28. BWINF, Runde 2, Aufgabe 1: Universeller Öffnungscode

Matthias Springer

31. März 2010

1 Lösungsidee

Für alle folgenden Kapitel gilt:

Eine Schalterauswahl ist eine Auswahl von bestimmten Schaltern. Es geht um die Indizes der ausgewählten Schalter. In der Regel werden diese in einer Liste gespeichert.

Eine Schalterstellung ist eine Möglichkeit, eine bestimmte Anzahl von Schaltern zu stellen. Ein Schalter kann auf 0, 1 oder 2 gestellt werden, wobei 2 bedeutet, dass der Schalter nicht verwendet wird und er jede beliebige Stellung aufweisen kann.

1.1 Generieren gültiger Lösungen

Eine Lösung für das Problem ist genau dann gültig, wenn die Menge der Bitstrings alle möglichen *Fälle* abdeckt. Damit sind die Auswahl der M Schalter und die 2^M Möglichkeiten, diese Schalter auf 0 oder 1 zu stellen, gemeint (siehe auch vorletzter Absatz der Aufgabenstellung). Ein Lösung ist gut, wenn die Anzahl der Bitstrings möglich klein ist.

Gültige Lösungen kann man generieren, indem man zunächst eine Liste aus Bitstrings mit jeder möglichen Schalterauswahl und Schalterbelegung (0 oder 1) generiert. Wenn ein Schalter nicht benötigt wird, soll er den Wert 2 haben. Eine solche Liste wird generiert, indem zunächst alle Möglichkeiten, M von N Schaltern auszuwählen, aufgelistet werden. Für jede solche Möglichkeit werden nun alle Schalterbelegungen generiert, das heißt alle 2^M Möglichkeiten, dass die M Schalter unabhängig voneinander entweder auf 0 oder 1 stehen. Die restlichen, nicht benutzten Schalter sollen auf 2 stehen, was bedeutet, dass der Schalter nicht verwendet wird. Alle so generierten Bitstrings (bzw. Bytestrings, da es jetzt auch die Möglichkeit 2 gibt) werden in einer Liste gespeichert. Bei dieser Liste handelt es sich nun schon um eine gültige Lösung, sie ist nur sehr lang. Sie enthält nämlich genau $\begin{pmatrix} N \\ M \end{pmatrix} 2^M$ Bitstrings. Bei $N = 18$ und $M = 4$ sind das bereits 48960 Bitstrings.

Beispiel 1 *Für $N = 3$ und $M = 2$ werden die Bitstrings*

$$\{(0,0,2),(0,1,2),(1,0,2),(1,1,2),(2,0,0),(2,0,1),\\(2,1,0),(2,1,1),(0,2,0),(0,2,1),(1,2,0),(1,2,1)\} \tag{1}$$

generiert.

1.2 Ein einfacher Algorithmus

Die Anzahl der Bitstrings kann reduziert werden, indem kompatible Bitstrings kombiniert werden. Zwei Bitstrings sind kompatibel, wenn keine zwei verwendeten Schalter an der gleichen Stelle unterschiedliche Stellungen haben. Wenn ein Schalter die Stellung 2 hat macht das nichts, weil er dann nicht verwendet wird. Wenn hingegen ein Schalter im ersten Bitstring an der Position i die Stellung 0 hat, der gleiche Schalter aber im zweiten Bitstring die Stellung 1, so sind die Bitstrings nicht kompatibel.

Beispiel 2 *Die beiden Bitstrings $(0,1,1,2,2,2)$ und $(0,2,1,1,2,2)$ sind kompatibel, während die beiden Bitstrings $(0,1,1,2,2,2)$ und $(1,1,1,2,2,2)$ nicht kompatibel sind.*

Wenn zwei kompatible Bitstrings kombiniert wurden, werden beide aus der Liste gelöscht und der kombinierte Bitstring eingefügt. Es gilt nun, so viele Kombinationen durchzuführen, wie möglich.

Der Algorithmus speichert die einzelnen Bitstrings anfangs noch nicht in einer Liste. Wenn ein Bitstring[1] generiert wurde, versucht der Algorithmus diesen Bitstring mit einem bereits vorhandenen Bitstring zu kombinieren. Dazu wird die Liste der Bitstrings sortiert. Es sind diejenigen Bitstrings ganz oben, die wenig unbenutzte Schalter haben und möglichst stark mit dem aktuellen Bitstring übereinstimmen. Ein Bitstring hat wenig benutzte Schalter, wenn die Stellung 2 selten vorkommt und zwei Bitstrings sind ähnlich, wenn sie möglichst viele gleiche Schalterstellungen haben. Wenn der Bitstring kombiniert werden kann, wird der kombinierte Bitstring in der Liste gespeichert und der verwendete Bitstring aus der Liste wird gelöscht. Gibt es keinen geeigneten Kandidaten zum Kombinieren (das ist zum Beispiel ganz am Anfang der Fall, wenn die Liste noch leer ist), so wird der Bitstring einfach zur Liste hinzugefügt.

Es handelt sich also um einen *Greedy-Algorithmus*, der seine Entscheidungen immer so trifft, dass er den zur Zeit am meisten erfolgsversprechenden Zug wählt. Leider trifft er nur ganz selten die optimale Lösung (in meinen Versuchen sogar nie), er verfängt sich in einem lokalen Minimum[2]. Es handelt sich bei der Aufgabe um ein Optimierungsproblem, das vielleicht mit einem geeigneten Optimierungsalgorithmus gelöst werden kann. Damit sind Algorithmen wie *Simulierte Abkühlung* oder *genetische Algorithmen* gemeint. Diese versuchen meist, eine gute Lösung zu raten. Bessere Lösungen kann man erzielen, indem die Lösung schrittweise verändert wird. Bei vielen Optimierungsproblemen kann man sich die Tatsache zu Nutze machen, dass ähnliche Lösungen eine ähnliche Güte

[1] Ein solcher Bitstring mit nur M verwendeten Schaltern.
[2] Minimum, weil die Anzahl der Bitstrings minimiert werden soll.

aufweisen. Das Problem, dass man sich in einem lokalen Extremum verfängt, wird beim Algorithmus der *Simulierten Abkühlung* so gelöst, dass anfangs auch schlechtere Lösungen zugelassen werden[3], wobei mit fortschreitender Zeit (und sinkender Temperatur) nur noch wenige schlechtere Lösungen zugelassen werden, der Algorithmus wird dann zu einem Greedy-Algorithmus. Bei *genetische Algorithmen* kann man lokale Extrema durch zufällige Mutation der Lösungen überwinden.

Optimimierungsalgorithmen könnten die Lösung so verändern, dass die Reihenfolge, in der die Bitstrings generiert werden, vertauscht werden. Um ein bisschen damit zu experimentieren habe ich den Algorithmus so verändert, dass die Bitstrings in zufälliger Reihenfolge generiert werden[4]. Danach wurde das Programm mehrere Male hintereinander ausgeführt. Die Ergebnisse sind im Kapitel *Programm-Ablaufprotokoll* zu finden.

Der einfache Algorithmus ohne Sortieren Zufällig habe ich einmal die Codezeile zum Sortieren der Liste der schon generierten Bitstrings auskommentiert. Dabei habe ich festgestellt, dass sich die Laufzeit sehr stark verbessert hat, die Anzahl der benötigten Bitstrings jedoch nur sehr leicht gestiegen ist. Deshalb habe ich im Kapitel *Programm-Ablaufprotokoll* beide Varianten *kein Sortieren* und *mit Sortieren* betrachtet.

1.3 Ein zweiter Algorithmus

Dieser Algorithmus versucht auf andere Weise, eine gute Lösung zu finden. Dazu wird zunächst eine Liste mit allen Möglichkeiten, M von N Schaltern auszuwählen, erstellt (Liste A). Jeder Listeneintrag speichert neben den ausgewählten Schalterindizes eine weitere Liste (Liste(n) B) mit allen Möglichkeiten, die M Schalter entweder auf 0 oder auf 1 zu setzten (2^M Möglichkeiten). Alle Listen A und B werden zufällig durchgemischt[5]. Der Algorithmus versucht, so viele kompatible Elemente aus Liste A gleichzeitig zu entnehmen, wie möglich. Dazu wird zuerst das erste Element entnommen. Dann wird für jedes weitere Element der Liste A geprüft, ob es kompatibel zu den bereits entnommenen Elementen ist[6] und - wenn das der Fall ist - auch entnommen. Die Anzahl der entnommen Elemente aus Liste A sei y. Zwei Elemente aus Liste A sind kompatibel, wenn die Mengen ihrer verwendeten Schalterindizes disjunkt sind. Eine Auswahl von Elementen aus Liste A ist also gültig, wenn jeder Schalterindex maximal einmal vorkommt (bzw. jeder Schalter maximal einmal verwendet wird).

Beispiel 3 *Die Auswahl* $\{\{1,2,3,4,5\},\{6,7,8,9,10\}\}$ *ist gültig, während die Auswahl* $\{\{1,2,3,4,5\},\{5,6,7,8,12\}\}$ *ungültig ist, weil der Schalter mit dem Index* 5 *zweimal*

[3]Mit einer bestimmten Wahrscheinlichkeit.

[4]Die Listen der 2^M Schalterstellungen und M aus N Auswahlmöglichkeiten werden gemischt. Mit den genannten Optimierungsalgorithmen hatte ich leider keinen Erfolg, ich bin mir aber ziemlich sicher, dass da noch was geht.

[5]Ich habe es auch mit Sortieren versucht. Es liegt schließlich nahe, die Entscheidung zu treffen, die gerade am besten aussieht. Deshalb lieferte der Algorithmus jedoch auch keine besseren Ergebnisse. Ich bin mir noch nicht ganz sicher, warum.

[6]Das neue Element muss paarweise kompatibel mit jedem bereits entnommenen Element sein.

verwendet wird[7]*.*

z sei die maximale Anzahl von Elementen in einer gerade entnommenen Liste B [8]. Jetzt können Bitstrings gebildet werden, wobei ein Bitstring immer y (oder weniger) Fälle (Schalterstellungen, je eine aus jeder Liste B) beinhaltet. Das ist möglich, weil diese y Fälle auf jeden Fall miteinander kompatibel sind. Es kann also nicht sein, dass einem Schalter zwei verschiedene Zustände zugewiesen werden. Insgesamt werden z Bitstrings gebildet, wobei es sein kann, dass nicht jeder Bitstring y Fälle enthält, weil möglicherweise nicht alle Listen B z Schalterstellungen enthalten.

Beispiel 4 *Es sei* $N = 6$ *und* $M = 3$*. Dann enthält Liste A unter anderem die zueinander kompatiblen Fälle (Schalterauswahl)* $\{1, 2, 3\}$ *und* $\{4, 5, 6\}$[9] *(Es gilt* $y = 2$*, was bedeutet, dass zwei Elemente aus Liste A entnommen werden.). Für das erste Element existiert diese Liste B*[10]*:*

$$\{(0,0,0),(0,0,1),(0,1,0),(1,0,0),(0,1,1),...,(1,1,1)\} \tag{2}$$

Selbiges gilt für das zweite Element, es existiert die gleiche Liste[11]*. Da die Kardinalität der beiden Mengen B* $2^3 = 8$ *ist und damit* $z = 8$ *gilt, werden* 8 *Bitstrings erstellt:*

$$\begin{gathered}\{(0,0,0,0,0,0),(0,0,1,0,0,1),(0,1,0,0,1,0),\\(1,0,0,1,0,0),(0,1,1,0,1,1),...,(1,1,1,1,1,1)\}\end{gathered} \tag{3}$$

Wenn Elemente aus der Liste A entnommen werden, können diese ganz gelöscht werden. Wie u.a. aus dem obigen Beispiel ersichtlich werden sollte, wurden alle möglichen Fälle bei jeder der beiden Schalterauswahlen berücksichtigt. Es mag einem sinnvoll erscheinen, die Elemente aus Liste A zuerst zu entnemen, die die meisten Elemente in ihrer Liste B aufweisen. In eigenen Experimenten habe ich festgestellt, dass das keinen Einfluss auf die Anzahl der Bitstrings hat[12]. Die generierten Bitstrings enthalten jedoch nicht nur die Fälle, die bei der Generierung aus den Listen B entnommen wurden. Es sind logischerweise auch Fälle für andere Elemente A enthalten.

Beispiel 5 *Wenn man die Bitstrings aus dem letzten Beispiel betrachtet, sind für die Schalterauswahl* $\{2, 3, 4\}$ *diese Schalterstellungen enthalten*[13]*:*

$$\{(0,0,0),(0,1,0),(1,0,0),(0,0,1),(1,1,0),...,(1,1,1)\} \tag{4}$$

[7] In diesem Beispiel gilt übrigends $M = 5$, was man daran erkennt, dass jede Untermenge 5 Indizes beinhaltet

[8] Gemeint sind die Listen B, die entnommen wurden, als Elemente von Liste A entnommen wurden.

[9] Diese Mengen enthalten die Indizes der ausgewählten Schalter.

[10] Wegen $N = 6$ muss jedes Tupel eigentlich 6 Elemente enthalten. Man kann jedes Tupel gedanklich mit drei Zweien hinten auffüllen, da die Schalter $\{4, 5, 6\}$ nicht verwendet werden.

[11] Dabei sind die Tupel aber gedanklich vorne mit jeweils drei Zweien aufzufüllen, da die Schalter $\{1, 2, 3\}$ nicht verwendet werden.

[12] Ich vermute dass es daran liegt, dass zum Schluss immer viele Schalterauswahlen übrig bleiben, die nur wenig Elemente in ihrer Liste B haben und nicht ohne weiteres effizient untergebracht werden können. Warum das so ist, sollte aus den nächsten Sätzen klar werden.

[13] Eigentlich $\{(2,0,0,0,2,2),...\}$.

Diese Schalterstellungen wurden automatisch mitgeneriert. In Wirklichkeit reduziert sich die Anzahl der noch zu betrachtenden Fälle nur dadurch drastisch, dass immer sehr viele solche Fälle mitgeneriert werden. Leider werden niemals/sehr selten[14] alle möglichen Fälle mitgeneriert, weil oftmals gleiche Schalterstellungen entstehen. Wenn man das vorherige Beispiel betrachtet bedeutet das, dass das die Menge mit den Tupeln mehrere gleiche Tupel enthalten kann[15] (anstatt lauter verschiedene). Die Effizienz des Verfahrens hängt von der Anordnung der Elemente in den Listen B ab. Diese werden zufällig durchgemischt. Dadurch kann der Algorithmus mehr Fälle mitgenerieren. Ich habe versucht, den Algorithmus in diese Richtung zu verbessern, was mir leider nicht gelungen ist. Meine Versuche möchte ich im nächsten Kapitel trotzdem kurz erklären.

Es gibt nicht nur Fälle die automatisch mitgeneriert werden. Wenn zum Beispiel $N = 18$ und $M = 4$ gelten, bleiben in jedem Bitstring immer zwei unbenutze Schalter übrig. Diese können nach dem Prinzip des Algorithmus aus dem vorherigen Kapitel besetzt werden. Es wird einfach nach noch nicht behandelten Fällen gesucht (aus allen Listen B) und wenn ein solcher Fall kombiniert werden kann, werden der Bitstring entsprechend erweitert und der Eintrag aus der Liste B entfernt. Das geht für kleine Werte M recht schnell, bei $M = 5$ gibt es zum Beispiel maximal $2^5 = 32$ Bitstrings zu vervollständigen.

Es wurden nun maximal 2^M Bitstrings erstellt. Dieses Verfahren wird so lange wiederholt, bis alle Listen B leer sind. Wenn eine Liste B leer ist, kann auch das dazugehörige Element aus Liste A gelöscht werden. Die Liste aller generierten Bitstrings stellt die Lösung dar.

1.4 Anordnung der Listen B

Es gilt $N = 6$ und $M = 3$. Es geht darum, die Listen B so anzuordnen, dass möglichst viele Fälle abgedeckt werden (mitgeneriert werden). Dazu habe ich ein eigenes Programm geschrieben, das zwei Mengen

$$\{(0,0,0),(0,0,1),(0,1,0),(1,0,0),(0,1,1),...,(1,1,1)\} \tag{5}$$

erzeugt, wobei Menge 1 der Schalterauswahl $\{1,2,3\}$ und Menge 2 der Schalterauswahl $\{4,5,6\}$ entsprechen. Dann werden alle Permutationen der Tupel in der zweiten Menge erzeugt. Das sind $8! = 40320$ Möglichkeiten. Für jede Permutation werden nun $z = 8$ Bitstrings erstellt; jeder Bitstring besteht aus zwei zusammengesetzten Tupeln, aus je einem der beiden Mengen (siehe auch vorherige Beispiele). Jetzt soll überprüft werden, wie viele Fälle (Schalterauswahl und Schalterstellung) mit diesen 8 Bitstrings nicht berücksichtigt werden können[16]. Dabei stellte sich heraus, dass im besten Fall nur 16 Fälle vorhanden sind, die die 8 Bitstrings nicht abdecken können, im Normalfall aber etwa 30.

```
1 48 fehlend: 48
```

[14] Da bin ich mir nicht ganz sicher.

[15] Dann ist es streng genommen keine Menge mehr.

[16] Es werden alle Möglichkeiten, M aus N Schalter auszuwählen und alle 2^M Möglichkeiten, diese Schalter zu setzen, betrachtet.

```
40 fehlend: 1440
36 fehlend: 4608
34 fehlend: 5760
32 fehlend: 10368
30 fehlend: 3456
28 fehlend: 6912
26 fehlend: 5760
24 fehlend: 672
22 fehlend: 1152
16 fehlend: 144
```

Das obige Listing zeigt die Ausgabe meines Programms. x *fehlend* bedeutet, dass mit einer Permutation x Fälle nicht abgedeckt werden konnten. Die Zahl dahinter sagt aus, bei wie vielen Permuationen das der Fall war[17]. Es gibt also 144 nahezu optimale Permutationen. Es ist äußerst unwahrscheinlich, diese zu treffen, wenn man die Listen B nach dem Zufallsprinzip umsortiert. Ich habe mir einige der 144 Permutationen genauer angeschaut, konnte jedoch keine Regelmäßigkeit finden. Leider ist es auch nicht möglich, alle Permutationen auszuprobieren, denn für $M = 5$ gibt es bereits 32! Permutationen. Wenn man weiter in diese Richtung geht, sind möglicherweise aber noch signifikante Optimierungen möglich.

1.5 Eigene Erweiterungen

1.5.1 Muster bei der Schalterauswahl

Aus Kostengründen kann es sinnvoll sein, die echten Schalter immer direkt nebeneinander zu platzieren. Das Programm soll nun einen Öffnungscode generieren, wenn bekannt ist, welches Muster die echten Schalter aufweisen. Das Muster *11111* sagt zum Beispiel aus, dass die fünf echten Schalter sich immer nebeneinander befinden müssen. Beim Muster *101010101* muss sich hingegen zwischen zwei echten Schaltern immer eine Attrappe befinden. Um das Problem etwas interessanter zu gestalten, sollen die Schalter zudem kreisförmig angeordnet sein.

Der eigentliche Algorithmus (der zweite Algorithmus) muss für diese Erweiterung nicht verändert werden, es genügt, die Prozedur zu verändern, die die Schalterauswahlen generiert. Es werden einfach alle möglichen Startpositionen für das Muster betrachtet (erster Schalter bis letzter Schalter). Dann wird für jede Startposition das Muster eingesetzt und die ausgewählten Schalter werden notiert. Dabei ist darauf zu achten, dass die Schalterindizes modulo Schalteranzahl angegeben werden, damit es bei einem *Umlauf*[18] nicht zu Problemen kommt. Die Anzahl der zu betrachtenden Fälle reduziert sich natürlich drastisch, weil es nur noch N Möglichkeiten gibt, M aus N Schaltern auszuwählen.

1.5.2 Mehrere Schalterzustände

Noch schwieriger wird das Problem, wenn es mehr als nur zwei Schalterzustände (0 und 1) gibt. Wenn es k Schalterzustände gibt, erhöht sich die Anzahl der Schalterstellungen von 2^M auf k^M. Der zweite Algorithmus muss dazu nicht großartig verändert werden.

[17] Alle diese Zahlen zusammen addieren sich zu 8!.

[18] D.h. wenn über die Schalterarraygrenzen iteriert werden soll.

Es genügt, die Liste der möglichen Schalterstellungen entsprechend anzupassen. Es muss jedoch beachtet werden, dass der dritte Schalterzustand nicht 2 sein darf, da der Zustand 2 bedeutet, dass der Schalter nicht verwendet wird. Stattdessen ist der dritte Schalterzustand -2. Diese Erweiterung ist sinnvoll, weil es dann noch schwieriger wird, einen Zugangscode zu erraten.

1.5.3 Ein Spiel: Wer kann besser raten?

Die Aufgabe soll nun grundlegend verändert werden. Auf Grund zahlreicher interner Hackerangriffe, wird in der Firma *BWINF* ein Wettbewerb gestartet, an dem die ganze Belegschaft teilnehmen darf. Um die kriminelle Energie der Mitarbeiter etwas abzubauen, gilt es, ein Sicherheitssystem mit N Schaltern, von denen M echt sind, zu knacken. Da jedoch einerseits das Eingeben eines universellen Öffnungscodes eine Zumutung ist, da es sich um mehrere hundert Bitstrings handeln kann, und andererseits nur möglichst wenig (Arbeits)zeit zur Verfügung steht, bekommt jeder Teilnehmer etwas Hilfe. Man kennt es aus Action-Filmen wie James Bond und co. Der Held steht vor einem großen Tresor mit mehreren großen Zahlenschlössern, und immer, wenn er ein Zahlenschloss in die richtige Position gebracht hat, klickt es. Dieses Prinzip soll auf den Wettbewerb angewandt werden. Nach jedem Angriffsversuch (bei jedem Ausprobieren eines Bitstrings) bekommt der Angreifer eine Mitteilung, wie viele Schalter in der richtigen Position sind. Es gewinnt der Angreifer, der mit möglichst wenigen Bitstring-Prüfungen auskommt. Dem Angreifer ist nicht bekannt, wie viele Schalter echt sind[19]. Die Auswahl der echten Schalter und die Stellung der echten Schalter erfolgt zufällig.

