28. Bundeswettbewerb Informatik

– 2. Runde –

Von:
Matthias Linder

Verwaltungs-Nr:
28.0180.01

Schule:
Franz-Haniel-Gymnasium

Datum:
7. April 2010

Inhaltsverzeichnis

A. Allgemeines

In der folgenden Dokumentation werde ich meine Lösungen zu den Aufgaben der zweiten Runde des 28. Bundeswettbewerbs in Informatik (Jahr 2009/2010) vorstellen. Dabei habe ich in dieser Runde aus den drei möglichen Aufgaben die erste (*Universeller Öffnungscode*) und die zweite Aufgabe (*Turmrestaurant*) ausgewählt und bearbeitet. Die Einsendung besteht aus den beiden vollständig dokumentierten Aufgaben (d. h. Lösungsidee, Programmabläufe, ...) und einer CD-ROM, welche die zu der Lösung des jeweiligen Problems notwendigen Programme und deren Quellcodes enthält.

Verwendete Programmiersprachen

Zur Erstellung der Programme ist – wie zuvor in der ersten Runde auch – die objektorientierte Programmiersprache C# (*C-Sharp*) als Teil des *Microsoft .NET Frameworks* zum Einsatz gekommen. Während die Programme zur Ausführung nur die Version 2.0 des Frameworks benötigen, so ist zum Kompilieren des Quelltextes hingegen mindestens der .NET 3.5 Compiler notwendig. Begründet liegt dies in den verwendeten neueren Programmierparadigmen und Strukturen wie zum Beispiel Lambda-Ausdrücken, welche erst in späteren Versionen des .NET Frameworks aufgenommen wurden, jedoch in von .NET 2.0 ausführbaren Code umgesetzt werden können.

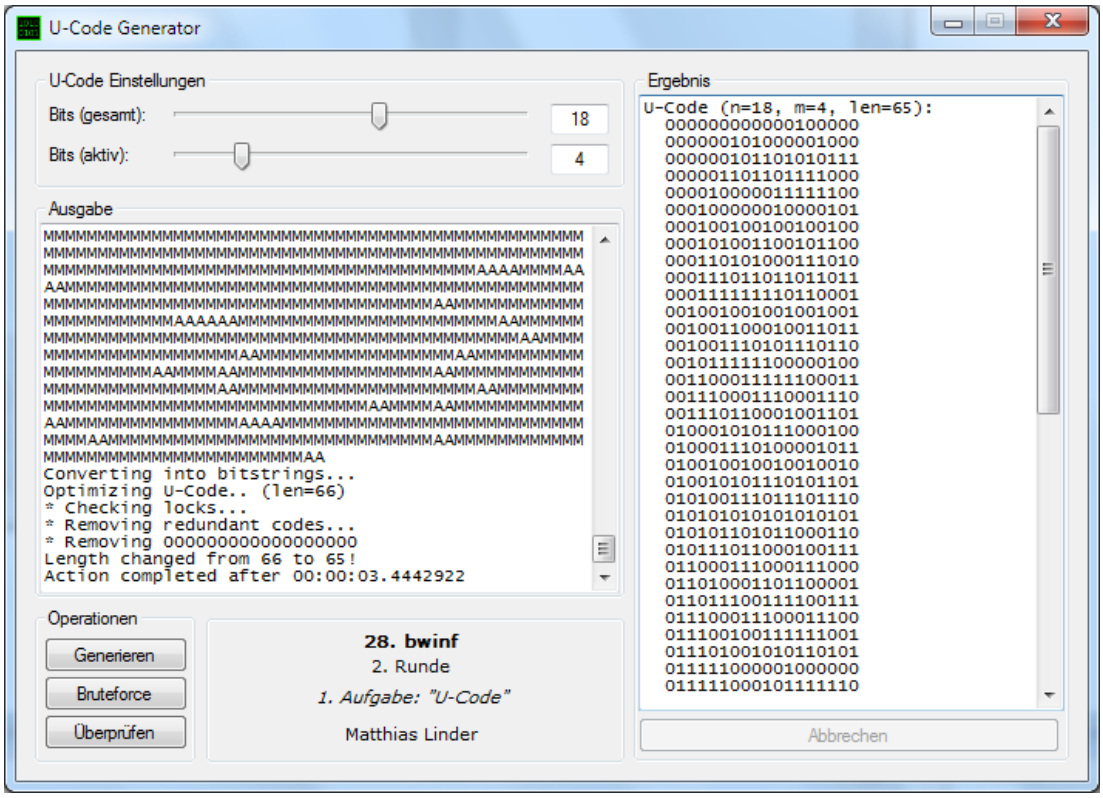
Die Wahl der Programmiersprache fiel einfach, da ich zum einem zuvor bereits hauptsächlich mit C# gearbeitet habe und ich mich dort dementsprechend auskenne, und andererseits das *.NET Framework* durch das weite Spektrum an vorhandenen Klassen und Methoden ein schnelles und effizientes Umsetzen der Lösungsidee ermöglicht. Die durch den Garbage-Collector entstehenden Performance-Verluste sind dabei auf heutigen Computern problemlos verkraftbar.

