aufgabe ist es, den nächsten Prioritäts-Knoten zu bearbeiten.
* Bevor ein Knoten der Todo-Liste hinzugefügt wird, wird seine Priorität mit Hilfe der Optimierungsstrategie ausgerechnet und ihm zugewiesen.
* Der Quellcode findet sich am Ende der Ausarbeitung.

### AnnealingOptimierer

Idee:

* Ein Knoten wird mit exponentiell fallender Wahrscheinlichkeit zufällig auf mögliche Nachfolger untersucht, anstatt ihn über die Priorität zu bestimmen. Durch diesen kleiner werdenden Zufallsfaktor kann man vom Prinzip des Annealing reden.
* Um den dafür notwendigen 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-Set für schon bearbeitete Knoten, um sie nicht aus der jeweilig anderen Liste löschen zu müssen (würde *O(n)* kosten). Hierfür wird keine Reimplementierung für equals/hashCode in Knoten benötigt. Es reicht aus, auf die Identität (gleiche Objektinstanzen) zu prüfen, da der Lookup lediglich dazu verwendet wird, um zu überprüfen, ob der Knoten bereits durch das jeweilig andere Verfahren (Priorität vs. Zufall) untersucht wurde.

Konstruktor:

* Berechnet den Faktor, zu dem die Wahrscheinlichkeit zufälliger Knotenwahl jedes tue vermindert wird. Als Parameter dafür dient die „Halbwertszeit“ dieser Wahrscheinlichkeit.

Methoden:

* tueNaechsten
* Bevor ein Knoten einer Liste entnommen wird, wird per Zufall entschieden, mit welchem Verfahren der nächste Knoten gewählt wird.
* Bevor der Knoten bearbeitet wird, wird die Wahrscheinlichkeit, dass der nächste zu bearbeitender Knoten zufällig gewählt wird, mit einem im Konstruktor errechneten Faktor (< 1) kleiner berechnet.
* Der Ablauf, den Knoten zu bearbeiten ist derselbe wie tueNaechsten von GierigOptimierer. Am Schluss wird jedoch der Knoten nicht nur der Prioritätsliste hinzugefügt, sondern auch der Zufallsliste.
* Der Quellcode findet sich am Ende der Ausarbeitung.

## Lauf

Alle Unterklassen von Lauf sind ausführbar. Sie erstellen jeweils eine anonyme Klasse für ihre Strategie und übergeben sie beim Konstruieren der gewählten Implementierung von Optimierer.

### (abstrakt) Lauf

Methode:

* *(statisch und protected)* outputErgebnis
  + Die Aufgabe ist es, das Ergebnis des Laufs zu formatieren und in eine entsprechend genannte Datei zu schreiben. Dabei sind Rahmendaten wie die Anzahl an Blaseoperationen zur Übersicht angegeben. (siehe Log-Verzeichnis)
  + Gibt mögliche IOException an die Unterklasse weiter. Sie wird dort auch nicht behandelt, da im Falle von Problemen bei Dateisystemoperationen am besten ein Stack-Trace ausgedruckt wird.

### Tiefensuche

* Diese durchsucht exhaustiv den Graphen bis zu eine vordefinierte Tiefe („Budget“). Dies liefert die Basis, um zu ermitteln, was das beste Ergebnis nach einer gewissen (kleinen) Anzahl von Schritten ist.
* Um die Tiefensuche als Durchsuchungsabfolge zu erreichen, wird als Priorität die Schritttiefe des Knotens genommen.

### Gierigsuche

* Diese verfolgt den Knoten mit dem besten Schulhof weiter, also der Schulhof mit dem maximalen Laub auf einem der Innenfelder oder das Zielfeld (wenn vorgegeben). Die Priorität ist damit genau die Menge an Laub auf diesem Feld. Dies führt bekanntlich nur zu einem lokalen Maximum.

### GierigsucheMitBlasekosten

* Diese berücksichtigt die Zahl der benötigten Blase-Operationen in der Priorität, nach der Knoten zur weiteren Verfolgung ausgewählt werden. Für Kosten von 10 erhält zum Beispiel ein Knoten der Schritttiefe 2 und einem Laubwert 200 dieselbe Priorität wie ein Knoten der Schritttiefe 4 und einem Laubwert 220.
* Die Zielfunktion, dass maximales Laub auf einem der Innenhoffelder oder auf dem Zielfeld (wenn vorgegeben) bleibt, bleibt wie in der Aufgabe bestehen und berücksichtigt die Zahl der benötigten Blase-Operationen nicht. Die Auswirkung der Blasekosten bleibt also auf die Untersuchungsreihenfolge der Folgeknoten beschränkt.

### GierigsucheHeuristikZentrum

* Diese berücksichtigt für die Priorität zum einem den Wert des maximalen Laubs mit einem größeren Gewicht. Dazu kommt noch die Summe des Laubes auf allen Innenhoffeldern.
* Diese Heuristik basiert auf der Annahme, dass das Laub der Innenhoffelder relativ einfach noch auf das Maximalfeld geblasen werden kann.

### Annealingsuche

* Diese hat dieselbe Priorität wie die Gierigsuche.
* Allerdings wird nicht der GierigOptimierer, sondern der AnnealingOptimierer verwendet.

# 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;

            }

        }

    }