Ein einfacher Algorithmus Es soll nun ein einfacher Algorithmus vorgestellt werden. Der Angreifer probiert als Erstes einen Bitstring aus lauter Nullen aus, d.h. alle Schalter stehen auf Null. Er merkt sich, wie viele Schalter in der richtigen Position standen. Jetzt wird der erste Schalter verändert (auf 1 gestellt). Wird der Bitstring dadurch besser[20], bleibt der Schalter auf dieser Stellung, ansonsten wird der Schalter wieder auf 0 zurückgestellt. Dieses Verfahren wird der Reihe nach mit allen N Schaltern durchgeführt.

Der Algorithmus braucht immer $N + 1$ Bitstrings (Prüfungen), weil immer dann die Güte des Bitstring geprüft wird, wenn ein noch nicht untersuchtes Bit umgedreht wird. Wenn ein Schalter fälschlicherweise umgedreht wird, wird er einfach nochmals umgedreht und im selben Schritt wird gleich der nächste Schalter auf 1 gesetzt. Ganz am Anfang (wenn noch kein Bit umgedreht wurde), muss natürlich auch geprüft werden, damit es einen Vergleichswert für die Güte des Bitstrings gibt. Es handelt sich um einen Online-Algorithmus, weil der Algorithmus während der Laufzeit neue Informationen über das Problem erhält und diese bei seinen Entscheidungen verwendet.

Ein zweiter Algorithmus: Immer zwei Schalter wählen Es soll nun ein zweiter Algorithmus vorgestellt werden. Der Algorithmus dreht immer zwei Bits auf einmal um. Dabei

[19] Der Angreifer kennt M nicht, er kennt lediglich N.

[20] Ein Bitstring wird besser, wenn die Anzahl der Schalter, die sich in der richtigen Position befinden, größer wird.

werden immer die Bits ganz links und ganz rechts gewählt, die noch nicht umgedreht wurden. Es werden also immer ein Zeiger auf das Bit auf der linken Seite und ein Zeiger auf das Bit auf der rechten Seite gespeichert. Wenn die Zeiger sich überschneiden, also $Zeiger_{links} > Zeiger_{rechts}$, wurden alle Bits bearbeitet. Die beiden ausgewählten Bits (links und rechts) werden umgedreht, danach wird die Güte des Bitstrings berechnet. Der Bitstring kann sich um maximal zwei Einheiten verbessern oder verschlechtern, weil nur zwei Bits verändert wurden. Wenn der Bitstring sich um zwei Einheiten verbessert, bleibt die Änderung erhalten, denn das ist der beste Fall, der eintreten kann. Wenn sich der Bitstring nur um eine Einheit verbessert, bleibt die Änderung auch erhalten. In diesem Fall wurde ein Bit richtigerweise umgedreht und der andere Schalter ist eine Attrappe. Wäre er keine Attrappe, so hätte sich der Bitstring entweder um zwei Einheiten verbessert (wenn der Schalter in der richtigen Position steht) oder gar nicht verbessert, wenn der Schalter in der falschen Position steht[21]. Falls sich der Bitstring um eine Einheit verschlechtert, werden beide Schalter einfach wieder umgedreht. In diesem Fall wurde ein Schalter fälschlicherweise umgedreht und der andere ist eine Attrappe. Wäre er keine Attrappe, so hätte sich der Bitstring entweder um zwei Einheiten verschlechtert (wenn der Schalter in der falschen Position steht) oder gar nicht verschlechtert, wenn der Schalter in der richtigen Position steht[22]. Für den Fall, dass sich die Güte des Algorithmus nicht verändert hat, muss jeder Schalter einzeln betrachtet werden, weil nicht bekannt ist, ob es möglicherweise noch besser geht. Es kann schließlich sein, dass ein Bit richtig und das andere falsch gestellt wurde, wobei nicht bekannt ist, welches richtig und welches falsch ist. Oder es sind beide Schalter Attrappen. Wenn alle Schalter falsch gestellt wurden (Verschlechterung um 2), genügt es, die Änderung rückgängig zu machen und die Schalter müssen nicht einzeln betrachtet werden.

Die folgende Tabelle liefert einen Überblick über alle gerade genannten Möglichkeiten. Dabei werden immer die Verbesserung des Bitstrings und die Anzahl der notwendigen Bitstringauswertungen (Prüfungen) angegeben. Es ist immer mindestens eine Prüfung notwendig, weil schließlich jeder Bitstring einmal geprüft werden muss, um festzustellen, inwiefern er sich verbessert oder verschlechtert hat.

Δ (Verbesserung)	**Prüfungen**
2	1
1	1
0	3 (Jeder Schalter einzeln)
-1	1
-2	1

Bei Betrachtung der Tabelle sollte man beachten, dass immer zwei Schalter gleichzeitig geprüft werden. Wenn der Fall $\Delta = 0$ nur selten zutrifft, spart man sich Bitstrings.

Verbesserung des zweiten Algorithmus Der zweite Algorithmus kann noch etwas verbessert werden. Man betrachte den Fall $\Delta = 0$. In diesem Fall sind entweder beide

[21] Dann hat wurde ein Schalter richtig gestellt und einer wurde falsch gestellt. Im Vergleich zur vorherigen Situation ist das weder eine Verbesserung, noch eine Verschlechterung

[22] Dann wurde ein Schalter falsch gestellt und einer wurde richtig gestellt.

Schalter Attrappen oder einer ist in der richtigen Position, der andere in der falschen Position. Es muss an dieser Stelle nicht jeder der beiden Schalter einzeln betrachtet werden. Es genügt, den ersten der beiden Schalter zu betrachten, dieser wird verändert. Verbessert sich der Bitstring dadurch, braucht der zweite Schalter gar nicht mehr betrachtet werden. Verschlechtert sich der Bitstring, wird der Schalter wieder zurückgesetzt und der zweite Schalter muss verändert werden. Hat die Veränderung des Schalters keine Auswirkung auf die Güte des Bitstrings, sind beide Schalter Attrappen und müssen nicht weiter betrachtet werden. In jedem der drei Fälle ist nur eine Prüfung notwendig. Insgesamt reduziert sich die Anzahl der Prüfungen für den Fall $\Delta = 0$ dadurch auf 2.

Ein dritter Algorithmus: Immer mehrere Schalter wählen Der zweite Algorithmus kann so verändert werden, dass nicht nur zwei, sondern noch mehr Schalter ausgewählt werden. Dabei werden jedoch nicht immer die unbearbeiteten Schalter am Rand gewählt, stattdessen werden die Schalter zufällig ausgewählt[23]. Für den Fall, dass nur zwei Schalter ausgewählt werden sollen, gleicht der Algorithmus dem zweiten Algorithmus und für den Fall, dass nur ein Schalter ausgewählt werden soll, gleicht er dem ersten Algorithmus. Leider können keine weiteren größeren Optimierungen wie im zweiten Algorithmus vorgenommen werden. Wenn sich beispielweise bei zehn ausgewählten Schaltern der Bitstring um fünf Einheiten verbessert, kann nicht festgestellt werden, ob es sich um Attrappen-Schalter handelt und ob es evtl. noch besser geht. Ausführliche Ergebnisse gibt es im Kapitel *Programm-Ablaufprotokoll*.

Warum ist diese Erweiterung sinnvoll? Ich halte es durchaus für vorstellbar, dass sich ein schlecht konstruiertes Sicherheitssystem mit den nötigen technischen Hilfsmittel so oder auf ähnliche Art und Weise knacken lässt.

1.5.4 Sortieren der generierten Bitstrings

Angenommen, es wurde eine lange Liste mit Bitstrings generiert und der Einbrecher steht vor dem Sicherheitssystem. Er wird jeden Bitstring nacheinander eingeben. Das kann sehr lange dauern. Schneller geht es, wenn die Liste der Bitstrings so vorsortiert ist, dass die Kosten, die Anzahl der Schalterveränderungen, minimal sind. Eine Liste mit Bitstrings soll also so sortiert werden, dass bei der Eingabe der Bitstrings möglichst wenige Schalteränderungen notwendig sind.

Diese Aufgabe soll ein Greedy-Algorithmus übernehmen, von dem ich mir leider nicht sicher bin, ob er immer die optimale Bitstringanordnung liefert. In die Liste der sortierten Bitstrings wird zunächst ein beliebiger Bitstring eingefügt. Dieser Bitstring wird als *fertig* markiert. Jetzt werden alle restlichen (noch nicht fertigen) Bitstrings untersucht. Es wird der Bitstring ausgewählt, der die geringste Differenz (Anzahl der unterschiedlichen Schalterstellungen) aufweist. Verglichen wird immer mit dem ersten und dem letzten Element in der Liste der schon sortierten Elemente. Das beste Element wird dann am

[23] Das hätte man auch beim zweiten Algorithmus machen können und es ist einfach zu implementieren. Prinzipiell sollte es keinen Unterschied machen, weil der Öffnungscode zufällig generiert wird.

Anfang oder am Ende der Liste eingefügt und als *fertig* markiert. Der Vorgang wird so lange wiederholt, bis alle Bitstrings als *fertig* markiert worden sind.

Die Laufzeit ist im Worst-Case-Szenario quadratisch von der Anzahl der Bitstrings n abhängig, weil immer n Bitstrings hinzugefügt werden und bei jedem Hinzufügen maximal n Bitstrings untersucht werden müssen.

Die Schwierigkeit dieser Erweiterung lag darin, auf die Idee zu kommen, einen geeigneten Algorithmus zu finden und die Güte des Algorithmus zu bewerten (siehe *Programm-Ablaufprotokoll*).

Wie wird verfahren, wenn ein Schalter auf 2 steht? Eine Funktion *BitstringDifferenz* berechnet die Differenz (Anzahl der unterschiedlichen Schalterstellungen) zwischen zwei Bitstrings. Wenn der Schalter nur im ersten Bitstring auf *2* steht, soll das keine Kosten verursachen. Wenn der Schalter hingegen nur im zweiten Bitstring auf *2* steht, soll das Kosten verursachen. Wenn ein Schalter eine Stellung *0* oder *1* hat und im nächsten Bitstring die Stellung egal ist, verursacht das keine Kosten. Es kann jedoch Kosten verurachen, wenn der Schalter auf *0* oder *1* gestellt wird. Das Programm überprüft nicht, welche Stellung der Schalter zu diesem Zeitpunkt wirklich hat. Auf jeden Fall entstehen nur einmal Kosten für einen unbenutzten Schalter. Deshalb werden nur Kosten veranschlagt, wenn lediglich der zweite Schalter auf *2* steht.

1.5.5 Weitere Überlegungen

Man stelle sich vor, das System zum Authentisieren soll verschiedene Öffnungscodes akzeptieren, zum Beispiel weil es mehrere Benutzer mit eigenen Codes gibt. Es soll nun zwei Schalterstellungen geben, die Zugang gewähren. Man könnte $2M$ echte Schalter verwenden, also zum Beispiel 10 echte Schalter anstelle von 5 echten Schaltern. Dabei haben immer zwei echte Schalter die gleiche Wirkung. Sie werden mit einer *Oder*-Schaltung zusammengeschlossen: Es genügt, dass nur einer der beiden doppelten Schalter die richtige Stellung hat.

Beispiel 6 *Es gilt* $N = 9$ *und* $M = 3$*. Der Öffnungscode für das System laute*

$$(1, 0, 1, 2, 2, 2, 2, 2, 2) \tag{6}$$

Der Schalter 1 *sei mit Schalter* 6 *verbunden, Schalter* 2 *mit Schalter* 7 *und Schalter* 3 *mit Schalter* 8*. Dann ist auch der Code*

$$(2, 2, 2, 2, 2, 1, 0, 1, 2) \tag{7}$$

ein Öffnungscode. Gleichzeitig sind weitere Öffnungscodes nun möglich, z.B.

$$(1, 2, 2, 2, 2, 2, 0, 1, 2) \tag{8}$$

Es ist jetzt möglich, einen kürzeren universellen Öffnungscode zu generieren. Dazu verringert man N einfach um eins. Man beachtet also den letzten Schalter einfach nicht. Schlimmstensfalls wird einer der doppelten Schalter dadurch nicht beachtet. Das macht aber nichts, weil jeder Schalter zweimal vorhanden ist.

2 Programm-Dokumentation

Hinweis zu beiden Algorithmen Beide Algorithmen wurden so erweitert, dass zum Schluss überprüft wird, ob die generierte Bitstringliste gültig ist. Dazu werden einfach alle Fälle generiert (M aus N auf je 2^M Möglichkeiten setzen) und überprüft, ob in der Liste der Bitstrings ein Bitstring vorhanden ist, der für diesen Fall passt.

2.1 Ein einfacher Algorithmus

Dieser Algorithmus befindet sich in der Klasse *EinfacherAlgorithmus*. Es finden mehrere Durchläufe statt, wobei die Funktion *DerAlgo* mehrmals aufgerufen wird, diese Funktion enthält den eigentlichen Algorithmus.

Zuerst wird der Benutzer aufgefordert, die Anzahl der Schalter, die Anzahl der echten Schalter und die Anzahl der Durchläufe festzulegen. Dann beginnt der Algorithmus und es gibt keine Benutzerinteraktion mehr. Die Ergebnisse werden in der Konsole ausgegeben und in die Datei *algo.txt* gespeichert. Dort wird ebenfalls der kürzeste Öffnungscode abgelegt, der gefunden wurde.

Der erste Schritt ist das Erstellen einer Liste (*schalterIndexArray*) mit den Indizes aller Schalter. Aus dieser Liste werden dann (später) alle M aus N Elemente entnommen[24].

Der zweite Schritt ist das Erstellen einer Liste (*echteSchalterStellungen*) mit allen 2^M Möglichkeiten, M Schalter entweder auf 0 oder auf 1 zu setzten. Dazu wird eine Liste mit zwei Integer-Arrays der Größe eines Bitstrings (Anzahl der Schalter) erstellt. Bei einem Array ist der erste Wert 0, beim anderen 1. Nun werden alle Arrays aus der Liste entnommen, dupliziert und wieder eingefügt, wobei bei der ersten Kopie der zweite Wert 0 und bei der zweiten Kopie der zweite Wert 1 ist. Dieser Vorgang wird so lange wiederholt (natürlich dann für den dritten Wert usw.), bis alle 2^M Möglichkeiten generiert wurden.

Nun kann der eigentliche Algorithmus loslegen. Dieser wird mehrmals nacheinander ausgeführt und greift auf die zuvor erstellen Listen zurück. Es wird auf eine externe Klasse von *Adrian Akison* von *CodeProject.com* zurückgegriffen, die Aufgaben wie *Generiere alle Möglichkeiten, M aus N Elementen auszuwählen* löst. Diese Bibliothek wurde auch in *Aufgabe 2* verwendet. Es werden nun alle Möglichkeiten generiert, M Schalterindizes aus der Liste aller Schalterindizies auszuwählen. Die Liste mit allen diesen Möglichkeiten wird nun durchgemischt. Ebenso wird die Liste der 2^M Schalterstellungen durchgemischt. Dann wird für jede Schalterauswahl und jede Schalterstellungskonstellation eine Kombinationsmöglichkeit mit bereits existierenden Bitstrings (in der aktuellen Lösung) gesucht. Dazu wird die Liste der existierenden Bitstrings sortiert (Funktion *Einzelner-Code.CompareTo*). Es sollen die Bitstrings ganz oben sein, die möglichst wenig freie Stellen haben (keinen Platz verschwenden) und möglichst viele Übereinstimmungen mit dem aktuellen Fall haben. Letztere Bedingung wird dabei fünffach gewichtet[25]. Wenn

[24] Es werden alle Möglichkeiten generiert, M Indizes zu entnehmen. So werden alle Möglichkeiten generiert, M Schalter auszuwählen.

[25] Bei diesem Wert liefert der Algorithmus ganz brauchbare Ergebnisse.

kein passender Kombinationspartner gefunden wurde, wird der Bitstring einfach zur Liste der Bitstrings (*ListeDerCodes*) hinzugefügt. Nicht verwendete Schalter werden dabei auf 2 gesetzt. Zum Schluss wird vom Programm die beste Lösung ausgegeben.

Hinweise zum Quelltext In der Funktion *DerAlgo* beinhaltet *vergleichsSchalterIndizes* die aktuell ausgewählten Schalterindizes. Wenn also zum Beispiel auf den zweiten ausgewählten Schalter zugegriffen werden soll, erhält man den Index über *vergleichsSchalterIndizes[1]*. *vergleichsSchalterStellungen* enthält die Schalterstellungen der M Schalter, es handelt sich also um eine Liste mit genau M Elementen.

2.2 Ein zweiter Algorithmus

Der zweite Algorithmus befindet sich in der Klasse *ZweiterAlgorithmus*. Der eigentliche Algorithmus befindet sich in der Funktion *Start*.

Wenn das Programm gestartet wird, wird der Benutzer aufgefordert, die Problemdaten einzugeben (Anzahl der Schalter usw.). Die Prozedur *GeneriereAlleZugangscodes* ist für das Erstellen der Listen A (*listeSchalterAuswahl*) und B (*SchalterStellungen*) zuständig. Dazu wird wie im vorherigen Algorithmus vorgegangen, es wurde auch die gleiche externe Bibliothek verwendet. Große Teile des Quelltextes habe ich einfach kopiert. Eine erneute Erklärung ist deshalb nicht notwendig (siehe auch Kommentare im Quelltext).

Es finden mehrere Durchläufe des Algorithmus statt. In jedem Durchlauf werden zuerst die Listen A und B erstellt. Dann läuft eine *While*-Schleife so lange, bis die Liste A keine Elemente mehr enthählt. Dann wurden alle Fälle berücksichtigt und die Liste der Bitstrings ist auf jeden Fall gültig[26]. Es müssen nun kompatible Schalterauswahlen, also kompatible Elemente aus Liste A, gefunden werden. Dazu wird Liste A zunächst zufällig durchgemischt, dann wird über alle Einträge iteriert. Für jeden Eintrag wird überprüft, ob er kompatibel mit den bereits entnommenen Fällen ist. Falls das der Fall ist, wird das Element entnommen (und in der Liste *aktuelleReihe* gespeichert). Jetzt können die einzelnen Bitstrings zusammengesetzt werden. Der Wert z (Maximale Anzahl von Elementen in einer Liste B) entspricht der Variable *maximumReiheBitstrings*. Der Wert y (Anzahl der entnommenen Elemente aus Liste A) entspricht dem Ausdruck *aktuelleReihe.Count*. Wenn die Liste der Bitstrings (*neueBitstrings*) erstellt wurde, wird für alle Schalterauswahlen über alle Schalterstellungen iteriert, die noch nicht bearbeitet wurden (also noch in der Liste A bzw. B enthalten sind). Für jeden solchen Fall wird überprüft, ob er von einem der Bitstrings abgedeckt wurde bzw. kombiniert werden kann. Es kann möglicherweise auch kombiniert werden, wenn ein Bitstring eine undefinierte Stelle (2) aufweist. Alle Elemente der Liste *neueBitstrings* werden zum Schluss zur Liste *listeDerCodes* hinzugefügt. Ein neuer Durchlauf der While-Schleife findet statt, solange bis Liste A leer ist. Dann kann ein neuer Durchlauf des Programms starten, wobei die Listen hoffentlich besser durchgemischt werden und eine bessere Lösung erzielt wird.

[26] Es wurden jede Schalterauswahl und für jede Schalterauswahl jede Schalterstellung berücksichtigt.

Dateiausgaben Bei Ausführung des Programms wird eine Datei *algo_2.txt* erstellt, die alle Konsolenausgaben und noch etwas mehr Informationen enthält. Außerdem wird eine Debug-Datei *debug.txt* angelegt, der man entnehmen kann, wie viele Fälle pro Durchlauf der *While*-Schleife mitgeneriert wurden.

2.3 Eigene Erweiterungen

2.3.1 Muster bei der Schalterauswahl

Diese Erweiterung ist nur für den zweiten Algorithmus verfügbar. Für diese Erweiterung genügt es, die Prozedur *GeneriereAlleZugangscodes* zu verändern. Es wurde eine neue Prozedur *GeneriereAlleZugangscodesMuster* erstellt, die sich von der ürsprünglichen Version darin unterscheidet, dass die Auswahl der Schalter nicht durch die externe Bibliothek erfolgt. Stattdessen iteriert das Programm über alle möglichen Startindizes für das Muster, also über alle Schalterindizes. Falls der Benutzer angibt, dass er keinen *Umlauf* wünscht, werden ungültige Fälle, also Fälle, bei denen einen Schalterauswahl über die Arraygrenzen verläuft, automatisch aussortiert[27]. Es wird eine Liste mit Integer-Listen erstellt, wobei die äußere Liste alle Schalterauswahlen beinhaltet und jede innere Liste die Indizes der einzelnen Schalter. Diese Liste entspricht dem dem Ausdruck *combinations* in der Prozedur *GeneriereAlleZugangscodes*. Es sind keine weiteren Änderungen an der Prozedur notwendig, der restliche Quelltext kann einfach übernommen werden, da sich an den einzelnen Schalterstellungen nichts ändert.

2.3.2 Mehrere Schalterzustände

Auch diese Erweiterung ist nur für den zweiten Algorithmus verfügbar. Es genügt, die Prozedur *GeneriereSchalterStellungen* anzupassen. Der Algorithmus wurde so entworfen, dass diese Erweiterung ohne größeren Aufwand implementiert werden kann. Man sollte jedoch beachten, dass die Laufzeit mit steigender Anzahl an Schalterzuständen exponentiell zunimmt. Anstatt nur zwei Kopien der Integer-Arrays zu erstellen, werden bei n Schalterzuständen einfach n Kopien erstellt.