Testumgebung

Die kompilierten Programme wurden unter *Windows 7* auf einem Intel Core2Duo Q6600 Quadcore Prozessor mit 2,4GHz Rechenleistung getestet. Die Berechnungszeiten für die einzelnen Lösungen können daher auf anderen Rechnern durchaus kürzer oder länger sein. Auch wenn die Programme speichersparend sind, so können in einigen Teilschritten der Algorithmen größere Mengen an Speicher (etwa 512mb freier Arbeitsspeicher) notwendig sein.

Da in den Programmen nur Standardmethoden und -klassen aus dem .NET Framework verwendet wurden, sollten die Programme auch unter anderen Plattformen, welche das .NET Framework implementieren (z. B. mono unter Unix), problemlos funktionieren.

B. Aufgabe 1: Universeller Öffnungscode



1. Lösungsidee

Zunächst will ich zur Lösung dieser Aufgabe einige Grundlagen legen:

Es sei ein *Bitstring* L mit der *Länge* N gegeben, welcher durch eine Aneinanderreihung von N-Bits entsteht. Eine möglicher Bitstring der Länge $N = 6$ sei zum Beispiel „`101010`“. Eine solche vollständige Aneinanderreihung wird im Folgenden als „Code“ bezeichnet.

Weiterhin sei ein *Schloss* K gegeben, welches durch einen Bitstring beschrieben wird. Ergänzend seien hier jedoch nicht nur die Zustände 1 und 0, sondern auch der Zustand „-“ (unbenutzt) erlaubt. Die Anzahl der genutzten Bits sei dabei die Anzahl der aktiven Schalter bzw. Bits M des Schlosses. Ein Beispiel für ein Schloss mit $N = 6 \wedge M = 2$ sei „`-1---0-`“.

Ist die Länge des Bitstrings N (Gesamtanzahl der Schalter) und die Anzahl der aktiven Bits M für ein Schloss bekannt, so ist es möglich einen universellen Öffnungscode (kurz: *U-Code*) für diese Art von Schlössern zu bilden. Der U-Code$_{N,M}$ besteht dabei aus einer Gruppierung verschiedener Bitstrings und erlaubt es durch Testen aller im U-Code enthaltener Codes jedes erdenklichen Schlosses$_{N,M}$ zu öffnen. Die Länge eines U-Codes entspricht dabei der Anzahl der Codes (Bitstrings) in ihm. Ein U-Code ist genau dann vollständig, wenn es für jedes Schloss$_{N,M}$ mindestens einen passenden Code innerhalb des U-Codes gibt, welcher dieses öffnen kann.