2.3.3 Ein Spiel: Wer kann besser raten?

Zunächst soll ein Öffnungscode für das Sicherheitssystem generiert werden. Die Soll-Stellungen der Schalter werden wieder in einem Int-Array gespeichert, wobei 2 einen unbenutzten Schalter andeutet. Nachdem der Benutzer die Werte N und M eingegeben hat, werden M Schalterindizes zufällig ausgewählt und für jeden ausgewählten Schalter wird zufallig 0 oder 1 vergeben. Die Funktion *schalterStellungBewerten* bewertet einen Bitstring. Je mehr Schalterstellungen übereinstimmen, desto höher ist der Wert. Es werden jetzt die einzelnen Algorithmen gestartet.

[27] Dann sind die Schalter nicht mehr kreisförmig angeordnet.

Ein einfacher Algorithmus Dieser Algorithmus befindet sich in der Funktion *einfacherAlgo*. Das Int-Array *aktuellerVersuch* speichert für jeden Schalter die Stellung im aktuellen Versuch (Bitstring). Es wird einmal über das gesamte Schalter-Array iteriert, jeder Schalter wird invertiert, und dann wird untersucht, ob die Lösung dadurch besser oder schlechter geworden ist. Es werden zwei Werte *situationVorher* und *situationNachher* verglichen, die die Anzahl der übereinstimmenden Schalter vor und nach der Invertierung des Schalters angeben. Die Anzahl der Prüfungen wird gespeichert und vom Algorithmus zurückgegeben. Zum Schluss wird sicherheitshalber überprüft, ob der Algorithmus einen richtigen Bitstring gefunden hat.

Ein zweiter Algorithmus: Immer zwei Schalter wählen Dieser Algorithmus hat viele Ähnlichkeiten mit dem vorherigen Algorithmus. Es werden zwei Zeiger *positionLinks* und *positionRechts* gespeichert, die auf die Schalter am linken und rechten Rand zeigen, die invertiert werden. In jedem Durchlauf der While-Schleife wandert der linke Zeiger um eine Einheit nach rechts und rechte Zeiger um eine Einheit nach links. Wenn die Zeiger sich überschneiden, kann abgebrochen werden, weil dann alle Schalter bearbeitet wurden. Man könnte die Schalter auch zufällig auswählen, jedoch sind die Ergebnisse dann nicht so schön reproduzierbar (da nicht deterministisch[28]). Es werden in jedem Durchlauf der While-Schleife die verschiedenen Möglichkeiten aus der Lösungsidee geprüft. Wenn sich die Güte des Bitstrings nicht verändert, werden beide Schalterindizes in eine Liste *zuUntersuchendeIndizes* hinzugefügt. Diese Liste wird zum Schluss verarbeitet, indem das Prinzip aus dem vorherigen Algorithmus angewendet wird. Die Schalter werden nacheinander invertiert und jedes Mal wird geprüft, wie sich die Güte des Bitstrings verändert. Auch dieses Mal wird der Bitstring zum Schluss sicherheitshalber auf Richtigkeit überprüft.

Verbesserung des zweiten Algorithmus Wenn die Funktion *zweiterAlgo* mit dem Parameter *verbessert=true* aufgerufen wird, wird bei der Verarbeitung der Liste *zuUntersuchendeIndizes* unterschiedlich verfahren. Es wird das erste Element entnommen und der Schalter wird invertiert. Abhängig von der Veränderung der Güte des Bitstrings wird nun der nächste Schalter in der Liste ebenfalls invertiert oder die Invertierung des aktuellen Schalters rückgängig gemacht (oder gar nichts gemacht). Beide Schalterindizes werden aus der Liste gelöscht. Es muss also nur die Hälfte der Einträge in der Liste verarbeitet werden. Dieses Verfahren ist möglich, weil die beiden Elemente eines zusammengehörigen Indizespaar[29] immer direkt aufeinander folgen.

Ein dritter Algorithmus: Immer mehrere Schalter wählen Dieser Algorithmus hat große Ähnlichkeit mit dem zweiten Algorithmus. Jedoch erhält er als Paramter die Anzahl der Schalter, die auf einmal invertiert werden sollen. In jedem Durchlauf der While-Schleife werden die Indizes der Schalter zufällig ausgewählt. Die Liste der im

[28] Aber nur, wenn der Bitstring manuell in Quelltext eingegeben wird.
[29] Ein Indizespaar ist das Paar von Schaltern, auf das der Fall $\Delta = 0$ zutrifft.

Anschluss noch einzeln zu prüfenden Schalter heißt *nachtraeglichZuUntersuchendeIndizes*. Diese werden wie im (nicht verbesserten) zweiten Algorithmus verarbeitet. Es ist nicht möglich[30], den Algorithmus analog zum zweiten Algorithmus zu verbessern, weil es viel mehr Fälle geben kann, dass eine Invertierung der Schalter keine Auswirkung auf die Güte des Algorithmus hat[31] (weil viel mehr Schalter auf einmal verändert werden dürfen). Auch dieses Mal wird der Bitstring wieder auf Richtigkeit überprüft.

2.3.4 Sortieren der generierten Bitstrings

Der Algorithmus befindet sich in der Klasse *BitstringsSortieren*. Als Argument erhält der Algorithmus eine Liste mit Bitstrings (Integer-Arrays). Diese Liste wird dupliziert (Wertekopie). Dies ist notwendig, weil ein Bitstring dann als *fertig* markiert ist, wenn er aus der Liste der noch zu bearbeitenden Bitstrings gelöscht wurde (die übergebene Liste soll nicht verändert werden). Eine While-Schleife läuft so lange, bis alle Bitstrings verarbeitet wurden. In jedem Durchlauf der Schleife wird ein Bitstring verarbeitet. Dazu wird eine For-Schleife verwendet, die jeden noch nicht *fertigen* Bitstring betrachtet und die Kosten berechnet, die entstehen würden, wenn der Bitstring am Anfang oder am Ende der sortierten eingefügt werden würde. Der beste Bitstring wird ausgewählt und an der entsprechenden Stelle in der sortierten Liste eingefügt (und als *fertig* markiert). Zum Schluss wird die sortierte Liste auf der Konsole ausgegeben.

3 Programm-Ablaufprotokoll

Hinweis zu den Laufzeiten Es wurden immer mehrere Instanzen des Programmes gleichzeitig gestartet, damit das Generieren aller Testfälle schneller geht. Das hat natürlich Einfluss auf die Laufzeiten (Laufzeiten werden größer). Es geht nur die groben Laufzeitverhältnisse. Die Ergebnisse wurden auf einem *Intel i7-860 (2,80GHz, 4 Kerne)* produziert.

Hinweis zu den Binärdateien Das Programm wurde in C# .NET geschrieben und ist deshalb nur unter Windows mit installiertem .NET Framework 3.5 lauffähig. Es handelt sich um *x86* Binärdateien.

Nutzungshinweise Es ist wichtig, dass keine ungültigen Eingaben in der Programmkonsole gemacht werden. Das Programm stürzt sonst einfach ab! Es werden keine Exceptions abgefangen und die Eingaben des Benutzers werden nicht auf Richtigkeit/Gültigkeit überprüft. Ich habe mich hier wirklich nur auf das Wesentliche konzentriert[32]. Ansonsten stellen die Laufzeitkomplexitäten die einzigen ernst zu nehmenden Nutzungsgrenzen dar. Es sollten keine Integer-Variablen überlaufen, ebenso sollte es nicht zu Speichermangel kommen, da das Programm nicht viel Arbeitsspeicher benötigt. Es kann

[30] Korrektur: Vermutlich nicht möglich, vllt. habe ich auch nur nicht lang genug nachgedacht.
[31] Das Problem ergibt sich übrigends nicht nur für $\Delta = 0$.
[32] Schuld daran war wohl hauptsächlich Zeitmangel.

lediglich sein, dass ein Programm sehr, sehr lange zur Berechnung braucht. Bei allen Beispielfällen habe ich versucht, an die Grenzen zu gehen, damit klar wird, bis zu welcher Problemgröße das Programm verwendbar ist.

3.1 Ein einfacher Algorithmus

Mit Sortieren Die folgende Tabelle zeigt Ergebnisse und Laufzeiten des einfachen Algorithmus mit Sortieren.

N	M	Durchläufe	min	∅	max	Laufzeit	∅ Laufzeit
6	3	10	14	15,00	16	8,0 ms	8,7 ms
6	3	100	12	14,94	18	391,0 ms	3,9 ms
6	3	1000	12	14,66	18	3243,2 ms	3,2 ms
6	4	10	26	27,80	30	169,0 ms	16,9 ms
6	4	100	25	28,36	32	1104,1 ms	11,0 ms
6	4	1000	24	28,32	34	10344,6 ms	10,3 ms
6	5	10	40	45,10	49	185,0 ms	18,5 ms
6	5	100	36	46,01	52	1491,1 ms	14,9 ms
6	5	1000	34	46,39	54	14415,8 ms	14,4 ms
6	6	10	64	64,00	64	122,0 ms	12,2 ms
6	6	100	64	64,00	64	488,0 ms	4,9 ms
6	6	1000	64	64,00	64	4089,2 ms	4,1 ms
8	4	10	35	37,00	39	873,0 ms	87,3 ms
8	4	100	34	37,55	41	8875,5 ms	88,8 ms
8	4	1000	33	37,6	43	87182,0 ms	87,2 ms
9	3	10	17	19,10	21	265,0 ms	26,5 ms
9	3	100	16	18,60	21	2456,1 ms	24,6 ms
9	3	1000	16	18,79	22	23858,4 ms	23,9 ms
9	3	10000	15	18,72	23	329279,8 ms	32,9 ms
18	4	10	67	68,00	71	181629,4 ms	18162,9 ms
18	4	100	65	68,38	70	1895385,4 ms	18953,9 ms
18	4	1000	64	68,29	72	16537842,9 ms	16537,8 ms
25	5	2	206	206,5	207	5781473,7 ms	2890736,8 ms

Wie man deutlich erkennen kann, ist die Laufzeit nicht besonders gut, für $M = 5$ und $N = 25$ ist die Berechnung gerade noch möglich. Das liegt daran, dass die Anzahl der Möglichkeiten exponentiell ansteigt (oder vllt. sogar schneller). Schließlich gilt es $\binom{N}{M} 2^M$ Möglichkeiten zu beachten. Nicht nur der Faktor 2^M ist schlecht, es gilt ebenfalls $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Für alle diese Möglichkeiten muss schließlich noch eine Liste mit allen bisherigen Bitstrings sortiert werden (Laufzeit $n \log n$).

Ohne Sortieren Spart man sich das Sortieren, ist der Algorithmus wesentlich schneller.

N	M	Durchläufe	min	∅	max	Laufzeit	∅ Laufzeit
6	3	100	12	14,70	18	38,0 ms	0,4 ms
6	3	1000	12	14,8	20	215,0 ms	0,2 ms
6	3	10000	12	14,7	20	1689,1 ms	0,2 ms
6	4	100	26	28,49	33	71,0 ms	0,7 ms
6	4	1000	24	28,31	34	415,0 ms	0,4 ms
6	4	10000	24	28,38	36	4189,2 ms	0,4 ms
6	5	100	36	46,37	52	99,0 ms	1,0 ms
6	5	1000	36	46,41	54	603,0 ms	0,6 ms
6	5	10000	34	46,53	54	5625,3 ms	0,6 ms
6	6	100	64	64,00	64	100,0 ms	1,0 ms
6	6	1000	64	64,00	64	348,0 ms	0,3 ms
6	6	10000	64	64,00	64	2973,2 ms	0,3 ms
8	4	100	34	37,65	43	244,0 ms	2,4 ms
8	4	1000	33	37,76	43	2081,1 ms	2,1 ms
8	4	10000	32	37,81	45	20259,16 ms	2,0 ms
9	3	100	17	18,95	21	114,0 ms	1,1 ms
9	3	1000	16	18,80	22	806,0 ms	0,8 ms
9	3	10000	16	18,82	24	7019,4 ms	0,7 ms
9	3	100000	15	18,84	24	75438,3 ms	0,8 ms
9	3	1000000	13	18,83	24	756359,3 ms	0,8 ms
18	4	100	66	69,08	73	10597,6 ms	106,0 ms
18	4	1000	65	68,99	74	106073,1 ms	106,1 ms
18	4	10000	63	68,97	75	1020861,4 ms	102,1 ms
18	4	100000	64	68,97	75	9033351,7 ms	90,3 ms
25	5	100	204	207,52	213	719540,2 ms	7195,4 ms
25	5	1000	202	208,05	214	6365494,1 ms	6365,5 ms

Es fällt auf, dass der Algorithmus ohne Sortieren viel schneller ist. Dafür liefert er nicht ganz so gute Ergebnisse, wie der Algorithmus mit Sortieren. Das kann man jedoch dadurch ausgleichen, dass man den Algorithmus einfach noch öfter nacheinander laufen lässt (mehr Durchläufe). Dann kommt man wieder auf ähnliche und zum Teil bessere Ergebnisse (bei ähnlicher Laufzeit).

Ein zweiter Algorithmus Der zweite Algorithmus liefert ähnliche Ergebnisse.

N	M	Durchläufe	min	∅	max	Laufzeit	∅ Laufzeit
6	3	100	12	14,69	17	468,0 ms	4,7 ms
6	4	100	25	28,17	32	607,0 ms	6,1 ms
6	5	100	40	46,95	52	391,0 ms	3,9 ms
6	5	1000	34	46,57	53	3730,2 ms	3,7 ms
6	6	100	64	64,00	64	163,0 ms	1,6 ms
8	4	100	34	38,04	42	1451,1 ms	14,5 ms
8	4	1000	34	38,25	43	14618,8 ms	14,6 ms
8	4	10000	33	38,18	44	157314,0 ms	15,7 ms
9	3	100	17	18,98	21	1035,1 ms	10,4 ms
9	3	1000	16	19,08	23	11509,7 ms	11,5 ms
9	3	10000	16	19,05	23	99460,7 ms	9,9 ms
18	4	100	67	70,44	74	39469,3 ms	394,7 ms
18	4	1000	66	70,45	75	391460,4 ms	391,5 ms
25	5	10	211	213,70	216	277364,9 ms	27736,5 ms
25	5	50	209	213,04	216	1231607,4 ms	24632,1 ms

Wenn man die Ergebnisse mit dem einfachen Algorithmus vergleicht, fällt auf, dass der zweite Algorithmus zwar schneller als der einfache Algorithmus mit Sortieren ist, wobei etwas mehr Bitstrings benötigt werden. Der einfache Algorithmus ohne Sortieren liefert jedoch bessere Ergebnisse bei besserer Laufzeit.

Der zweite Algorithmus hat eine ähnliche Laufzeitkomplexität. Bei einem Durchlauf der While-Schleife, wird zwar für alle Schalterauswahlen über alle Schalterstellungen iteriert ($\left(\begin{array}{c} N \\ M \end{array}\right) 2^M$ Möglichkeiten), jedoch wird die Menge der noch nicht bearbeiteten Fälle mit jedem Durchlauf kleiner. Während der erste Durchlauf noch relativ lange dauert, halbiert sich die Zeit für den zweiten Durchlauf (oder sogar noch schneller), weil so viele Fälle mitgeneriert wurden. Es wird eine Debug-Datei bei der Ausführung des Programms angelegt. Bei einem Test mit $N = 25$ und $M = 5$ wurden im ersten Durchlauf der While-Schleife bereits 1156151 Fälle mitgeneriert (insgesamt gibt es $\left(\begin{array}{c} 25 \\ 5 \end{array}\right) 2^5 = 1700160$ Fälle). Die Anzahl dieser mitgenierten Fälle nimmt dann mit jedem Schleifendurchlauf ab (zweiter Durchlauf: 185559, dritter Durchlauf: 106501, vierter Durchlauf: 55857, ...).

3.2 Eigene Erweiterungen

3.2.1 Muster bei der Schalterauswahl

Diese Erweiterung wurde nur für den zweiten Algorithmus implementiert (obwohl sie durchaus auch für den ersten Algorithmus möglich wäre). Dadurch, dass nur bestimmte Muster erlaubt sind, reduziert sich die Anzahl der zu beachtenden Fälle enorm. Hier einige Beispiele.

N = 9 und M = 3 mit Muster 111 und 1000 Durchläufen, Umläufe sind erlaubt .

```
Bester Aufruf mit 8 Elementen.
Schlechtester Aufruf mit 14 Elementen.
Durschnittlich mit 11,29 Elementen.
Vergangene Zeit: 2750,1573ms.
Durchschnittliche Zeit pro Durchlauf: 2,7501573ms.
{1;1;1;1;1;1;0;1;1;}
{0;0;1;1;0;0;1;0;0;}
{0;1;1;0;1;0;1;1;0;}
{0;0;0;0;1;1;1;1;1;}
{1;1;0;0;0;0;0;1;0;}
{1;0;0;1;1;0;0;0;0;}
{0;1;0;1;0;1;0;0;1;}
{1;0;1;0;0;1;1;0;1;}

Sortierter Code:
{1;0;1;0;0;1;1;0;1;}
{0;0;0;0;1;1;1;1;1;}
{0;1;1;0;1;0;1;1;0;}
{0;0;1;1;0;0;1;0;0;}
{0;1;0;1;0;1;0;0;1;}
{1;1;1;1;1;1;0;1;1;}
{1;1;0;0;0;0;0;1;0;}
{1;0;0;1;1;0;0;0;0;}

Durch das Sortieren wurden Kosten in Höhe von 5 eingespart.
Der Code ist gültig.
```

N = 9 und M = 3 mit Muster 10101 und 1000 Durchläufen, Umläufe sind erlaubt .

```
Bester Aufruf mit 8 Elementen.
Schlechtester Aufruf mit 14 Elementen.
Durschnittlich mit 11,343 Elementen.
Vergangene Zeit: 2501,143ms.
Durchschnittliche Zeit pro Durchlauf: 2,501143ms.

[...]

Sortierter Code:
{1;0;1;0;0;0;1;1;1;}
{1;1;1;0;1;0;1;0;1;}
{1;0;0;0;1;1;1;0;0;}
{0;1;0;0;0;1;1;1;0;}
{1;1;0;1;0;1;0;1;1;}
{0;1;1;1;1;0;0;1;0;}
{0;0;1;1;0;1;0;0;0;}
{0;0;0;1;1;0;0;0;1;}
Durch das Sortieren wurden Kosten in Höhe von 11 eingespart.
Der Code ist gültig.
```

N = 9 und M = 3 mit Muster 111000000 und 1000 Durchläufen, Umläufe sind erlaubt .

```
Bester Aufruf mit 8 Elementen.
Schlechtester Aufruf mit 14 Elementen.
Durschnittlich mit 11,364 Elementen.
Vergangene Zeit: 2389,1367ms.
Durchschnittliche Zeit pro Durchlauf: 2,3891367ms.

[...]

Sortierter Code:
```

```
{1;0;1;0;0;0;0;0;1;}
{1;1;1;1;0;0;1;1;1;}
{1;1;0;1;1;0;1;0;0;}
{0;0;1;1;1;1;1;0;1;}
{0;0;0;1;0;1;0;0;0;}
{1;0;0;0;0;1;1;1;0;}
{0;1;0;0;1;1;0;1;1;}
{0;1;1;0;1;0;0;1;0;}
Durch das Sortieren wurden Kosten in Höhe von 7 eingespart.
Der Code ist gültig.
```

N = 9 und M = 3 mit Muster 111000000 und 1000 Durchläufen, Umläufe sind nicht erlaubt Bei diesem Beispiel ist es vollkommen richtig, dass die letzten sechs Schalter nicht benötigt werden. Es ist kein Umlauf erlaubt und weil das Muster genau neun Zeichen lang ist, gibt es nur eine Möglichkeit, die Schalter auszuwählen. In dieser Möglichkeit sind lediglich die ersten drei Schalter eingeschlossen.

```
Bester Aufruf mit 8 Elementen.
Schlechtester Aufruf mit 8 Elementen.
Durschnittlich mit 8 Elementen.
Vergangene Zeit: 843,0482ms.
Durchschnittliche Zeit pro Durchlauf: 0,8430482ms.

{0;0;1;2;2;2;2;2;2;}
{0;1;0;2;2;2;2;2;2;}
{1;1;0;2;2;2;2;2;2;}
{0;1;1;2;2;2;2;2;2;}
{1;1;1;2;2;2;2;2;2;}
{1;0;0;2;2;2;2;2;2;}
{0;0;0;2;2;2;2;2;2;}
{1;0;1;2;2;2;2;2;2;}

Sortierter Code:
{1;0;1;2;2;2;2;2;2;}
{1;1;1;2;2;2;2;2;2;}
{1;1;0;2;2;2;2;2;2;}
{0;1;0;2;2;2;2;2;2;}
{0;1;1;2;2;2;2;2;2;}
{0;0;1;2;2;2;2;2;2;}
{0;0;0;2;2;2;2;2;2;}
{1;0;0;2;2;2;2;2;2;}

Durch das Sortieren wurden Kosten in Höhe von 4 eingespart.
Der Code ist gültig.
```

N = 9 und M = 3 mit Muster 10101 und 1000 Durchläufen, Umläufe sind nicht erlaubt .

```
Bester Aufruf mit 8 Elementen.
Schlechtester Aufruf mit 14 Elementen.
Durschnittlich mit 11,343 Elementen.
Vergangene Zeit: 2501,143ms.
Durchschnittliche Zeit pro Durchlauf: 2,501143ms.

[...]

Sortierter Code:
{1;0;1;0;0;0;1;1;1;}
{1;1;1;0;1;0;1;0;1;}
{1;0;0;0;1;1;1;0;0;}
{0;1;0;0;0;1;1;1;0;}
```

```
{1;1;0;1;0;1;0;1;1;}
{0;1;1;1;1;0;0;1;0;}
{0;0;1;1;0;1;0;0;0;}
{0;0;0;1;1;0;0;0;1;}

Durch das Sortieren wurden Kosten in Höhe von 11 eingespart.
Der Code ist gültig.
```

N = 9 und M = 3 mit Muster 111 und 1000 Durchläufen, Umläufe sind nicht erlaubt .

```
Bester Aufruf mit 8 Elementen.
Schlechtester Aufruf mit 14 Elementen.
Durschnittlich mit 10,638 Elementen.
Vergangene Zeit: 2128,1217ms.
Durchschnittliche Zeit pro Durchlauf: 2,1281217ms.

[...]

Sortierter Code:
{1;1;0;0;0;1;0;1;0;}
{1;1;1;0;0;0;0;1;1;}
{1;0;1;0;1;0;1;1;1;}
{1;0;0;0;1;1;1;0;1;}
{0;0;0;1;1;1;0;0;0;}
{0;1;1;1;1;0;0;0;1;}
{0;1;0;1;0;0;1;0;0;}
{0;0;1;1;0;1;1;1;0;}

Durch das Sortieren wurden Kosten in Höhe von 15 eingespart.
Der Code ist gültig.
```

3.2.2 Mehrere Schalterzustände

Auch diese Erweiterung wurde nur für den zweiten Algorithmus implementiert. Sie kann auch mit den Erweiterungen *Bitstringliste sortieren* und *Muster bei der Schalterauswahl* kombiniert werden. Es folgen ein paar Beispielfälle, aus denen klar werden sollte, wie stark die Anzahl der Bitstrings steigt, wenn die Anzahl der Schalterzustände nur leicht angehoben wird. Das liegt daran, dass die Anzahl der zu betrachtenden Fälle exponentiell von der Anzahl der Schalterzustände abhängt.

N	**M**	**Zustände**	**Durchläufe**	**min**	**∅**	**max**	**Laufzeit**	**∅ Laufzeit**
6	3	2	1000	12	14,55	20	4339,2 ms	4,3 ms
6	3	2	10000	12	14,55	20	47423,7 ms	4,7 ms
6	3	3	1000	51	55,85	63	8976,5 ms	9,0 ms
6	3	3	10000	50	55,93	64	92653,3 ms	9,3 ms
6	3	4	1000	128	139,90	149	13458,8 ms	13,5 ms
6	3	4	10000	129	139,78	152	137498,9 ms	13,7 ms
6	3	5	1000	266	281,79	299	26163,50 ms	26,2 ms
6	3	5	10000	266	281,81	301	263998,1 ms	26,4 ms
9	3	2	1000	16	19,03	23	9732,6 ms	9,7 ms
9	3	3	1000	67	73,41	80	25710,5 ms	25,7 ms
18	4	2	100	67	70,44	73	22035,3 ms	220,4 ms
18	4	3	100	417	424,25	429	261823,0 ms	2618,2 ms

Beispielfall für N = 6 und M = 3 mit 3 Zuständen Weil der Schalterzustand 2 bereits für einen nicht verwendeten Schalter reserviert ist, wird für den dritten Schalterzustand -2 verwendet.

```
Bester Aufruf mit 51 Elementen.
Schlechtester Aufruf mit 63 Elementen.
Durschnittlich mit 55,854 Elementen.
Vergangene Zeit: 8976,5135ms.
Durchschnittliche Zeit pro Durchlauf: 8,9765135ms.

[...]

Sortierter Code:
{1;0;1;0;1;1;}
{1;0;1;2;0;2;}
{0;0;1;-2;0;0;}
{1;1;0;-2;1;0;}
{0;1;0;0;1;-2;}
{0;1;-2;0;1;1;}
{-2;1;-2;1;1;-2;}
{1;1;-2;1;0;1;}
{0;1;0;1;0;1;}
{0;1;1;0;0;1;}
{0;1;1;1;1;1;}
{-2;1;1;-2;1;1;}
{-2;1;-2;-2;-2;0;}
{-2;1;-2;0;0;0;}
{-2;1;0;0;-2;0;}
{1;1;0;0;-2;1;}
{1;0;-2;0;-2;1;}
{1;1;-2;0;-2;-2;}
{1;1;1;1;-2;-2;}
{1;-2;1;1;-2;0;}
{0;-2;-2;1;-2;0;}
{0;1;1;1;-2;0;}
{0;0;1;1;-2;1;}
{0;-2;1;1;0;-2;}
{-2;0;1;1;0;-2;}
{-2;0;-2;1;0;0;}
{-2;0;-2;-2;1;0;}
{1;0;-2;1;1;0;}
{1;0;0;1;1;-2;}
{1;0;0;-2;-2;-2;}
{1;1;0;-2;0;-2;}
{0;1;-2;-2;0;-2;}
{0;0;-2;0;0;-2;}
{0;0;-2;0;1;-2;}
{-2;0;1;0;1;-2;}
{-2;-2;1;0;1;0;}
{-2;-2;0;0;1;-2;}
{-2;-2;0;1;1;1;}
{-2;-2;0;0;0;1;}
{1;-2;0;0;0;0;}
{0;-2;0;-2;1;0;}
{0;-2;0;-2;-2;1;}
{1;-2;-2;-2;1;1;}
{1;-2;1;-2;1;-2;}
{-2;-2;1;-2;-2;-2;}
{-2;-2;1;0;-2;1;}
{0;-2;-2;0;-2;-2;}
{0;0;0;0;-2;0;}
{-2;0;0;1;-2;0;}
{-2;0;0;-2;0;1;}
{-2;-2;-2;-2;0;1;}

Kosten der unsortierten Liste: 197.
```

```
63 Kosten der sortierten Liste: 111.
64 Durch das Sortieren wurden Kosten in Höhe von 86 eingespart.
65 Der Code ist gültig.
```

3.2.3 Ein Spiel: Wer kann besser raten?

Das Programm wurde so verändert, dass beliebig viele Durchläufe stattfinden können. Bei jedem Durchlauf wird ein neuer Öffnungscode zufällig generiert. Zum Schluss wird der Durchschnittswert der Anzahl der benötigten Bitstrings berechnet.

Die folgende Tabelle enthält Ausgaben für den einfachen und den zweiten Algorithmus. Es kann verglichen werden, mit wie vielen Bitstrings die einzelnen Algorithmen auskommen. Die Algorithmen wurden immer mehrmals hintereinander ausgeführt (Durchläufe), auf verschiedenen Öffnungscodes. Alle angegebenen Werte sind Durchschnittswerte.

N	**M**	**Durchläufe**	**einf. Algo.**	**zw. Algo.**	**zw. Algo. (verbessert)**
26	0	10000	27	40,00	27,00
26	1	10000	27	38,00	26,00
26	5	10000	27	31,23	22,61
26	10	10000	27	25,39	19,70
26	15	10000	27	22,60	18,30
26	20	10000	27	22,80	18,40
26	26	10000	27	26,83	20,41
500	50	1000	501	658,24	454,62
500	100	1000	501	580,39	415,69
500	150	1000	501	517,86	384,43
500	200	1000	501	470,87	360,93
500	250	1000	501	438,60	344,80
500	300	1000	501	419,64	335,32
500	350	1000	501	418,11	334,56
500	400	1000	501	430,39	340,69
500	450	1000	501	458,66	354,83
500	500	1000	501	500,81	375,91

Das folgende Diagramm zeigt die Anzahl der benötigten Bitstrings bei Verwendung des dritten Algorithmus. Auf die horizontale Achse ist der Parameter des Algorithmus (Anzahl der pro Schritt ausgewählten Schalter) aufgetragen, auf die vertikale Achse ist die Anzahl der in diesem Fall benötigten Bitstrings aufgetragen. Es gilt $N = 26$ und bei allen im Diagramm verwendeten Werten handelt es sich um Durchschnittswerte (1000 Durchläufe). Die einzelnen Linien stehen jeweils für einen bestimmen Wert M (Anzahl der echten Schalter), die dazugehörige Legende befindet sich auf der rechten Seite.

Wie man dem Diagramm entnehmen kann, werden bei *parameter=1* genau so viele Bitstrings benötigt, wie beim einfachen Algorithmus. Das liegt daran, dass - wie beim einfachen Algorithmus - immer nur ein Schalter verändert wird, und dann geschaut wird, wie sich die Güte des Bitstrings verändert. Ähnlich verhält es sich mit *parameter=2*, es werden ähnlich viele Bitstrings wie beim zweiten (nicht verbesserten) Algorithmus benötigt. Noch größere Parameterwerte lohnen sich nicht. Wenn man das Diagramm

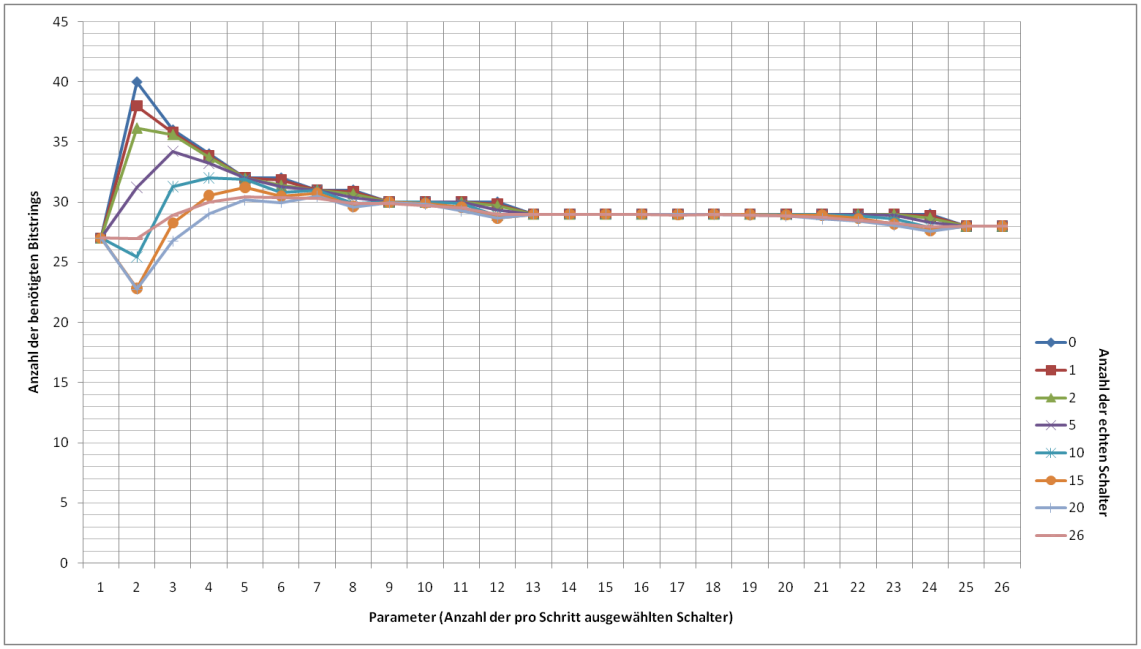

Abbildung 1: Der dritte Algorithmus: Anzahl der benötigten Bitstrings

mit vorheriger Tabelle vergleicht, stellt man fest, dass es sich überhaupt nicht lohnt, den dritten Algorithmus zu verwenden (was meiner Meinung nach nicht von Anfang an klar war). Man sollte stattdessen immer den zweiten (verbesserten) Algorithmus verwenden.

Weitere nebensächliche Anmerkungen zum Diagramm Es fällt auf, dass es bei *parameter=2* und $M = 0$ einen deutlichen Ausschlag nach oben gibt. Dann nehmen der Werte mit zunehmendem Parameterwert ab. Das liegt daran, dass bei einem Parameterwert von Zwei $\frac{26}{2} + 26 = 39$ Fälle betrachtet werden müssen. Der Bruch ergibt sich aus der Tatsache, dass dreizehn Mal zwei Schalter auf einmal betrachtet werden und der Summand 26 rührt daher, dass jeder der 26 Schalter noch einmal einzeln betrachtet werden muss[33]. Wenn hingegen 26 Schalter auf einmal betrachtet werden, sind nach analoger Rechnung nur $1 + 26$ Fälle zu betrachten. Das ist natürlich immer noch viel zu viel.

3.2.4 Sortieren der generierten Bitstrings

Wenn der einfache Algorithmus oder der zweite Algorithmus ausgeführt werden, wird die Liste der generierten Bitstrings zum Schluss sortiert ausgegeben. Auf der CD-ROM enthalten alle Programmausgaben eine unsortierte und eine sortierte Liste mit Bitstrings. Zusätzlich habe ich mir folgendes Prinzip überlegt, um zu testen, wie gut der Algorithmus arbeitet.

Es gibt 2^N Möglichkeiten, N Schalter (unterschiedlich) zu setzten. Intuitiv sollte klar

[33] Weil alle Schalter Attrappen sind, was jedoch anfangs nicht bekannt ist.

sein, dass die Kosten der besten Anordnung dieser 2^N Möglichkeiten mindestens 2^N betragen müssen, weil sich jede Schalterstellung von der Nächsten um mindestens 1 unterscheiden muss (sonst gäbe es doppelte Schalterstellungen). Es wurde ein Programm geschrieben, das n Bitstrings der Länge k generiert. Die Bits dieser Bitstrings werden zufällig gesetzt. Die Liste der Bitstrings wird sortiert und die Kosten der sortierten Liste werden ausgegeben. Je näher sich dieser Kostenwert an 2^k befindet, desto besser. Der Wert 2^k kann meiner Meinung nach auch dann erreicht werden, wenn manche Schalterstellungen fehlen (da zufällig generiert). Mehrfach generierte Schalterstellungen erzeugen keine weiteren Kosten. Angenommen, es liegt eine perfekt generierte Schalterstellung mit den Kosten 2^N vor, wobei jede Schalterstellung genau einmal vorkommt. Jetzt wird eine beliebige Schalterstellung entfernt. Dann ändern sich die Kosten der gesamten Listenanordnung in der Regel nicht (zumindest können sie sich nicht verschlechtern), weil bei der Schalterstellung nach der gelöschten Schalterstellung nur die Kosten 2 (zuvor zweimal 1) auftreten. Bei jeder beliebigen Wahl von k sollten die Kosten der Listenanordnung möglichst nahe an 2^N liegen, dann ist der Algorithmus gut.

Die folgende Tabelle zeigt für verschiedene Werte von k und N die Kosten der generierten Anordnung. k sollte dabei möglichst groß gewählt werden, weil sonst womöglich einige Schalterstellungen nicht auftreten (dadurch wird das Problem einfacher). Noch ein Hinweis zur Tabelle: Man sollte die Kosten nicht mit 2^N, sondern mit $2^N - 1$ vergleichen, weil für die erste Schalterstellung keine Kosten fällig werden[34].

N	**k**	**Kosten**	2^N
2	1000	3	4
3	1000	7	8
4	1000	16	16
4	10000	17	16
4	1000	15 (zweiter Versuch)	16
5	1000	32	32
5	10000	33	32
6	1000	69	64
6	10000	69	64
7	1000	136	128
7	10000	141	128
8	1000	275	256
8	10000	278	256
9	1000	499	512
9	10000	573	512

Ganz offensichlich ist der Algorithmus nicht optimal. Im Fall $N = 9$ und $k = 1000$ wurden scheinbar nicht alle 2^N Schalterstellungen generiert (ist ja Zufall). Man könnte den Algorithmus natürlich entsprechend verändern, es war mir aber wichtig, möglichst repräsentative Bitstringlisten zu sortieren.

[34] Es handelt sich dabei um den ersten Bitstring der eingegeben wird. Für diesen Bitstring sind per Definition keine Kosten fällig, da kein vorheriger Vergleichswert vorhanden.

Inwiefern haben sich die Kosten verringert? Um diese Frage beantworten zu können, werden die Kosten einer nicht sortierten Bitstring-Liste ausgewertet. Dazu werden möglichst große Werte N und möglichst kleine Werte k gewählt, damit mehrfach vorkommende gleiche Schalterstellungen unwahrscheinlicher werden. Diese erzeugen bei nicht sortierten Listen nämlich Kosten.

N	k	Kosten	Kosten unsortiert
100	25	732	1224
100	50	1722	2504
100	100	3900	4992
100	200	8402	9960

Es ist auf jeden Fall eine Verringerung der Kosten sichtbar. Mit steigendem k wird es aber offensichtlich schwieriger, die Bitstrings effizient anzuordnen, weil es so viele mögliche Schalterstellungen gibt, von denen viele nicht generiert wurden (es wurden immer nur 100 Bitstrings generiert). Es sind deshalb (prozentual bezogen auf die unsortierten Kosten) keine so großen Einsparungen mehr möglich. Bei größerem N sind sicherlich ähnliche Einsparungen möglich.

Einige Beispielausgaben .

```
anzahlDerSchalter=4, echteSchalter=4, durchlaeufe=100.

{1;0;1;1;}
{1;0;0;1;}
{0;0;0;0;}
{0;1;0;1;}
{1;1;0;1;}
{0;1;1;0;}
{1;0;0;0;}
{0;0;0;1;}
{0;0;1;1;}
{1;1;0;0;}
{1;0;1;0;}
{0;1;1;1;}
{1;1;1;1;}
{0;1;0;0;}
{0;0;1;0;}
{1;1;1;0;}

Sortierte Liste:
{0;1;0;0;}
{1;1;0;0;}
{1;0;0;0;}
{0;0;0;0;}
{0;0;0;1;}
{0;1;0;1;}
{1;1;0;1;}
{1;0;0;1;}
{1;0;1;1;}
{0;0;1;1;}
{0;1;1;1;}
{0;1;1;0;}
{0;0;1;0;}
{1;0;1;0;}
{1;1;1;0;}
{1;1;1;1;}

Kosten der unsortierten Liste: 32.
Kosten der sortierten Liste: 15.
```

```
Durch das Sortieren wurden Kosten in Höhe von 17 eingespart.

anzahlDerSchalter=5, echteSchalter=2, durchlaeufe=1000.

{1;0;1;1;1;}
{0;1;2;1;1;}
{0;1;1;0;0;}
{0;0;0;0;1;}
{1;1;0;1;0;}
{1;0;2;0;0;}

Sortierte Liste:
{1;0;2;0;0;}
{0;0;0;0;1;}
{0;1;1;0;0;}
{0;1;2;1;1;}
{1;0;1;1;1;}
{1;1;0;1;0;}

Kosten der unsortierten Liste: 15.
Kosten der sortierten Liste: 13.
Durch das Sortieren wurden Kosten in Höhe von 2 eingespart.

anzahlDerSchalter=5, echteSchalter=3, durchlaeufe=1000.

{0;0;0;1;0;}
{0;0;0;0;1;}
{1;0;0;0;0;}
{1;1;1;1;1;}
{0;0;1;1;1;}
{1;1;0;1;0;}
{1;1;0;0;1;}
{1;0;1;0;1;}
{0;1;0;1;1;}
{0;1;1;0;0;}
{1;0;1;1;0;}

Sortierte Liste:
{0;1;1;0;0;}
{1;0;1;1;0;}
{1;0;1;0;1;}
{1;1;0;0;1;}
{1;1;0;1;0;}
{1;0;0;0;0;}
{0;0;0;0;1;}
{0;0;0;1;0;}
{0;0;1;1;1;}
{1;1;1;1;1;}
{0;1;0;1;1;}

Kosten der unsortierten Liste: 28.
Kosten der sortierten Liste: 21.
Durch das Sortieren wurden Kosten in Höhe von 7 eingespart.

anzahlDerSchalter=6, echteSchalter=6, durchlaeufe=100.

{1;0;1;0;0;1;}
{1;0;1;1;1;1;}
{0;1;0;1;0;0;}
[...] (Siehe Datei algo_1_keinsort-6-6-100.txt)

Sortierte Liste:
{0;1;0;1;1;0;}
{0;0;0;1;1;1;}
{1;1;0;1;0;1;}
[...]
```

```
13
14 Kosten der unsortierten Liste: 198.
15 Kosten der sortierten Liste: 73.
16 Durch das Sortieren wurden Kosten in Höhe von 125 eingespart.

 1 anzahlDerSchalter=25, echteSchalter=5, durchlaeufe=100.
 2
 3 {1;1;1;0;1;0;1;0;0;0;0;0;1;1;1;1;1;1;1;1;1;1;1;1;1;}
 4 {0;0;1;1;0;1;0;1;1;1;0;1;1;1;1;1;1;0;1;1;0;1;1;0;0;}
 5 {1;1;1;1;1;0;0;1;1;1;0;1;0;1;0;1;1;0;1;1;1;0;0;0;1;}
 6 {0;1;1;0;1;1;0;1;0;0;0;0;0;1;0;0;0;0;1;1;1;0;0;0;0;}
 7 [...] (siehe algo_1_keinsort-25-5-100.txt)
 8
 9 Sortierte Liste:
10 {2;2;2;2;2;2;2;2;2;0;1;1;2;2;2;2;1;2;2;0;2;2;2;2;2;}
11 {2;2;2;1;2;1;2;0;1;2;0;0;2;0;2;0;2;1;0;2;2;2;1;1;0;}
12 {0;2;2;0;2;0;2;2;0;2;0;1;2;2;1;2;2;2;1;1;2;2;2;2;2;}
13 {0;2;0;1;2;0;1;0;0;2;1;0;1;2;1;2;2;0;0;1;1;1;0;2;1;}
14 [...]
15
16 Kosten der unsortierten Liste: 2638.
17 Kosten der sortierten Liste: 1497.
18 Durch das Sortieren wurden Kosten in Höhe von 1141 eingespart.
```

Beachtung jeder Schalterstellung Es ist interessant, wie sich der Algorithmus verhält, wenn jede Schalterstellung genau einmal vorkommt. Dazu wird die Erweiterung *Muster bei der Schalterauswahl* verwendet. Wenn $N = M$ mit Muster 1^N (N mal die Eins) und Umläufe nicht erlaubt sind, werden alle 2^M Möglichkeiten generiert, die M Schalter zu setzen. Diese lassen sich natürlich nicht zusammenfassen (kombinieren).