1.1. Bruteforce – Eine Lösung?

Einen funktionierenden U-Code zu bilden ist dabei nicht all zu schwer: Im einfachsten Fall würde man alle 2^N möglichen Codes zu einem U-Code zusammen fassen. Während das bei $N = 2$ nur vier Codes sind, sind dies bei $N = 16$ schon 65536 – eine Menge an Codes, die ein normaler Mensch nur mit sehr viel Geduld durchprobieren kann. Es gilt also, die Menge der notwendigen Codes so kurz wie möglich zu halten.

Wird eine optimale Lösung für ein Problem gesucht, bietet sich als erstes ein Bruteforce-Algorithmus an. Mithilfe eines solchen Algorithmus kann jede mögliche Zusammenstellung der 2^N Codes geprüft und der kürzeste U-Code am Ende zurückgegeben werden. Da dieser Baum jedoch eine Höhe von N hätte, und jeweils N, $N-1$, $N-2$, ... Äste besitzt, ist die Komplexität mit $O(N!)$ zu hoch, um das Problem für höhere N's mithilfe eines Bruteforce-Algorithmus zu lösen.

Da dieses Problem vermutlich NP-Vollständig ist, verwerfe ich in diesem Fall den Gedanke eine optimale Lösung zu finden. Stattdessen werde ich versuchen eine möglichst gute Lösung mithilfe eines selbst entwickelten Algorithmus zu generieren.

1.2. Die Alternative: Berechnung aller möglichen Schlösser

Statt alle möglichen Codes zu berechnen, berechne ich daher stattdessen alle möglichen Schlösser. Die Anzahl der möglichen Schlösser ist dabei schnell errechnet: Es gibt $\binom{N}{M}$ mögliche Positionen um die aktiven Bits innerhalb eines Schlosses zu platzieren, und, da jedes benutzte Bit entweder '1' oder '0' sein kann, gibt es jeweils 2^M Möglichkeiten die aktiven Bits zu schalten. Die Anzahl der möglichen Schlösser K ist also:

$$n_{N,M}(K) = \binom{N}{M} \cdot 2^M \tag{1}$$

	$M=1$	$M=2$	$M=3$	$M=4$	$M=5$	$M=6$
$N=1$	2	–	–	–	–	–
$N=2$	4	4	–	–	–	–
$N=3$	6	12	8	–	–	–
$N=4$	8	24	32	16	–	–
$N=5$	10	40	80	80	32	–
$N=6$	12	60	160	240	192	64

Abbildung 1: Anzahl der möglichen Schlösserkombination bei gegebenem N und M

Die Anzahl der möglichen Elemente steigt dabei mit steigendem N und M auch stark an – haben wir also ein Problem gegen ein anderes getauscht? Nein. Einerseits ist der Anstieg der Komplexität bei $O(n_{N,M}(K))$ deutlich geringer als bei $O(N!)$: Der Bruteforce-Ansatz würde für $N = 6 \wedge M = 4$ nämlich etwa $6! = 720$ Operationen

benötigen, der hier vorgestellte Ansatz jedoch nur 240. Anderseits ist dieser Ansatz in linearer Zeit berechenbar, da alle Kombinationen nur nacheinander innerhalb des U-Codes untergebracht werden müssen und keine Rekursion erforderlich ist. Deshalb ist dieser Ansatz – im Gegensatz zu der Bruteforce-Methode – in einem realistischen Zeitrahmen berechenbar.