N	**Kosten**	**Kosten (unsortiert)**	2^N
3	7	13	8
4	16	33	16
4 (2. Versuch)	15	29	16
5	33	76	32
6	67	201	64
7	141	436	128
8	280	1041	256
9	558	2315	512
10	1107	5122	1024
15	35028	245396	32768

Noch größere Werte von M habe ich nicht getestet, weil es bei $M = 15$ schon ziemlich lange dauert, die $2^{15} = 32768$ Schalterstellungen zu generieren und dann zu sortieren (Nutzungsgrenze). Wie man an den Versuchen mit $N = 4$ sehen kann, ist der Algorithmus nicht optimal, abhängig von der ursprünglichen Anordnung der Bitstrings[35] ergeben sich Kosten von 16 oder 15. Wenn man die Zeile mit $N = 15$ betrachtet, sieht man jedoch, dass der Algorithmus schon ziemlich gut arbeitet. Es entstehen Kosten in Höhe von 35028, wobei die minimalen Kosten bei 32768 liegen. Die Ausgaben des Algorithmus (mit sortierten und unsortierten Bitstringlisten) befinden sich auf der CD-ROM im Ordner *sortieren*.

[35] Die Liste der Schalterstellungen wird anfangs zufällig durchgemischt, siehe auch Lösungsidee zum zweiten Algorithmus.

Die Ausgabe des zweiten Versuchs bei $N = 4$ gebe ich trotzdem kurz an.

```
Bester Aufruf mit 16 Elementen.
Schlechtester Aufruf mit 16 Elementen.
Durschnittlich mit 16 Elementen.
Vergangene Zeit: 72,0041ms.
Durchschnittliche Zeit pro Durchlauf: 72,0041ms.

{0;1;1;0;}
{0;0;1;1;}
{1;1;1;0;}
{0;1;0;1;}
{1;1;0;1;}
{1;0;1;0;}
{1;1;0;0;}
{0;0;1;0;}
{0;0;0;0;}
{1;0;0;0;}
{0;1;0;0;}
{0;0;0;1;}
{1;0;0;1;}
{1;1;1;1;}
{1;0;1;1;}
{0;1;1;1;}

Sortierter Code:
{0;1;1;1;}
{1;1;1;1;}
{1;0;1;1;}
{1;0;0;1;}
{0;0;0;1;}
{0;0;1;1;}
{0;0;1;0;}
{1;0;1;0;}
{1;1;1;0;}
{0;1;1;0;}
{0;1;0;0;}
{0;1;0;1;}
{1;1;0;1;}
{1;1;0;0;}
{1;0;0;0;}
{0;0;0;0;}

Kosten der unsortierten Liste: 29.
Kosten der sortierten Liste: 15.
Durch das Sortieren wurden Kosten in Höhe von 14 eingespart.
Der Code ist gültig.
```

3.3 Einige universelle Öffnungscodes

Es folgen nun einige Programmausgaben, die auch auf der CD-ROM zu finden sind. Sehr lange Öffnungcodes wurden bewusst abgeschnitten, da wahrscheinlich von Seiten der Bewerter kein Interesse besteht, diese per Hand zu überprüfen (oder was auch immer...). Auf der CD-ROM befinden sich die vollständigen Codes. Zu allen Öffnungscodes gibt es eine unsortierte und eine sortierte Variante und alle Öffnungscodes wurden auf Gültigkeit überprüft[36].

[36] In die Ausgaben des Programms hat sich ein kleiner Fehler eingeschlichen. Anstatt des letzten Satzes *Der Bitstring ist gueltig* sollte es *Der Öffnungscode ist gültig* heißen.

N = 6 und M = 3, einfacher Algorithmus ohne Sortieren .

```
Bester Aufruf mit 12 Elementen.
Schlechtester Aufruf mit 20 Elementen.
Durschnittlich mit 14,748 Elementen.
Vergangene Zeit: 1689,0966ms.
Durchschnittliche Zeit pro Durchlauf: 0,16890966ms.
anzahlDerSchalter=6, echteSchalter=3, durchlaeufe=10000.

{0;0;0;1;1;1;}
{0;1;0;1;0;0;}
{1;1;0;0;0;1;}
{1;0;1;1;0;0;}
{1;0;0;0;1;0;}
{1;1;1;1;1;1;}
{0;1;1;0;1;0;}
{0;0;1;0;0;1;}
{0;1;0;0;1;1;}
{1;0;0;1;0;1;}
{0;0;1;1;1;0;}
{1;1;1;0;0;0;}

Sortierte Liste:
{0;0;1;0;0;1;}
{1;1;1;1;1;1;}
{0;1;0;1;0;0;}
{1;1;1;0;0;0;}
{0;1;1;0;1;0;}
{0;0;1;1;1;0;}
{1;0;1;1;0;0;}
{1;0;0;1;0;1;}
{1;1;0;0;0;1;}
{0;1;0;0;1;1;}
{0;0;0;1;1;1;}
{1;0;0;0;1;0;}

Kosten der unsortierten Liste: 38.
Kosten der sortierten Liste: 28.
Durch das Sortieren wurden Kosten in Höhe von 10 eingespart.
Der Bitstring ist gueltig.
```

N = 9 und M = 3, einfacher Algorithmus mit Sortieren .

```
Bester Aufruf mit 15 Elementen.
Schlechtester Aufruf mit 23 Elementen.
Durschnittlich mit 18,7214 Elementen.
Vergangene Zeit: 329279,8337ms.
Durchschnittliche Zeit pro Durchlauf: 32,92798337ms.
anzahlDerSchalter=9, echteSchalter=3, durchlaeufe=10000.

{0;1;1;1;0;1;0;1;0;}
{1;1;0;1;1;1;1;0;0;}
{0;0;0;0;1;0;1;0;1;}
{1;1;1;0;0;0;1;0;1;}
{1;0;1;1;1;1;0;0;1;}
{0;0;1;0;1;1;1;1;0;}
{0;0;0;0;0;1;0;0;0;}
{1;1;0;0;1;0;0;1;0;}
{0;1;0;1;1;1;1;1;1;}
{0;1;1;1;1;0;0;1;1;}
{0;0;0;1;0;0;0;0;1;}
{1;0;1;1;0;0;1;1;0;}
{0;1;1;0;1;0;0;0;0;}
{1;0;0;0;0;1;0;1;1;}
{0;1;0;0;0;1;1;2;1;}
```

```
23
24 Sortierte Liste:
25 {0;1;0;0;0;1;1;2;1;}
26 {1;0;1;1;1;1;0;0;1;}
27 {1;1;0;1;1;1;1;0;0;}
28 {0;1;0;1;1;1;1;1;1;}
29 {0;1;1;1;1;0;0;1;1;}
30 {0;1;1;1;0;1;0;1;0;}
31 {0;0;1;0;1;1;1;1;0;}
32 {0;0;0;0;1;0;1;0;1;}
33 {0;0;0;1;0;0;0;0;1;}
34 {0;0;0;0;0;1;0;0;0;}
35 {1;0;0;0;0;1;0;1;1;}
36 {1;1;0;0;1;0;0;1;0;}
37 {0;1;1;0;1;0;0;0;0;}
38 {1;1;1;0;0;0;1;0;1;}
39 {1;0;1;1;0;0;1;1;0;}
40
41 Kosten der unsortierten Liste: 67.
42 Kosten der sortierten Liste: 51.
43 Durch das Sortieren wurden Kosten in Höhe von 16 eingespart.
44 Der Bitstring ist gueltig.
```

N = 9 und M = 3, einfacher Algorithmus ohne Sortieren .

```
1  Bester Aufruf mit 13 Elementen.
2  Schlechtester Aufruf mit 24 Elementen.
3  Durschnittlich mit 18,833092 Elementen.
4  Vergangene Zeit: 756359,2613ms.
5  Durchschnittliche Zeit pro Durchlauf: 0,7563592613ms.
6  anzahlDerSchalter=9, echteSchalter=3, durchlaeufe=1000000.
7
8  {0;0;0;1;0;0;1;1;1;}
9  {0;0;0;0;1;1;0;0;1;}
10 {0;1;1;0;0;1;1;1;0;}
11 {1;1;0;1;0;1;1;0;0;}
12 {1;1;1;1;1;0;0;0;1;}
13 {1;0;1;0;1;0;1;1;0;}
14 {1;0;1;1;0;1;0;1;1;}
15 {0;1;0;1;1;0;0;1;0;}
16 {1;0;0;0;0;0;0;0;0;}
17 {0;0;1;1;1;1;1;0;0;}
18 {1;1;0;0;1;1;1;1;1;}
19 {0;1;1;0;0;0;1;0;1;}
20 {0;1;1;0;0;1;0;1;0;}
21
22 Sortierte Liste:
23 {1;1;0;0;1;1;1;1;1;}
24 {1;0;1;0;1;0;1;1;0;}
25 {0;0;1;1;1;1;1;0;0;}
26 {1;1;0;1;0;1;1;0;0;}
27 {1;0;0;0;0;0;0;0;0;}
28 {0;0;0;0;1;1;0;0;1;}
29 {0;1;0;1;1;0;0;1;0;}
30 {1;1;1;1;1;0;0;0;1;}
31 {1;0;1;1;0;1;0;1;1;}
32 {0;0;0;1;0;0;1;1;1;}
33 {0;1;1;0;0;0;1;0;1;}
34 {0;1;1;0;0;1;1;1;0;}
35 {0;1;1;0;0;1;0;1;0;}
36
37 Kosten der unsortierten Liste: 62.
38 Kosten der sortierten Liste: 45.
39 Durch das Sortieren wurden Kosten in Höhe von 17 eingespart.
40 Der Bitstring ist gueltig.
```

N = 18 und M = 4, einfacher Algorithmus ohne Sortieren .

```
Bester Aufruf mit 63 Elementen.
Schlechtester Aufruf mit 75 Elementen.
Durschnittlich mit 68,9742 Elementen.
Vergangene Zeit: 1020861,39ms.
Durchschnittliche Zeit pro Durchlauf: 102,086139ms.
anzahlDerSchalter=18, echteSchalter=4, durchlaeufe=10000.

{0;0;1;0;0;1;0;1;1;1;1;0;1;1;0;1;1;1;}
{1;0;1;0;0;0;0;0;0;0;0;0;1;0;0;0;1;}
{1;1;1;1;0;0;1;0;1;0;0;1;1;0;1;1;1;1;}
[...]

Sortierte Liste:
{0;0;1;0;1;0;0;0;0;1;1;0;0;0;0;1;1;0;}
{1;0;0;0;1;0;1;1;0;2;1;0;1;1;1;1;1;0;}
{1;0;1;1;1;0;2;1;0;1;1;1;1;0;1;1;0;1;}
{1;0;1;1;0;1;0;1;0;1;0;0;0;0;1;1;0;1;}
{0;1;0;1;1;1;0;0;0;1;0;0;0;1;0;0;0;1;}
{0;0;0;1;1;1;1;0;0;1;1;1;1;0;0;0;0;1;}
{0;1;0;0;0;1;1;1;1;0;0;1;1;0;1;0;0;0;1;}
{0;1;0;0;0;0;0;1;1;0;1;0;1;1;0;0;1;0;1;}
{1;1;0;0;0;1;0;1;0;1;0;1;1;1;0;1;0;0;}
{0;1;1;0;0;1;0;0;0;0;0;1;1;1;1;1;0;1;}
{0;0;0;0;1;0;0;0;0;0;0;1;0;1;1;1;1;1;}
{1;0;0;1;1;1;1;0;0;1;0;1;0;1;1;0;1;1;}
{1;0;1;0;1;1;1;0;0;0;0;0;0;0;1;0;0;0;}
{0;1;1;0;1;1;1;0;0;1;0;0;1;1;1;0;1;0;}
{0;1;1;1;0;1;1;0;1;1;0;1;1;1;1;0;0;0;}
{0;1;1;1;0;0;1;0;1;1;1;0;0;0;0;0;0;1;}
{0;0;1;1;0;1;1;0;1;0;0;0;0;0;0;0;0;0;}
{0;0;1;1;1;1;1;1;1;1;0;1;0;0;1;1;0;1;0;}
{0;0;0;1;1;1;1;1;1;0;0;0;1;1;1;1;0;0;}
{0;0;0;1;1;0;1;1;1;1;0;0;1;0;0;1;1;0;}
{0;1;0;1;1;0;1;1;0;1;1;0;0;0;1;1;1;1;}
{1;1;0;1;0;0;0;1;0;0;1;0;1;0;0;1;1;0;}
{1;1;1;1;1;1;1;0;0;0;0;1;0;1;0;1;1;1;}
{1;0;0;1;0;1;0;0;0;0;1;1;1;0;1;0;1;0;}
{0;1;0;1;1;0;0;0;0;0;1;1;1;1;0;0;1;0;}
{1;1;0;1;0;0;0;0;0;1;1;1;0;1;0;1;1;0;}
{1;0;0;0;0;0;0;1;0;0;1;1;1;0;0;1;1;0;}
{1;0;0;0;0;0;0;1;0;1;1;1;1;1;1;1;0;0;}
{1;0;0;0;0;0;0;1;0;1;0;0;0;1;1;0;1;0;}
{0;0;1;1;0;0;1;0;0;0;0;0;1;1;0;1;0;1;}
{1;0;1;0;0;0;0;0;0;0;0;0;0;1;0;0;0;1;}
{1;1;1;1;1;0;0;1;0;0;1;0;0;1;0;0;0;1;}
{1;1;0;1;1;0;1;1;1;1;1;0;1;1;0;0;0;0;}
{1;1;1;1;1;0;1;1;1;1;0;1;0;1;1;1;0;0;}
{1;1;1;0;1;0;1;1;1;0;1;1;0;0;1;1;0;1;}
{1;1;1;1;0;0;1;0;1;0;0;1;1;0;1;1;1;1;}
{0;0;1;1;0;0;0;0;0;1;1;1;1;1;1;1;1;1;}
{0;0;1;0;0;1;0;1;1;1;1;0;1;1;0;1;1;1;}
{0;1;1;0;1;1;0;0;1;1;1;0;1;1;1;1;0;0;}
{0;1;1;0;1;0;0;1;1;0;1;0;1;0;0;1;0;0;}
{0;1;1;1;1;1;0;1;1;0;0;1;1;0;0;0;0;1;}
{1;0;1;1;1;1;0;1;1;1;0;1;1;1;0;0;1;0;}
{1;0;1;0;0;1;1;1;0;1;1;1;1;0;0;0;1;0;}
{1;0;1;0;0;0;1;1;1;0;1;0;1;0;1;0;1;1;}
{0;0;0;0;1;0;0;1;1;1;0;0;1;0;1;0;0;1;}
{0;1;0;0;1;1;0;1;1;1;1;1;1;0;1;0;1;1;}
{0;1;0;0;1;1;1;1;0;1;0;1;1;1;0;2;1;1;0;}
{1;0;0;0;1;1;1;0;1;1;0;1;1;0;0;0;1;0;0;}
{1;0;0;1;1;1;1;0;1;0;1;0;0;1;0;1;1;1;}
{1;1;1;0;1;1;1;0;1;1;0;1;0;1;0;1;1;1;}
{1;1;0;0;0;1;0;0;1;1;0;0;0;0;0;0;1;0;}
{0;1;0;1;0;1;0;0;1;0;0;0;0;0;1;1;0;0;}
```

```
{0;1;1;0;0;0;0;1;1;0;0;0;0;1;1;1;1;0;}
{0;1;1;0;0;0;0;1;0;1;0;1;0;0;1;0;0;0;}
{0;0;1;1;0;0;1;1;0;0;0;1;0;0;0;0;1;0;}
{0;0;0;1;0;0;1;1;1;1;0;1;0;1;0;2;0;1;}
{0;0;0;0;0;0;1;0;2;1;0;0;0;1;1;0;1;1;}
{1;1;0;0;0;1;0;1;0;0;0;0;1;1;1;0;1;1;}
{1;1;0;0;1;0;1;0;0;0;0;1;1;0;0;0;1;1;}
{1;1;0;1;0;0;0;0;1;0;1;1;0;1;1;0;0;1;}
{1;1;2;2;1;1;1;0;1;0;2;0;1;1;1;0;2;1;}
{2;2;2;1;1;0;0;0;1;2;0;2;2;2;2;2;0;2;}
{1;2;2;1;0;1;1;1;2;1;1;1;0;2;2;1;1;2;}

Kosten der unsortierten Liste: 639.
Kosten der sortierten Liste: 384.
Durch das Sortieren wurden Kosten in Höhe von 255 eingespart.
Der Bitstring ist gueltig.
```

N = 25 und M = 5, einfacher Algorithmus ohne Sortieren .

```
Bester Aufruf mit 204 Elementen.
Schlechtester Aufruf mit 213 Elementen.
Durschnittlich mit 207,52 Elementen.
Vergangene Zeit: 719540,1554ms.
Durchschnittliche Zeit pro Durchlauf: 7195,401554ms.
anzahlDerSchalter=25, echteSchalter=5, durchlaeufe=100.

{1;1;1;0;1;0;1;0;0;0;0;0;1;1;1;1;1;1;1;1;1;1;1;1;1;}
{0;0;1;1;0;1;0;1;1;1;0;1;1;1;1;1;1;0;1;1;0;1;1;0;0;}
{1;1;1;1;1;0;0;1;1;1;0;1;0;1;0;1;1;0;1;1;1;0;0;0;1;}
[...]

Sortierte Liste:
{2;2;2;2;2;2;2;2;2;0;1;1;2;2;2;2;1;2;2;0;2;2;2;2;2;}
{2;2;2;1;2;1;2;0;1;2;0;0;2;0;2;0;2;1;0;2;2;2;1;1;0;}
{0;2;2;0;2;0;2;2;0;2;0;1;2;2;1;2;2;2;1;1;2;2;2;2;2;}
[...]

Kosten der unsortierten Liste: 2638.
Kosten der sortierten Liste: 1497.
Durch das Sortieren wurden Kosten in Höhe von 1141 eingespart.
Der Bitstring ist gueltig.
```

N = 25 und M = 5, 1000 Durchläufe, einfacher Algorithmus ohne Sortieren .

```
Bester Aufruf mit 202 Elementen.
Schlechtester Aufruf mit 214 Elementen.
Durschnittlich mit 208,047 Elementen.
Vergangene Zeit: 6365494,0854ms.
Durchschnittliche Zeit pro Durchlauf: 6365,4940854ms.
anzahlDerSchalter=25, echteSchalter=5, durchlaeufe=1000.

{1;0;0;0;1;1;1;0;1;1;0;0;1;1;1;1;1;0;1;0;0;1;0;0;1;}
{0;0;1;0;0;0;1;1;0;0;1;0;1;1;1;1;0;0;1;0;0;0;0;0;1;}
{0;1;0;1;0;1;0;1;0;1;0;1;0;1;0;1;0;0;0;0;0;1;0;0;1;}
[...]

Sortierte Liste:
{0;1;1;0;1;0;1;0;1;0;2;0;2;1;2;2;1;1;0;0;0;2;0;1;1;}
{1;0;0;1;2;0;0;1;1;1;1;0;2;0;1;0;1;1;0;0;0;1;2;0;1;}
{1;1;1;1;0;0;1;1;0;1;0;0;0;0;0;1;1;1;0;0;0;0;0;0;1;}
[...]

Kosten der unsortierten Liste: 2655.
```

```
20 Kosten der sortierten Liste: 1509.
21 Durch das Sortieren wurden Kosten in Höhe von 1146 eingespart.
22 Der Bitstring ist gueltig.
```

4 Programm-Quelltext

4.1 Program.cs

Diese Datei enthält die *Main*-Funktion des Programms. Sie enthält das Menü, das angezeigt wird, wenn das Programm gestartet wird.

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\28_2_bwinf_1\Program.cs 1

```
using System;
using _28_2_bwinf_1_4;
using _28_2_bwinf_1_game;

namespace _28_2_bwinf_1
{
    class Program
    {
        public static void Main()
        {
            Console.WriteLine("28. BWINF, Aufgabe 1, Matthias Springer");
            Console.WriteLine("\nWelchen Programmteil wollen Sie starten?");
            Console.WriteLine("[1] Einfacher Algorithmus");
            Console.WriteLine("[2] Zweiter Algorithmus");
            Console.WriteLine("[3] Erweiterung: Das Spiel");
            Console.WriteLine("[4] Erweiterung: Bitstrings sortieren - Testfall generieren"↙
    );

            int eingabe = Convert.ToInt32(Console.ReadLine());

            if (eingabe == 1)
            {
                EinfacherAlgorithmus.Start();
            }
            else if (eingabe == 2)
            {
                ZweiterAlgorithmus.Start();
            }
            else if (eingabe == 3)
            {
                Spiel.Start();
            }
            else if (eingabe == 4)
            {
                BitstringsSortieren.StartDemo();
            }
        }
    }
}
```

4.2 BitstringsSortieren.cs

Diese Datei enthält die Erweiterung *Bitstrings sortieren*.

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\BitstringsSortieren.cs 1

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace _28_2_bwinf_1
{
    class BitstringsSortieren
    {
        /// <summary>
        /// Ermittelt eine gute Bitstringanordnung.
        /// </summary>
        public static List<int[]> Start(List<int[]> bitstrings)
        {
            List<int[]> zuBearbeitendeBitstrings = new List<int[]>();
            // Wertekopie der Liste anlegen
            foreach (int[] element in bitstrings) zuBearbeitendeBitstrings.Add(element);

            List<int[]> sortierteListe = new List<int[]>();
            sortierteListe.Add(zuBearbeitendeBitstrings[0]);
            zuBearbeitendeBitstrings.RemoveAt(0);

            while (zuBearbeitendeBitstrings.Count != 0)
            {
                int besterIndex = -1;
                int besteDifferenz = int.MaxValue;
                bool unten = true;

                // Alle Bitstrings durchgehen und Differenz pruefen
                for (int indexBitstring = 0; indexBitstring < zuBearbeitendeBitstrings.
    Count; indexBitstring++)
                {
                    // Mit dem ersten Element vergleichen
                    int dieseDifferenz = BitstringDifferenz(sortierteListe[0],
    zuBearbeitendeBitstrings[indexBitstring]);

                    if (dieseDifferenz < besteDifferenz)
                    {
                        besteDifferenz = dieseDifferenz;
                        besterIndex = indexBitstring;
                        unten = false;
                    }

                    // Mit dem letzten Element vergleichen
                    dieseDifferenz = BitstringDifferenz(sortierteListe[sortierteListe.Count
     - 1],
                                                        zuBearbeitendeBitstrings
    [indexBitstring]);

                    if (dieseDifferenz < besteDifferenz)
                    {
                        besteDifferenz = dieseDifferenz;
                        besterIndex = indexBitstring;
                        unten = true;
                    }

                    if (besteDifferenz == 0) break; // Besser geht es nicht
                }

                if (unten)
                {
                    sortierteListe.Add(zuBearbeitendeBitstrings[besterIndex]);
                }
                else
                {
                    sortierteListe.Insert(0, zuBearbeitendeBitstrings[besterIndex]);
                }

                zuBearbeitendeBitstrings.RemoveAt(besterIndex);
            }
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\BitstringsSortieren.cs 2

```
            Console.WriteLine("Sortierte Liste:");
            int gesamtKosten = 0;

            for (int i = 0; i < sortierteListe.Count; i++)
            {
                if (i > 0)
                {
                    gesamtKosten += BitstringDifferenz(sortierteListe[i - 1], sortierteListe[i]);
                }

                Console.WriteLine(intArrayAusgeben(sortierteListe[i]));
            }

            Console.WriteLine("Die Gesamtkosten betragen " + gesamtKosten.ToString() + ".");

            return sortierteListe;
        }

        /// <summary>
        /// Ermittelt zu zwei Bitstrings die Anzahl der unterschiedlichen Schalterstellungen.
        /// </summary>
        public static int BitstringDifferenz(int[] string1, int[] string2)
        {
            int rueckgabe = 0;

            for (int i = 0; i < string1.Count(); i++)
            {
                if (string2[i] != string1[i] && string1[i] != 2) rueckgabe++;
            }

            return rueckgabe;
        }

        /// <summary>
        /// Berechnet die Kostendifferenz zwischen zwei Bitstrings-Listen.
        /// </summary>
        public static int BerechneListenDifferenz(List<int[]> bitstrings1, List<int[]> bitstrings2)
        {
            int kosten1 = 0, kosten2 = 0;

            for (int i = 1; i < bitstrings1.Count; i++)
            {
                kosten1 += BitstringDifferenz(bitstrings1[i - 1], bitstrings1[i]);
            }

            for (int i = 1; i < bitstrings2.Count; i++)
            {
                kosten2 += BitstringDifferenz(bitstrings2[i - 1], bitstrings2[i]);
            }

            return kosten1 - kosten2;
        }

        /// <summary>
        /// Berchnet die Kosten einer Liste.
        /// </summary>
        public static int KostenListe(List<int[]> liste)
        {
            int rueckgabe = 0;

            for (int i = 1; i < liste.Count; i++)
            {
                rueckgabe += BitstringDifferenz(liste[i - 1], liste[i]);
            }

            return rueckgabe;
```

```
        }

        /// <summary>
        /// Startet eine Demo des Algorithmus (siehe Doku).
        /// </summary>
        public static void StartDemo()
        {
            List<int[]> demoListe = new List<int[]>();
            int anzahlDerBitstrings;
            int bitstringLaenge;

            Console.WriteLine("Anzahl der zu sortierenden Bitstrings:");
            anzahlDerBitstrings = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Länge eines Bitstrings:");
            bitstringLaenge = Convert.ToInt32(Console.ReadLine());

            Random zufall = new Random();

            for (int i = 0; i < anzahlDerBitstrings; i++)
            {
                int[] bitstring = new int[bitstringLaenge];

                for (int j = 0; j < bitstringLaenge; j++)
                {
                    bitstring[j] = zufall.Next(0, 2);
                }

                demoListe.Add(bitstring);
            }

            Console.WriteLine("Soll sortiert werden [j/n]?");
            if (Console.ReadLine() == "j")
            {
                Start(demoListe);
            }
            else
            {
                int gesamtKosten = 0;

                for (int i = 0; i < demoListe.Count; i++)
                {
                    if (i > 0)
                    {
                        gesamtKosten += BitstringDifferenz(demoListe[i - 1], demoListe[i]);
                    }

                    Console.WriteLine(intArrayAusgeben(demoListe[i]));
                }

                Console.WriteLine("Die Gesamtkosten betragen " + gesamtKosten.ToString() + ↙
    ".");
            }

            Console.ReadLine();
        }

        /// <summary>
        /// Wandelt ein Integer-Array in einen String um (fuer Debugzwecke und Ausgabe).
        /// </summary>
        public static string intArrayAusgeben(int[] wert)
        {
            string test = "{";

            for (int i = 0; i < wert.Length; i++)
            {
                test += wert[i].ToString() + ";";
            }

            test += "}";
            return test;
```

```
        }
    }
}
```

4.3 EinfacherAlgorithmus.cs

Diese Datei enthält den einfachen Algorithmus (mit und ohne Sortieren).

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\EinfacherAlgorithmus.cs 1

```
using System;
using System.Collections.Generic;
using System.IO;
using Facet.Combinatorics;

namespace _28_2_bwinf_1
{
    class EinfacherAlgorithmus
    {
        /// <summary>
        /// Enthaelt nur die M Schalterstellungen
        /// </summary>
        public static int[] vergleichsSchalterStellungen;

        /// <summary>
        /// Speichert nur die Indizes um die es gerade geht
        /// </summary>
        public static IList<int> vergleichsSchalterIndizes;


        static int anzahlDerSchalter = 6;
        static int echteSchalter = 3;
        static int durchlaeufe = 10;
        static bool sortieren = true;

        private static List<int[]> echteSchalterStellungen;
        private static int[] schalterIndexArray;
        private static List<IList<int>> variationen = new List<IList<int>>();

        private static Random rand = new Random();
        private static Random zufall = new Random();

        /// <summary>
        /// Mischt eine Liste zufaellig. (Wobei der Standard-Zufallsgenerator von .NET
    nicht besonders gut sein soll, hier koennte man u.U. noch was verbessern). Diese
    Funktion habe ich nicht selber geschrieben (was aber ohne weiteres moeglich ware...).
        /// </summary>
        public static void Shuffle<T>(IList<T> ilist)
        {
            int iIndex;
            T tTmp;
            for (int i = 1; i < ilist.Count; ++i)
            {
                iIndex = rand.Next(i + 1);
                tTmp = ilist[i];
                ilist[i] = ilist[iIndex];
                ilist[iIndex] = tTmp;
            }
        }

        /// <summary>
        /// Der Algorithmus: Ein Durchlauf.
        /// </summary>
        /// <returns>Liste mit Bitstrings</returns>
        public static List<EinzelnerCode> DerAlgo()
        {
            // Liste mit Bitstrings
            List<EinzelnerCode> ListeDerCodes = new List<EinzelnerCode>();

            // Console.WriteLine(variationen.Count.ToString() + " Auswahlmoeglichkeiten der
     echten Schalter gefunden.");

            // Durchmischen
            Shuffle<IList<int>>(variationen);
            Shuffle<int[]> (echteSchalterStellungen);

            foreach (IList<int> variation in variationen)
            {
                Shuffle(echteSchalterStellungen);
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\EinfacherAlgorithmus.cs 2

```
            foreach (int[] stellung in echteSchalterStellungen)
            {
                // Fuer jeden "Fall", also jede M aus N Schalter und deren Schalterstellungen, also alle (M aus N)*2^M Faelle
                vergleichsSchalterIndizes = variation;
                vergleichsSchalterStellungen = stellung;

                // Sortiere die Liste mit den schon vorhandenen Bitstrings fuer den aktuellen Fall, es wird auf die eben gesetzten Variablen zurueckgegriffen
                if (sortieren) ListeDerCodes.Sort((s1, s2) => s1.CompareTo(s2));
                // Shuffle(ListeDerCodes);

                bool schonKombiniert = false;

                for (int i = 0; i < ListeDerCodes.Count; i++)
                {
                    // Suche einen "Partner" zum kombinieren
                    bool kombinierbar = true;

                    // Versuche zu kombinieren: Ist das mit diesem Bitstring moeglich?
                    for (int j = 0; j < vergleichsSchalterIndizes.Count; j++)
                    {
                        if (vergleichsSchalterStellungen[j] != ListeDerCodes[i].Schalter[vergleichsSchalterIndizes[j]] && ListeDerCodes[i].Schalter[vergleichsSchalterIndizes[j]] != 2)
                        {
                            kombinierbar = false;
                            break;
                        }
                    }

                    if (kombinierbar)
                    {
                        // Mit dem aktuellen Bitstring kann kombiniert werden, also aendere den Bitstring entsprechend
                        for (int j = 0; j < vergleichsSchalterIndizes.Count; j++)
                        {
                            ListeDerCodes[i].Schalter[vergleichsSchalterIndizes[j]] =
                                vergleichsSchalterStellungen[j];
                        }

                        schonKombiniert = true;
                        break;
                    }
                }

                if (!schonKombiniert)
                {
                    // Es wurde kein passender Bitstring zum Kombinieren gefunden, Bitstring einfach einfuegen
                    EinzelnerCode neuerCode = new EinzelnerCode(anzahlDerSchalter);
                    for (int j = 0; j < vergleichsSchalterIndizes.Count; j++)
                    {
                        neuerCode.Schalter[vergleichsSchalterIndizes[j]] = vergleichsSchalterStellungen[j];
                    }

                    ListeDerCodes.Add(neuerCode);
                }
            }

        }

        // Console.WriteLine(ListeDerCodes.Count.ToString() + " viele Codes benoetigt.");

        return ListeDerCodes;
    }

    /// <summary>
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\EinfacherAlgorithmus.cs 3

```
        /// Hier startet das Programm.
        /// </summary>
        public static void Start()
        {
            // Eingabedaten von der Konsole einlesen
            Console.WriteLine("Anzahl der Schalter:");
            anzahlDerSchalter = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Echte Schalter:");
            echteSchalter = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Anzahl der Durchlaeufe:");
            durchlaeufe = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Soll sortiert werden [j/n]?");
            if (Console.ReadLine() == "j")
            {
                sortieren = true;
            }
            else
            {
                sortieren = false;
            }

            // Timer starten
            DateTime uhrzeitStart = DateTime.Now;

            // Erstelle Liste mit den Indizes aller Schalter, um dann M auszuwaehlen
            schalterIndexArray = new int[anzahlDerSchalter];
            for (int i = 0; i < anzahlDerSchalter; i++)
            {
                schalterIndexArray[i] = i;
            }

            // Generiere alle 2^M Moeglichkeiten, M Schalter zu setzen
            echteSchalterStellungen = new List<int[]>();
            int[] tmp1 = new int[echteSchalter];
            echteSchalterStellungen.Add(tmp1);
            tmp1 = new int[echteSchalter];
            tmp1[0] = 1;
            echteSchalterStellungen.Add(tmp1);


            for (int i = 1; i < echteSchalter; i++)
            {
                List<int[]> neueSchalterStellugen = new List<int[]>();
                foreach (int[] stellung in echteSchalterStellungen)
                {
                    int[] tmp2 = intArrayKopieren(stellung);
                    tmp2[i] = 0;
                    neueSchalterStellugen.Add(tmp2);
                    tmp2 = intArrayKopieren(stellung);
                    tmp2[i] = 1;
                    neueSchalterStellugen.Add(tmp2);
                }

                echteSchalterStellungen = neueSchalterStellugen;
            }

            // Nur zum Test: Alle 2^M Moeglichkeiten ausgeben
            for (int i = 0; i < echteSchalterStellungen.Count; i++)
            {
                Console.WriteLine(intArrayAusgeben(echteSchalterStellungen[i]));
            }

            // Generiere M aus N: echteSchalter aus anzahlDerSchalter
            Combinations<int> variationen2 = new Combinations<int>(schalterIndexArray,
echteSchalter);
            foreach (IList<int> vara in variationen2)
            {
                variationen.Add(vara);
            }
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\EinfacherAlgorithmus.cs 4

```
            // Speichere den besten, durschnittlichen und schlechtesten Wert
            int schlechteste = int.MinValue;
            int beste = int.MaxValue;
            double durchschnitt = 0;

            // Speichere doe beste Loesung (Bitstrings)
            List<EinzelnerCode> besteListe = new List<EinzelnerCode>();

            // Den Algorithmus mehrmals nacheinander ausfuehren
            for (int i = 0; i < durchlaeufe; i++)
            {
                List<EinzelnerCode> code = DerAlgo();

                int neu = code.Count;

                if (neu < beste)
                {
                    beste = neu;
                    besteListe = code;
                }

                if (neu > schlechteste) schlechteste = neu;
                durchschnitt += neu;
            }

            // Durchschnittswert berechnen
            durchschnitt /= durchlaeufe;

            // Timer stoppen
            DateTime uhrzeitEnde = DateTime.Now;
            TimeSpan dauer = uhrzeitEnde - uhrzeitStart;

            // Loesung in Datei schreiben
            StreamWriter datei;
            if (sortieren)
            {
                datei = new StreamWriter("algo_1_sort-" + anzahlDerSchalter.ToString() + "-
" + echteSchalter.ToString() + "-" + durchlaeufe.ToString() + ".txt", true);
            }
            else
            {
                datei = new StreamWriter("algo_1_keinsort-" + anzahlDerSchalter.ToString()
+ "-" + echteSchalter.ToString() + "-" + durchlaeufe.ToString() + ".txt", true);
            }

            datei.WriteLine("Bester Aufruf mit " + beste.ToString() + " Elementen.");
            datei.WriteLine("Schlechtester Aufruf mit " + schlechteste.ToString() + "
Elementen.");
            datei.WriteLine("Durschnittlich mit " + durchschnitt.ToString() + " Elementen.
");
            datei.WriteLine("Vergangene Zeit: " + dauer.TotalMilliseconds.ToString() + "ms.
");
            datei.WriteLine("Durchschnittliche Zeit pro Durchlauf: " + dauer.
TotalMilliseconds / durchlaeufe + "ms.");
            datei.WriteLine("anzahlDerSchalter=" + anzahlDerSchalter.ToString() + ",
echteSchalter=" + echteSchalter.ToString() + ", durchlaeufe=" + durchlaeufe.ToString()
+ ".");
            datei.WriteLine("");

            for (int i = 0; i < besteListe.Count; i++ )
            {
                datei.WriteLine(intArrayAusgeben(besteListe[i].Schalter));
            }

            // Int-Array-Liste erstellen
            List<int[]> schalterStellungenListe = new List<int[]>();
            foreach (EinzelnerCode code in besteListe) schalterStellungenListe.Add(code.
Schalter);

            // Sortierte Liste ausgeben
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\EinfacherAlgorithmus.cs 5

```
            List<int[]> sortiert = BitstringsSortieren.Start(schalterStellungenListe);
            datei.WriteLine("Sortierte Liste:");
            Console.WriteLine("Sortierte Liste:");
            for (int i = 0; i < sortiert.Count;i++ )
            {
                Console.WriteLine(intArrayAusgeben(sortiert[i]));
                datei.WriteLine(intArrayAusgeben(sortiert[i]));
            }
            datei.WriteLine("Kosten der unsortierten Liste: " + BitstringsSortieren.
    KostenListe(schalterStellungenListe).ToString() + ".");
            datei.WriteLine("Kosten der sortierten Liste: " + BitstringsSortieren.
    KostenListe(sortiert).ToString() + ".");
            Console.WriteLine("Kosten der unsortierten Liste: " + BitstringsSortieren.
    KostenListe(schalterStellungenListe).ToString() + ".");
            Console.WriteLine("Kosten der sortierten Liste: " + BitstringsSortieren.
    KostenListe(sortiert).ToString() + ".");
            Console.WriteLine("Durch das Sortieren wurden Kosten in Höhe von " +
    BitstringsSortieren.BerechneListenDifferenz(schalterStellungenListe, sortiert).ToString
    () + " eingespart.");
            datei.WriteLine("Durch das Sortieren wurden Kosten in Höhe von " +
    BitstringsSortieren.BerechneListenDifferenz(schalterStellungenListe, sortiert).ToString
    () + " eingespart.");

            // Loesung (bis auf Bitstrings) auf der Console ausgeben
            Console.WriteLine("Bester Aufruf mit " + beste.ToString() + " Elementen.");
            Console.WriteLine("Schlechtester Aufruf mit " + schlechteste.ToString() + "
    Elementen.");
            Console.WriteLine("Durschnittlich mit " + durchschnitt.ToString() + " Elementen
    .");
            Console.WriteLine("Vergangene Zeit: " + dauer.TotalMilliseconds.ToString() +
    "ms.");
            Console.WriteLine("Durchschnittliche Zeit pro Durchlauf: " + dauer.
    TotalMilliseconds/durchlaeufe + "ms.");

            if (LoesungPruefen(sortiert))
            {
                Console.WriteLine("Der Bitstring ist gueltig.");
                datei.WriteLine("Der Bitstring ist gueltig.");
            }
            else
            {
                Console.WriteLine("Der Bitstring ist UNGUELTIG.");
                datei.WriteLine("Der Bitstring ist UNGUELTIG.");
            }

            datei.Close();
            Console.ReadLine();
        }

        /// <summary>
        /// Prueft eine Liste mit Bitstrings auf Gueltigkeit.
        /// </summary>
        private static bool LoesungPruefen(List<int[]> bitstrings)
        {
            foreach (List<int> schalterAuswahl in variationen)
            {
                foreach (int[] schalterStellung in echteSchalterStellungen)
                {
                    bool bitstringGefunden = false;

                    foreach (int[] bitstring in bitstrings)
                    {
                        // Pruefe diesen Bitstring
                        bool bitstringPassend = true;

                        for (int index = 0; index < schalterAuswahl.Count; index++)
                        {
                            if (schalterStellung[index] != bitstring[schalterAuswahl
    [index]]) bitstringPassend = false;
                        }
```

```
                        if (bitstringPassend)
                        {
                            bitstringGefunden = true;
                            break;
                        }
                    }

                    if (!bitstringGefunden)
                    {
                        Console.WriteLine("Es fehlt eine Lösung!");
                        return false;
                    }
                }
            }

            return true;
        }

        /// <summary>
        /// Erstellt eine Wertekopie eines Integer-Arrays.
        /// </summary>
        public static int[] intArrayKopieren(int[] wert)
        {
            int[] result = new int[wert.Length];

            for (int i = 0; i < wert.Length; i++)
            {
                result[i] = wert[i];
            }

            return result;
        }

        /// <summary>
        /// Wandelt ein Integer-Array in einen String um (fuer Debugzwecke und Ausgabe).
        /// </summary>
        public static string intArrayAusgeben(int[] wert)
        {
            string test = "{";

            for (int i = 0; i < wert.Length; i++)
            {
                test += wert[i].ToString() + ";";
            }

            test += "}";
            return test;
        }
    }

    /// <summary>
    /// Ein Bitstring.
    /// </summary>
    public struct EinzelnerCode : IComparable
    {
        /// <summary>
        /// Stellungen der einzelnen Schalter.
        /// </summary>
        public int[] Schalter;


        /// <summary>
        /// Initialisierung den Bitstring mit ausreichend Schaltern.
        /// </summary>
        public EinzelnerCode(int anzahlDerSchalter)
        {
            Schalter = new int[anzahlDerSchalter];

            for (int i = 0; i < anzahlDerSchalter; i++)
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\EinfacherAlgorithmus.cs 7

```
            {
                Schalter[i] = 2;
            }
        }

        /// <summary>
        /// Gibt "wahr" zurueck, wenn dieser Bitstring mit dem Bitstring "code" kombiniert werden kann.
        /// </summary>
        public bool Kombinierbar(EinzelnerCode code)
        {
            for (int i = 0; i < code.Schalter.Length; i++)
            {
                if (code.Schalter[i] != Schalter[i] && code.Schalter[i] != 2 && Schalter[i] != 2)
                {
                    return false;
                }
            }

            return true;
        }

        /// <summary>
        /// Gibt die Anzahl der undefinierten Stellen (unbenutzte Schalter, Schalter=2) zurueck.
        /// </summary>
        public int AnzahlUndefinierterSchalter()
        {
            int rueckgabe = 0;

            for (int i = 0; i < Schalter.Length; i++)
            {
                if (Schalter[i] == 2) rueckgabe++;
            }

            return rueckgabe;
        }

        /// <summary>
        /// Gibt die Anzahl der Uebereinstimmungen mit dem Vergleichsbitstring (globale Variable) an.
        /// </summary>
        public int WertVergleichsArray()
        {
            int uebereinstimmung = 0;
            for (int i = 0; i < EinfacherAlgorithmus.vergleichsSchalterIndizes.Count; i++)
            {
                if (Schalter[EinfacherAlgorithmus.vergleichsSchalterIndizes[i]] == EinfacherAlgorithmus.vergleichsSchalterStellungen[i])
                {
                    uebereinstimmung++;
                }
                else if (Schalter[EinfacherAlgorithmus.vergleichsSchalterIndizes[i]] != 2)
                {
                    // Dieser Slot hat schon einen anderen, nicht kompaktiblen Wert!
                    uebereinstimmung = -30000; // TO-DO: Wert hinreichend klein machen!
                    return uebereinstimmung;
                }
            }

            return uebereinstimmung;
        }

        /// <summary>
        /// Legt eine Regel zum Sortieren der Liste der Bitstrings fest.
        /// </summary>
        public int CompareTo(object obj)
        {
            EinzelnerCode value = (EinzelnerCode)obj;
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\EinfacherAlgorithmus.cs 8

```
            int wert1 = AnzahlUndefinierterSchalter() - value.AnzahlUndefinierterSchalter()
    ; // Freie Stellen vergleichen, wenige freie Stellen ganz oben
            int wert2 = value.WertVergleichsArray() - WertVergleichsArray();        //
    Viele Uebereinstimmungen ganz oben

            return wert1 + wert2 * 5;   // Entsprechend gewichten, hier koennte man noch
    experimentieren
        }

        /// <summary>
        /// Erstellt eine Wertekopie dieser Struktur.
        /// </summary>
        /// <returns></returns>
        public EinzelnerCode KopiereStruktur()
        {
            EinzelnerCode neuerCode = new EinzelnerCode(this.Schalter.Length);

            for (int i = 0; i < Schalter.Length; i++)
            {
                neuerCode.Schalter[i] = Schalter[i];
            }

            return neuerCode;
        }
    }
}
```

4.4 Spiel.cs

Diese Datei enthält die Erweiterung *Wer kann besser raten*.

F:\Users\meister\Documents\Visual Studio 2008\Projects\28_2_bwinf_1\28_2_bwinf_1\Spiel.cs 1

```
using System;
using System.Linq;
using System.Collections.Generic;

namespace _28_2_bwinf_1_game
{
    class Spiel
    {
        public static int[] schalterLoesung;

        public static void Start()
        {
            int N;
            int M;
            int durchlaefe;
            int durchlaeufe3;

            // Eingabedaten von der Konsole einlesen
            Console.WriteLine("Anzahl der Schalter:");
            N = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Echte Schalter:");
            M = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Anzahl der Durchläufe:");
            durchlaefe = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Wie oft soll der dritte Algorithmus pro Durchlauf ausgeführt ↙
     werden (Parameter in 1er-Schritten zunehmend)?");
            durchlaeufe3 = Convert.ToInt32(Console.ReadLine());

            // Durchschnittswerte speichern
            double avg_algo1 = 0;
            double avg_algo2a = 0;
            double avg_algo2b = 0;
            double[] avg_algo3 = new double[N + 1];

            for (int durchlauf = 0; durchlauf < durchlaefe; durchlauf++)
            {
                Console.WriteLine("Generiere zufälligen Testfall.");
                Random zufall = new Random();

                schalterLoesung = new int[N];
                List<int> echteSchalterIndizes = new List<int>();

                while (echteSchalterIndizes.Count < M)
                {
                    int naechsterIndex = zufall.Next(0, N);

                    if (!echteSchalterIndizes.Contains(naechsterIndex))
                    {
                        // Dieser Schalter wurde noch nicht ausgewaehlt
                        echteSchalterIndizes.Add(naechsterIndex);
                    }
                }

                // Schalterstellungen zuweisen
                for (int i = 0; i < N; i++)
                {
                    schalterLoesung[i] = 2;
                }

                foreach (int ausgewaehlterSchalter in echteSchalterIndizes)
                {
                    schalterLoesung[ausgewaehlterSchalter] = zufall.Next(0, 2);
                }

                Console.WriteLine("Es wurde der Fall " + intArrayAusgeben(schalterLoesung) ↙
    + " erstellt.");

                int algo1 = einfacherAlgo();
                avg_algo1 += algo1;
                Console.WriteLine("Der einfache Algorithmus braucht " + algo1.ToString() + ↙
```

```
    " Versuche.");

            int algo2a = zweiterAlgo(false);
            avg_algo2a += algo2a;
            Console.WriteLine("Der zweite Algorithmus braucht " + algo2a.ToString() + "
 Versuche.");

            int algo2b = zweiterAlgo(true);
            avg_algo2b += algo2b;
            Console.WriteLine("Der verbesserte zweite Algorithmus braucht " + algo2b.
ToString() +
                              " Versuche.");

            for (int i = 1; i < durchlaeufe3 + 1; i++)
            {
                int algo3 = dritterAlgo(i);
                avg_algo3[i] += algo3;
                Console.WriteLine("Dritter Algorithmus (" + i.ToString() + ") braucht "
 + algo3.ToString() + " Versuche.");
            }
        }

        // Bilde Durchschnittswerte
        avg_algo1 /= Convert.ToDouble(durchlaefe);
        avg_algo2a /= Convert.ToDouble(durchlaefe);
        avg_algo2b /= Convert.ToDouble(durchlaefe);

        Console.WriteLine("\nDie folgenden Werte sind Durchschnittswerte!");
        Console.WriteLine("Der einfache Algorithmus braucht " + avg_algo1.ToString() +
" Versuche.");
        Console.WriteLine("Der zweite Algorithmus braucht " + avg_algo2a.ToString() + "
 Versuche.");
        Console.WriteLine("Der verbesserte zweite Algorithmus braucht " + avg_algo2b.
ToString() +
                          " Versuche.");
        for (int i = 1; i < durchlaeufe3 + 1; i++)
        {
            avg_algo3[i] /= Convert.ToDouble(durchlaefe);
            Console.WriteLine("Dritter Algorithmus (" + i.ToString() + ") braucht " +
avg_algo3[i].ToString() + " Versuche.");

        }

        Console.ReadLine();
    }

    /// <summary>
    /// Wandelt ein Integer-Array in einen String um (fuer Debugzwecke und Ausgabe).
    /// </summary>
    public static string intArrayAusgeben(int[] wert)
    {
        string test = "{";

        for (int i = 0; i < wert.Length; i++)
        {
            test += wert[i].ToString() + ";";
        }

        test += "}";
        return test;
    }

    /// <summary>
    /// Gibt die Anzahl der richtigen Schalterstellungen zurueck.
    /// </summary>
    /// <param name="schalterStellungen">Schalterstellungen</param>
    static int schalterStellungBewerten(int[] schalterStellungen)
    {
        int rueckgabe = 0;
```

```
            for (int i = 0; i < schalterLoesung.Count(); i++)
            {
                if (schalterStellungen[i] == schalterLoesung[i])
                {
                    rueckgabe++;
                }
            }

            return rueckgabe;
        }

        static int einfacherAlgo()
        {
            int[] aktuellerVersuch = new int[schalterLoesung.Count()];
            int schalterVeraenderungen = 0;
            int schalterTests = 0;
            int situationVorher = schalterStellungBewerten(aktuellerVersuch);
            schalterTests++;

            for (int i = 0; i < schalterLoesung.Count(); i++)
            {
                // Schalter umstellen
                if (aktuellerVersuch[i] == 1)
                {
                    aktuellerVersuch[i] = 0;
                }
                else
                {
                    aktuellerVersuch[i] = 1;
                }
                schalterVeraenderungen++;

                int situationNachher = schalterStellungBewerten(aktuellerVersuch);
                schalterTests++;

                if (situationVorher > situationNachher)
                {
                    // Die Loesung wurde dadurch nur verschlechtert
                    // Schalter umstellen
                    if (aktuellerVersuch[i] == 1)
                    {
                        aktuellerVersuch[i] = 0;
                    }
                    else
                    {
                        aktuellerVersuch[i] = 1;
                    }
                    schalterVeraenderungen++;
                }
                else
                {
                    situationVorher = situationNachher;
                }
            }

            if (!intArrayGleichheit(schalterLoesung, aktuellerVersuch))
            {
                Console.WriteLine("FEHLER IM ALGORITHMUS!!");
            }

            return schalterTests;
        }

        static int zweiterAlgo(bool verbessert)
        {
            int[] aktuellerVersuch = new int[schalterLoesung.Count()];
            for (int i = 0; i < schalterLoesung.Count(); i++) aktuellerVersuch[i] = 0;

            int schalterVeraenderungen = 0;
            int schalterTests = 0;
```

```
            int positionLinks = 0;
            int positionRechts = schalterLoesung.Count() - 1;
            int situationVorher = schalterStellungBewerten(aktuellerVersuch);
            schalterTests++;
            List<int> zuUntersuchendeIndizes = new List<int>();

            while (!(positionLinks > positionRechts))
            {
                // Zwei Schalter umdrehen
                if (aktuellerVersuch[positionLinks] == 0)
                {
                    aktuellerVersuch[positionLinks] = 1;
                }
                else
                {
                    aktuellerVersuch[positionLinks] = 0;
                }

                if (aktuellerVersuch[positionRechts] == 0)
                {
                    aktuellerVersuch[positionRechts] = 1;
                }
                else
                {
                    aktuellerVersuch[positionRechts] = 0;
                }

                schalterVeraenderungen += 2;
                int situationNachher = schalterStellungBewerten(aktuellerVersuch);
                schalterTests++;

                if (situationNachher - situationVorher == 2)
                {
                    // Situation hat sich um 2 Punkte verbessert perfekt.
                    situationVorher = situationNachher;
                }
                else if (situationVorher - situationNachher == 2)
                {
                    // Situation hat sich um 2 Punkte verschlechtert, also wieder beide ↙
    umdrehen.
                    // Zwei Schalter umdrehen
                    if (aktuellerVersuch[positionLinks] == 0)
                    {
                        aktuellerVersuch[positionLinks] = 1;
                    }
                    else
                    {
                        aktuellerVersuch[positionLinks] = 0;
                    }

                    if (aktuellerVersuch[positionRechts] == 0)
                    {
                        aktuellerVersuch[positionRechts] = 1;
                    }
                    else
                    {
                        aktuellerVersuch[positionRechts] = 0;
                    }

                    schalterVeraenderungen += 2;
                }
                else if (situationNachher == situationVorher)
                {
                    // Entweder beide Schalter werden nicht verwendet oder sie wurden so ↙
    umgestellt, dass sich die Wirkung wieder aufhebt.
                    // Zwei Schalter umdrehen

                    zuUntersuchendeIndizes.Add(positionLinks);
                    zuUntersuchendeIndizes.Add(positionRechts);
```

```
            }
            else if (situationNachher - situationVorher == 1)
            {
                // Situation hat sich um 1 verbessert, der andere Schalter ist Attrappe
.
                situationVorher = situationNachher;
            }
            else if (situationVorher - situationNachher == 1)
            {
                // Situation hat sich um 1 verschlechtert, es ist jedoch unbekannt
durch welchen der beiden Schalter.
                // Zwei Schalter umdrehen
                if (aktuellerVersuch[positionLinks] == 0)
                {
                    aktuellerVersuch[positionLinks] = 1;
                }
                else
                {
                    aktuellerVersuch[positionLinks] = 0;
                }

                if (aktuellerVersuch[positionRechts] == 0)
                {
                    aktuellerVersuch[positionRechts] = 1;
                }
                else
                {
                    aktuellerVersuch[positionRechts] = 0;
                }
            }

            positionLinks++;
            positionRechts--;
        }

        situationVorher = schalterStellungBewerten(aktuellerVersuch);

        while (zuUntersuchendeIndizes.Count != 0)
        {
            int index = zuUntersuchendeIndizes[0];
            zuUntersuchendeIndizes.RemoveAt(0);

            // Probiere alle Schalter aus
            // Schalter umstellen
            if (aktuellerVersuch[index] == 1)
            {
                aktuellerVersuch[index] = 0;
            }
            else
            {
                aktuellerVersuch[index] = 1;
            }
            schalterVeraenderungen++;

            int situationNachher = schalterStellungBewerten(aktuellerVersuch);
            schalterTests++;

            if (situationVorher > situationNachher)
            {
                // Die Loesung wurde dadurch nur verschlechtert
                // Schalter umstellen
                if (aktuellerVersuch[index] == 1)
                {
                    aktuellerVersuch[index] = 0;
                }
                else
                {
                    aktuellerVersuch[index] = 1;
                }
                schalterVeraenderungen++;
```

F:\Users\meister\Documents\Visual Studio 2008\Projects\28_2_bwinf_1\28_2_bwinf_1\Spiel.cs 6

```
                    if (verbessert)
                    {
                        int index2 = zuUntersuchendeIndizes[0];
                        zuUntersuchendeIndizes.RemoveAt(0);

                        // Schalter umstellen
                        if (aktuellerVersuch[index2] == 1)
                        {
                            aktuellerVersuch[index2] = 0;
                        }
                        else
                        {
                            aktuellerVersuch[index2] = 1;
                        }
                        schalterVeraenderungen++;
                        situationVorher++;
                    }
                }
                else
                {
                    situationVorher = situationNachher;

                    if (verbessert)
                    {
                        zuUntersuchendeIndizes.RemoveAt(0);
                    }
                }
            }

            if (!intArrayGleichheit(schalterLoesung, aktuellerVersuch))
            {
                Console.WriteLine("FEHLER IM ALGORITHMUS!!");
            }

            return schalterTests;
        }

        /// <summary>
        /// Gibt wahr zurueck, wenn es sich um eine gerade Zahl handelt.
        /// </summary>
        static bool istIntGerade(int zahl)
        {
            return zahl % 2 == 0;
        }

        static int dritterAlgo(int schalterProSchritt)
        {
            int[] aktuellerVersuch = new int[schalterLoesung.Count()];
            int schalterVeraenderungen = 0;
            int schalterTests = 0;

            int situationVorher = schalterStellungBewerten(aktuellerVersuch);
            schalterTests++;

            // Schalter, die nachtraeglich untersucht werden muessen (wie im zweiten Algorithmus)
            List<int> nachtraeglichZuUntersuchendeIndizes = new List<int>();

            // Schalter, die noch nicht umgedreht werden
            List<int> zuUntersuchendeSchalter = new List<int>();
            for (int i = 0; i < schalterLoesung.Count(); i++) zuUntersuchendeSchalter.Add(i);
            Shuffle(zuUntersuchendeSchalter);

            while (zuUntersuchendeSchalter.Count != 0)
            {
                // Schalter auswaehlen und umdrehen
                int anzahlSchalter = schalterProSchritt;
                if (anzahlSchalter > zuUntersuchendeSchalter.Count) anzahlSchalter =
```

F:\Users\meister\Documents\Visual Studio 2008\Projects\28_2_bwinf_1\28_2_bwinf_1\Spiel.cs 7

```
    zuUntersuchendeSchalter.Count;

                // Schalter auswaehlen und in einer Liste speichern
                List<int> ausgewaehlteSchalter = new List<int>();
                for (int i = 0; i < anzahlSchalter; i++)
                {
                    // Elemente immer am Anfang der Liste entnehmen
                    ausgewaehlteSchalter.Add(zuUntersuchendeSchalter[0]);
                    zuUntersuchendeSchalter.RemoveAt(0);
                }

                // Schalter umdrehen
                foreach (int schalterIndex in ausgewaehlteSchalter)
                {
                    if (aktuellerVersuch[schalterIndex] == 0)
                    {
                        aktuellerVersuch[schalterIndex] = 1;
                    }
                    else
                    {
                        aktuellerVersuch[schalterIndex] = 0;
                    }
                    schalterVeraenderungen++;
                }

                int situationNachher = schalterStellungBewerten(aktuellerVersuch);
                schalterTests++;

                if (situationNachher - situationVorher == anzahlSchalter)
                {
                    // Alles bestens, die Schalter wurden richtigerweise umgestellt
                    situationVorher = situationNachher;
                }
                else if (situationNachher - situationVorher == anzahlSchalter - 1)
                {
                    // Ein Schalter ist eine Attrappe, die restlichen wurden richtig
    gesetzt
                    situationVorher = situationNachher;
                }
                else if (situationVorher - situationNachher == anzahlSchalter)
                {
                    // Alle Aenderungen rueckgaengig machen, es wurden alle Schalter falsch
     gesetzt
                    // Schalter umdrehen
                    foreach (int schalterIndex in ausgewaehlteSchalter)
                    {
                        if (aktuellerVersuch[schalterIndex] == 0)
                        {
                            aktuellerVersuch[schalterIndex] = 1;
                        }
                        else
                        {
                            aktuellerVersuch[schalterIndex] = 0;
                        }
                        schalterVeraenderungen++;
                    }
                }
                else if (situationVorher - situationNachher == anzahlSchalter - 1)
                {
                    // Ein Schalter ist eine Attrappe, die restlichen wurden falsch gesetzt
                    // Schalter umdrehen
                    foreach (int schalterIndex in ausgewaehlteSchalter)
                    {
                        if (aktuellerVersuch[schalterIndex] == 0)
                        {
                            aktuellerVersuch[schalterIndex] = 1;
                        }
                        else
                        {
                            aktuellerVersuch[schalterIndex] = 0;
```

F:\Users\meister\Documents\Visual Studio 2008\Projects\28_2_bwinf_1\28_2_bwinf_1\Spiel.cs 8

```
                    }
                    schalterVeraenderungen++;
                }
            }
            else
            {
                // Die Schalter einzeln ueberpruefen
                //Console.WriteLine("ADD " + ausgewaehlteSchalter.Count);
                nachtraeglichZuUntersuchendeIndizes.AddRange(ausgewaehlteSchalter);
                situationVorher = situationNachher;
            }
        }

        foreach (int index in nachtraeglichZuUntersuchendeIndizes)
        {
            // Probiere alle Schalter aus
            // Schalter umstellen
            if (aktuellerVersuch[index] == 1)
            {
                aktuellerVersuch[index] = 0;
            }
            else
            {
                aktuellerVersuch[index] = 1;
            }
            schalterVeraenderungen++;

            int situationNachher = schalterStellungBewerten(aktuellerVersuch);
            schalterTests++;

            if (situationVorher > situationNachher)
            {
                // Die Loesung wurde dadurch nur verschlechtert
                // Schalter umstellen
                if (aktuellerVersuch[index] == 1)
                {
                    aktuellerVersuch[index] = 0;
                }
                else
                {
                    aktuellerVersuch[index] = 1;
                }
                schalterVeraenderungen++;
            }
            else
            {
                situationVorher = situationNachher;
            }
        }

        if (!intArrayGleichheit(schalterLoesung, aktuellerVersuch))
        {
            Console.WriteLine("FEHLER IM ALGORITHMUS!!");
            Console.ReadLine();
        }

        return schalterTests;
    }

    private static Random rand = new Random();

    /// <summary>
    /// Mischt eine Liste zufaellig. (Wobei der Standard-Zufallsgenerator von .NET ↙
nicht besonders gut sein soll, hier koennte man u.U. noch was verbessern).
    /// </summary>
    public static void Shuffle<T>(IList<T> ilist)
    {
        int iIndex;
        T tTmp;
        for (int i = 1; i < ilist.Count; ++i)
```

```
            {
                iIndex = rand.Next(i + 1);
                tTmp = ilist[i];
                ilist[i] = ilist[iIndex];
                ilist[iIndex] = tTmp;
            }
        }

        /// <summary>
        /// Prueft zwei gleich grosse Int-Arrays auf Gleichheit.
        /// </summary>
        static bool intArrayGleichheit(int[] array1, int[] array2)
        {
            bool falsch = false;

            for (int i = 0; i < array1.Length; i++)
            {
                if (array1[i] != array2[i] && array1[i] != 2 && array2[i] != 2)
                {
                    Console.WriteLine("Stelle " + i.ToString() + ".");
                    falsch = true;
                }
            }

            return !falsch;
        }
    }
}
```

4.5 ZweiterAlgorithmus.pdf

Diese Datei enthält den zweiten Algorithmus und die Erweiterungen *Mehrere Schalterzustände* und *Muster bei der Schalterauswahl*.

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\ZweiterAlgorithmus.cs 1

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using _28_2_bwinf_1;
using Facet.Combinatorics;

namespace _28_2_bwinf_1_4
{
    public class ZweiterAlgorithmus
    {
        private static int anzahlDerSchalter = 25;
        private static int echteSchalter = 5;
        private static List<int[]> listeDerCodes;
        private static List<SchalterAuswahl> listeSchalterAuswahl;
        private static int durchlaeufe;
        private static List<int> schalterIndizes;
        private static List<int[]> echteSchalterStellungen;
        private static int[] muster;
        private static bool mitUmlauf = false;
        private static bool mitMuster = false;
        private static int anzahlSchalterZustaende = 2;

        /// <summary>
        /// Generiert alle Schalterstellungen.
        /// </summary>
        static void GeneriereSchalterStellungen()
        {
            // Generiere alle 2^M Moeglichkeiten, M Schalter zu setzen
            echteSchalterStellungen = new List<int[]>();
            int[] tmp1 = new int[echteSchalter];
            echteSchalterStellungen.Add(tmp1);

            for (int i = 1; i < anzahlSchalterZustaende; i++)
            {
                int wert = i;
                if (wert == 2) wert = -2;

                tmp1 = new int[echteSchalter];
                tmp1[0] = wert;
                echteSchalterStellungen.Add(tmp1);
            }

            for (int i = 1; i < echteSchalter; i++)
            {
                List<int[]> neueSchalterStellugen = new List<int[]>();
                foreach (int[] stellung in echteSchalterStellungen)
                {
                    for (int j = 0; j < anzahlSchalterZustaende; j++)
                    {
                        int wert = j;
                        if (wert == 2) wert = -2;

                        int[] tmp2 = intArrayKopieren(stellung);
                        tmp2[i] = wert;
                        neueSchalterStellugen.Add(tmp2);
                    }
                }

                echteSchalterStellungen = neueSchalterStellugen;
            }
        }

        /// <summary>
        /// Generiert die Liste A mit allen Listen B (siehe Doku).
        /// </summary>
        static void GeneriereAlleZugangscodes()
        {
            listeDerCodes = new List<int[]>();
            listeSchalterAuswahl = new List<SchalterAuswahl>();
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\ZweiterAlgorithmus.cs 2

```
            GeneriereSchalterStellungen();

            // Liste der Schalterindizes
            schalterIndizes = new List<int>();
            for (int i = 0; i < anzahlDerSchalter; i++) schalterIndizes.Add(i);

            // Liste SchalterAuswahl initialisieren: Waehle M aus N Schalterindizes
            Combinations<int> combinations = new Combinations<int>(schalterIndizes,
    echteSchalter);

            foreach (IList<int> combination in combinations)
            {
                SchalterAuswahl aktuelleAuswahl = new SchalterAuswahl();
                aktuelleAuswahl.SchalterStellungen = new List<int[]>();

                for (int i = 0; i < echteSchalterStellungen.Count; i++)
                {
                    int[] schalterStellungen = bitstringErstellen(anzahlDerSchalter);
                    for (int schalterIndexIndex = 0; schalterIndexIndex < combination.Count
    ; schalterIndexIndex++)
                    {
                        schalterStellungen[combination[schalterIndexIndex]] =
                            echteSchalterStellungen[i][schalterIndexIndex];
                    }

                    aktuelleAuswahl.SchalterStellungen.Add(schalterStellungen);
                }

                Shuffle(aktuelleAuswahl.SchalterStellungen);
                aktuelleAuswahl.AusgewaehlteSchalter = combination.ToArray();

                listeSchalterAuswahl.Add(aktuelleAuswahl);
            }

            Shuffle(listeSchalterAuswahl);
            DebugMessage("Liste SchalterAuswahl wurde mit " + listeSchalterAuswahl.Count.
    ToString() + " Elementen erstellt.");
        }

        /// <summary>
        /// Generiert die Liste A mit allen Listen B (siehe Doku), wobei ein Muster
    beachtet wird (Erweiterung).
        /// </summary>
        static void GeneriereAlleZugangscodesMuster()
        {
            listeDerCodes = new List<int[]>();
            listeSchalterAuswahl = new List<SchalterAuswahl>();

            GeneriereSchalterStellungen();
            List<List<int>> listeSchalterIndizes = new List<List<int>>();
            for (int schalterIndex = 0; schalterIndex < anzahlDerSchalter; schalterIndex++)
            {
                // Schalterindex ist die Startposition
                // Ein Fall ist unguelig, wenn ein Umlauf erforderlich ist, Umlaeufe aber
    deaktiviert sind
                bool aktuellerFallGueltig = true;
                List<int> aktuelleSchalterAuswahl = new List<int>();

                // Iteriere ueber die ganze Musterlaenge
                for (int i = 0; i < muster.Count(); i++)
                {
                    if (i + schalterIndex >= anzahlDerSchalter && !mitUmlauf)
                    {
                        // Jetzt weare ein Umlauf notwendig, der aber nicht erlaubt ist
                        aktuellerFallGueltig = false;
                        break;
                    }

                    if (muster[i] == 1)
```

```
                    {
                        // Schalter ist gesetzt
                        aktuelleSchalterAuswahl.Add((i + schalterIndex)%anzahlDerSchalter);
                    }
                }

                if (aktuellerFallGueltig && aktuelleSchalterAuswahl.Count != 0)
                {
                    listeSchalterIndizes.Add(aktuelleSchalterAuswahl);
                }
            }

            // Generiere die Liste A
            foreach (IList<int> combination in listeSchalterIndizes)
            {
                SchalterAuswahl aktuelleAuswahl = new SchalterAuswahl();
                aktuelleAuswahl.SchalterStellungen = new List<int[]>();

                // Generiere die Listen B
                for (int i = 0; i < echteSchalterStellungen.Count; i++)
                {
                    int[] schalterStellungen = bitstringErstellen(anzahlDerSchalter);
                    for (int schalterIndexIndex = 0; schalterIndexIndex < combination.Count
; schalterIndexIndex++)
                    {
                        schalterStellungen[combination[schalterIndexIndex]] =
                            echteSchalterStellungen[i][schalterIndexIndex];
                    }

                    aktuelleAuswahl.SchalterStellungen.Add(schalterStellungen);
                }

                Shuffle(aktuelleAuswahl.SchalterStellungen);
                aktuelleAuswahl.AusgewaehlteSchalter = combination.ToArray();

                listeSchalterAuswahl.Add(aktuelleAuswahl);
            }

            Shuffle(listeSchalterAuswahl);
            DebugMessage("Liste SchalterAuswahl wurde mit " + listeSchalterAuswahl.Count.
ToString() + " Elementen erstellt.");
        }

        static void DebugMessage(string text)
        {
            StreamWriter datei = new StreamWriter("debug- " + anzahlDerSchalter.ToString()
+ "-" + echteSchalter.ToString() + "-" + durchlaeufe.ToString() + ".txt", true);
            datei.WriteLine(text);
            datei.Close();
        }

        public static void Start()
        {
            // Eingabedaten von der Konsole einlesen
            Console.WriteLine("Anzahl der Schalter:");
            anzahlDerSchalter = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Echte Schalter:");
            echteSchalter = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Anzahl der Durchläufe:");
            durchlaeufe = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Wollen Sie ein Muster fuer die Schalterauswahl eingeben
(Erweiterung) [j/n]?");

            if (Console.ReadLine() == "j")
            {
                mitMuster = true;
                Console.WriteLine("Geben Sie das Muster ein (0=Schalter nicht benutzt, 1=
Schalter in Verwendung):");
                string musterEingabe = Console.ReadLine();
                muster = new int[musterEingabe.Length];
```

```
            for (int i = 0; i < musterEingabe.Length; i++)
            {
                if (musterEingabe[i] == '1')
                {
                    muster[i] = 1;
                }
            }

            Console.WriteLine("Sollen Umläufe erlaubt sein?");

            if (Console.ReadLine() == "j")
            {
                Console.WriteLine("Umläufe sind erlaubt.");
                mitUmlauf = true;
            }
        }

        Console.WriteLine("Wie viele Schalterzustände gibt es?");
        anzahlSchalterZustaende = Convert.ToInt32(Console.ReadLine());

        // Timer starten
        DateTime uhrzeitStart = DateTime.Now;

        // Beste Werte speichern
        int besterWert = int.MaxValue;
        int schlechtesterWert = int.MinValue;
        double durschnittsWert = 0.0;
        List<int[]> besterCode = new List<int[]>();

        for (int durchlaufIndex = 0; durchlaufIndex < durchlaeufe; durchlaufIndex++)
        {
            if (mitMuster)
            {
                GeneriereAlleZugangscodesMuster();
            }
            else
            {
                GeneriereAlleZugangscodes();
            }

            // Hier gehts los, Algorithmus laeuft so lange, bis alle SchalterAuswahlen ↙
bearbeitet wurden
            while (listeSchalterAuswahl.Count != 0)
            {
                DebugMessage("Noch " + listeSchalterAuswahl.Count.ToString() + "       ↙
Auswahlen uebrig.");

                // Generiere eine Reihe an kompatiblen SchalterAuswahlen
                Shuffle(listeSchalterAuswahl);

                List<SchalterAuswahl> aktuelleReihe = new List<SchalterAuswahl>();
                int maximumReiheBitstrings = 0;

                // Ueberpruefe, welche miteinander kompatibel sind
                for (int i = 0; i < listeSchalterAuswahl.Count; i++)
                {
                    // Ueberpruefe, ob die aktuelle SchalterAuswahl kompatibel mit den ↙
bereits ausgewaehlten ist
                    bool istKompatibel = true;

                    for (int j = 0; j < aktuelleReihe.Count; j++)
                    {
                        if (!aktuelleReihe[j].IstKompatibel(listeSchalterAuswahl[i]))   ↙
istKompatibel = false;
                    }

                    if (istKompatibel)
                    {
                        // Die aktuelle SchalterAuswahl ist kompatibel, also auswaehlen
```

```
                            aktuelleReihe.Add(listeSchalterAuswahl[i]);
                            DebugMessage("-- Kompatible Schalterauswahl gefunden: " +
                                         intArrayAusgeben(listeSchalterAuswahl[i].
    AusgewaehlteSchalter));

                            // Sortieren
                            Shuffle(listeSchalterAuswahl[i].SchalterStellungen);
                            maximumReiheBitstrings = Math.Max(maximumReiheBitstrings,
 listeSchalterAuswahl[i].SchalterStellungen.Count);
                            //listeSchalterAuswahl.RemoveAt(i);
                            i--;
                        }
                    }

                    DebugMessage("Kompatible Auswahl gefunden.");

                    // Es liegt nun eine Liste mit komapatiblen SchalterAuswahlen vor
                    List<int[]> neueBitstrings = new List<int[]>();

                    for (int indexBitstring = 0; indexBitstring < maximumReiheBitstrings;
    indexBitstring++)
                    {
                        // Erstelle die einzelnen Bitstrings
                        int[] aktuellerBitstring = bitstringErstellen(anzahlDerSchalter);

                        // Uebertrage Schalterstellungen aus den SchalterAuswahlen
                        for (int indexSchalterAuswahl = 0;
                             indexSchalterAuswahl < aktuelleReihe.Count;
                             indexSchalterAuswahl++)
                        {
                            if (indexBitstring < aktuelleReihe[indexSchalterAuswahl].
    SchalterStellungen.Count)
                            {
                                // Nur weitermachen, wenn es in der SchalterAuswahl
    ueberhaupt so viele Stellungen gibt
                                // Uebernehme Schalter Stellungen
                                for (int schalterIndex = 0;
                                     schalterIndex <
                                     aktuelleReihe[indexSchalterAuswahl].SchalterStellungen
    [indexBitstring].Length;
                                     schalterIndex++)
                                {
                                    if (
                                        aktuelleReihe[indexSchalterAuswahl].
    SchalterStellungen[indexBitstring][
                                            schalterIndex] != 2)
                                        aktuellerBitstring[schalterIndex] =
                                            aktuelleReihe[indexSchalterAuswahl].
    SchalterStellungen[indexBitstring][
                                                schalterIndex];
                                }
                            }
                        }

                        neueBitstrings.Add(aktuellerBitstring);
                    }

                    DebugMessage("Basis Bitstrings erstellt.");

                    // Die neuen Bitstrings liegen nun vor: Versuche, ob kombiniert werden
    kann
                    Shuffle(neueBitstrings);
                    Shuffle(listeSchalterAuswahl);

                    DebugMessage("Bitstrings kombinieren, Entfernen der fertigen Faelle...
    ");
                    int mitgenerierteFaelle = 0;

                    // Entferne alle fertigen Faelle
                    for (int schalterAuswahlIndex = 0;
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\ZweiterAlgorithmus.cs 6

```
                        schalterAuswahlIndex < listeSchalterAuswahl.Count;
                        schalterAuswahlIndex++)
                {
                    for (int schalterStellungIndex = 0;
                         schalterStellungIndex < listeSchalterAuswahl
[schalterAuswahlIndex].SchalterStellungen.Count;
                         schalterStellungIndex++)
                    {
                        for (int bitstringIndex = 0; bitstringIndex < neueBitstrings.
Count; bitstringIndex++)
                        {
                            // Pruefe, ob der Bitstring die aktuelle Schalterstellung
schon beinhaltet
                            if (sindIntArrayKompatibel(neueBitstrings[bitstringIndex],
listeSchalterAuswahl[schalterAuswahlIndex].SchalterStellungen
[schalterStellungIndex]))
                            {
                                // Kombiniere die Bitstrings
                                for (int indexSchalter = 0;
                                     indexSchalter < neueBitstrings[bitstringIndex].
Length;
                                     indexSchalter++)
                                {
                                    if (
                                        listeSchalterAuswahl[schalterAuswahlIndex].
SchalterStellungen[
                                            schalterStellungIndex][indexSchalter] != 2)
                                    {
                                        neueBitstrings[bitstringIndex][indexSchalter] =
                                            listeSchalterAuswahl[schalterAuswahlIndex].
SchalterStellungen[
                                                schalterStellungIndex][indexSchalter];
                                    }
                                }

                                listeSchalterAuswahl[schalterAuswahlIndex].
SchalterStellungen.RemoveAt(
                                    schalterStellungIndex);
                                schalterStellungIndex--;
                                mitgenerierteFaelle++;
                                break;
                            }
                        }
                    }

                    if (listeSchalterAuswahl[schalterAuswahlIndex].SchalterStellungen.
Count == 0)
                    {
                        // Komplett enfernen
                        listeSchalterAuswahl.RemoveAt(schalterAuswahlIndex);
                        schalterAuswahlIndex--;
                    }
                }

                DebugMessage("Es wurden " + mitgenerierteFaelle.ToString() + " Faelle
mitgeneriert.");


                // Es wurden alle Faelle entfernt
                listeDerCodes.AddRange(neueBitstrings);
            }

            //Console.WriteLine("Es wurden " + listeDerCodes.Count.ToString() + "
Bitstrings benoetigt.");

            if (listeDerCodes.Count < besterWert)
            {
                besterCode = listeDerCodes;
                besterWert = listeDerCodes.Count;
            }
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\ZweiterAlgorithmus.cs 7

```
            if (listeDerCodes.Count > schlechtesterWert) schlechtesterWert = listeDerCodes.Count;

            durschnittsWert += listeDerCodes.Count;
        }

        // Timer stoppen
        DateTime uhrzeitEnde = DateTime.Now;
        TimeSpan dauer = uhrzeitEnde - uhrzeitStart;
        durschnittsWert /= durchlaeufe;

        for (int i = 0; i < besterCode.Count; i++)
        {
            Console.WriteLine(intArrayAusgeben(besterCode[i]));
        }

        // Sortieren und ausgeben
        List<int[]> sortiert = BitstringsSortieren.Start(besterCode);

        Console.WriteLine("Bester Aufruf mit " + besterWert.ToString() + " Elementen.");
        Console.WriteLine("Schlechtester Aufruf mit " + schlechtesterWert.ToString() + " Elementen.");
        Console.WriteLine("Durschnittlich mit " + durschnittsWert.ToString() + " Elementen.");
        Console.WriteLine("Vergangene Zeit: " + dauer.TotalMilliseconds.ToString() + "ms.");
        Console.WriteLine("Durchschnittliche Zeit pro Durchlauf: " + dauer.TotalMilliseconds / durchlaeufe + "ms.");

        // Loesung in Datei schreiben
        StreamWriter datei = new StreamWriter("algo_2-" + anzahlDerSchalter.ToString() + "-" + echteSchalter.ToString() + "-" + durchlaeufe.ToString() + ".txt", true);
        datei.WriteLine("Bester Aufruf mit " + besterWert.ToString() + " Elementen.");
        datei.WriteLine("Schlechtester Aufruf mit " + schlechtesterWert.ToString() + " Elementen.");
        datei.WriteLine("Durschnittlich mit " + durschnittsWert.ToString() + " Elementen.");
        datei.WriteLine("Vergangene Zeit: " + dauer.TotalMilliseconds.ToString() + "ms.");
        datei.WriteLine("Durchschnittliche Zeit pro Durchlauf: " + dauer.TotalMilliseconds / durchlaeufe + "ms.");

        for (int i = 0; i < besterCode.Count; i++)
        {
            datei.WriteLine(intArrayAusgeben(besterCode[i]));
        }

        datei.WriteLine("Sortierter Code:");
        for (int i = 0; i < sortiert.Count; i++)
        {
            datei.WriteLine(intArrayAusgeben(sortiert[i]));
        }

        datei.WriteLine("Kosten der unsortierten Liste: " + BitstringsSortieren.KostenListe(besterCode).ToString() + ".");
        datei.WriteLine("Kosten der sortierten Liste: " + BitstringsSortieren.KostenListe(sortiert).ToString() + ".");
        Console.WriteLine("Kosten der unsortierten Liste: " + BitstringsSortieren.KostenListe(besterCode).ToString() + ".");
        Console.WriteLine("Kosten der sortierten Liste: " + BitstringsSortieren.KostenListe(sortiert).ToString() + ".");
        datei.WriteLine("Durch das Sortieren wurden Kosten in Höhe von " + BitstringsSortieren.BerechneListenDifferenz(besterCode, sortiert).ToString() + " eingespart.");

        if (CodeUeberpruefen(besterCode))
        {
            datei.WriteLine("Der Code ist gültig.");
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\ZweiterAlgorithmus.cs 8

```
            }
            else
            {
                datei.WriteLine("Der Code ist UNGÜLTIG.");
            }
            datei.Close();

            Console.ReadLine();
        }

        public static bool CodeUeberpruefen(List<int[]> bitstrings)
        {
            if (mitMuster)
            {
                GeneriereAlleZugangscodesMuster();
            }
            else
            {
                GeneriereAlleZugangscodes();
            }

            int fehlendeFaelle = 0;

            foreach (SchalterAuswahl schalterAuswahl in listeSchalterAuswahl)
            {
                foreach (int[] stellung in schalterAuswahl.SchalterStellungen)
                {
                    bool passenderBitstringGefunden = false;

                    foreach (int[] bitstring in bitstrings)
                    {
                        bool codePasst = true;

                        // Teste diesen Bitstring
                        for (int i = 0; i < bitstring.Length; i++)
                        {
                            if (stellung[i] != 2 && bitstring[i] != stellung[i])
                            {
                                codePasst = false;
                                break;
                            }
                        }

                        if (codePasst)
                        {
                            passenderBitstringGefunden = true;
                            break;
                        }
                    }

                    if (!passenderBitstringGefunden)
                    {
                        //Console.WriteLine("Der folgende Fall wurde nicht beruecksichtigt:↙
    " + intArrayAusgeben(stellung) + ".");
                        fehlendeFaelle++;
                    }
                }
            }

            Console.WriteLine("Es fehlen " + fehlendeFaelle.ToString() + " Faelle.");

            if (fehlendeFaelle == 0)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\ZweiterAlgorithmus.cs 9

```
        static int CompareToIntArrayFrei(int[] array1, int[] array2)
        {
            int anzahl1 = 0;
            int anzahl2 = 0;

            for (int i = 0; i < array1.Length; i++)
            {
                if (array1[i] == 2) anzahl1++;
                if (array2[i] == 2) anzahl2++;
            }

            return anzahl2 - anzahl1;
        }

        private static Random rand = new Random();

        /// <summary>
        /// Mischt eine Liste zufaellig. (Wobei der Standard-Zufallsgenerator von .NET nicht besonders gut sein soll, hier koennte man u.U. noch was verbessern).
        /// </summary>
        public static void Shuffle<T>(IList<T> ilist)
        {
            int iIndex;
            T tTmp;
            for (int i = 1; i < ilist.Count; ++i)
            {
                iIndex = rand.Next(i + 1);
                tTmp = ilist[i];
                ilist[i] = ilist[iIndex];
                ilist[iIndex] = tTmp;
            }
        }

        /// <summary>
        /// Erstellt eine Wertekopie eines Integer-Arrays.
        /// </summary>
        public static int[] intArrayKopieren(int[] wert)
        {
            int[] result = new int[wert.Length];

            for (int i = 0; i < wert.Length; i++)
            {
                result[i] = wert[i];
            }

            return result;
        }

        public static bool sindIntArrayKompatibel(int[] array1, int[] array2)
        {
            for (int i = 0; i < array1.Length; i++)
            {
                if (array1[i] != array2[i] && array1[i] != 2 && array2[i] != 2) return false;
            }

            return true;
        }

        /// <summary>
        /// Wandelt ein Integer-Array in einen String um (fuer Debugzwecke und Ausgabe).
        /// </summary>
        public static string intArrayAusgeben(int[] wert)
        {
            string test = "{";

            for (int i = 0; i < wert.Length; i++)
            {
                test += wert[i].ToString() + ";";
```

```
            }

            test += "}";
            return test;
        }

        public static string listeAusgeben<T>(List<T> liste)
        {
            string test = "{";

            for (int i = 0; i < liste.Count; i++)
            {
                test += liste[i].ToString() + ";";
            }

            test += "}";
            return test;
        }

        public static List<int[]> listIntArrayKopieren(List<int[]> liste)
        {
            List<int[]> rueckgabe = new List<int[]>();

            for (int i = 0; i < liste.Count; i++)
            {
                rueckgabe.Add(intArrayKopieren(liste[i]));
            }

            return rueckgabe;
        }

        public static int[] bitstringErstellen(int anzahlSchalter)
        {
            int[] rueckgabe = new int[anzahlSchalter];

            for (int i = 0; i < anzahlSchalter; i++) rueckgabe[i] = 2;

            return rueckgabe;
        }
    }

    struct SchalterAuswahl
    {
        public int[] AusgewaehlteSchalter;

        public List<int[]> SchalterStellungen;

        public int CompareToAnzahlStellungen(SchalterAuswahl s1)
        {
            // TO-DO: ggf. umdrehen
            return -s1.SchalterStellungen.Count + SchalterStellungen.Count;
        }

        public bool IstKompatibel(SchalterAuswahl andereAuswahl)
        {
            for (int i = 0; i < AusgewaehlteSchalter.Length; i++)
            {
                for (int j = 0; j < andereAuswahl.AusgewaehlteSchalter.Length; j++)
                {
                    if (AusgewaehlteSchalter[i] == andereAuswahl.AusgewaehlteSchalter[j])  ↙
    return false;
                }
            }

            return true;
        }

        public bool EqualsAuswahl(int[] andereAuswahl)
        {
            foreach (int i in AusgewaehlteSchalter)
```

F:\Users\meister\Documents\Visual Studio ...\28_2_bwinf_1\ZweiterAlgorithmus.cs 11

```
            {
                if (!andereAuswahl.Contains(i)) return false;
            }

            return true;
        }

        public bool EntferneSchalterStellung(int[] stellung)
        {
            for (int aktuelleStellung = 0; aktuelleStellung < SchalterStellungen.Count;
    aktuelleStellung++)
            {
                // Pruefe aktuelle Stellung
                bool stellungGefunden = true;

                for (int aktuellerSchalter = 0; aktuellerSchalter < stellung.Length;
    aktuellerSchalter++)
                {
                    if (SchalterStellungen[aktuelleStellung][aktuellerSchalter] != 2 &&
    stellung[aktuellerSchalter] != SchalterStellungen[aktuelleStellung][aktuellerSchalter])
                    {
                        stellungGefunden = false;
                        break;
                    }
                }

                if (stellungGefunden)
                {
                    SchalterStellungen.RemoveAt(aktuelleStellung);
                    return true;
                }
            }

            return false;
        }

        public override bool Equals(object obj)
        {
            // Es muessen die gleichen Schalterindizes ausgewaehlt sein
            SchalterAuswahl vergleich = (SchalterAuswahl) obj;
            return EqualsAuswahl(vergleich.AusgewaehlteSchalter);
        }
    }
}
```