1.3. Kombination von Schlössern zu Codes

Die Beschreibung meiner Lösungsidee erfolgt dabei vorrangig an dem Beispiel $N = 4 \wedge M = 2$. Dazu möchte ich jedoch zunächst einige Werte und Bitstrings für dieses N und M darstellen damit die spätere Argumentation besser ersichtlich ist:

Möglichen Positionen für aktive Bits:	$\binom{4}{2} = 6$
Möglichen Werte für aktive Bits:	$2^2 = 4$
Anzahl d. Schlösser:	$6 \cdot 4 = 24$
Bruteforce:	0000 0011 0101 1001 1110
Eigene Lösung:	0000 0011 0101 1010 1100 1111

Schlösser:	11--	--11
	10--	--10
	01--	--01
	00--	--00
	1-1-	-1-1
	1-0-	-1-0
	0-1-	-0-1
	0-0-	-0-0
	1--1	-11-
	1--0	-10-
	0--1	-01-
	0--0	-00-

Abbildung 2: Schlösser und U-Codes für $N = 4 \wedge M = 2$

Die verschiedenen möglichen Schlösserkombinationen lassen sich rekursiv erzeugen: Man zähle jeweils die verbleibenden Bits und die zu verteilenden aktiven Bits, und füge in jedem Rekursionsschritt zu dem Bitstring entweder eine '0', eine '1' oder ein '-' hinzu. Macht man dies nun solange bis alle Bits verteilt sind, und achtet dabei auch darauf, dass die entsprechenden Zählvariablen nur aus gültige Werte bestehen, so erhält man alle gültigen Schlosskombinationen.

Um nun einen vollständigen U-Code zu erhalten gilt es alle möglichen Schlösserkombinationen innerhalb dieses U-Codes unterbringen. Wird jede Kombination innerhalb des U-Codes untergebracht, so ist dieser automatisch vollständig. Dieser Fakt wird in meinem Ansatz genutzt: Ich werde nacheinander alle Schlösser in den U-Code einfügen und lasse dabei kein Schloss aus – deshalb muss ich mir über die Vollständigkeit desselben keine mehr Sorgen machen – Zeit, sich auf die Generierung eines möglichst kurzen U-Codes zu konzentrieren!

Betrachtet man die verschiedenen möglichen Schlösser, so fällt auf, dass diese sich in ihrem Aufbau nicht zwangsläufig widersprechen. So können mehrere Schlösser, bei denen die gemeinsamen aktiven Bits entweder vom Wert komplett identisch sind (z. B. „`11--`“ und „`-10-`“) oder die unbelegten Bits durch das jeweilige andere Schloss ersetzt werden können (z. B. „`11--`“ und „`--11`“), zu einem einzigen Code zusammen gefasst werden (siehe Abbildung). Dadurch ist es später nicht mehr notwendig z. B. alle drei verschiedene Kombinationen zu prüfen, sondern es genügt den einen, zusammengesetzten Code einzugeben, da dieser bereits alle drei Kombinationen enthält. Es ist uns also möglich die 240 denkbaren Schlösser durch eine deutlich geringere Anzahl an Codes abzudecken.

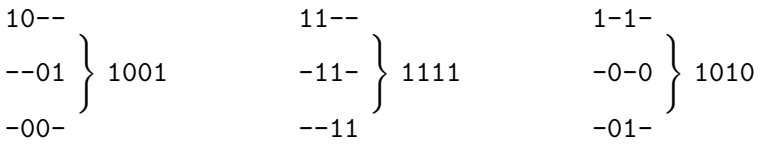


Abbildung 3: Kombinierbarkeit mehrerer Schlösser ($N = 4 \wedge M = 2$)

Gleichzeitig sei anzumerken, dass ein so gebildeter Code aber nicht nur die z. B. drei Schlösser, mit welchen er erweitert wurde, umfasst, sondern auch noch andere Kombinationen wie z. B. „`-a-a`“ oder „`a--a`“. Als Seiteneffekt werden also gleichzeitig noch viele andere, bisher nicht beachtete Codes abgedeckt.

Es gibt jedoch gibt es auch Schlösser, welche man nicht zu einem gemeinsamen Code erweitern kann. Dieser Fall tritt ein, sobald ein Schloss und der Code an der selben Stelle ein aktives Bit haben, dieses sich jedoch im Wert unterscheidet: „`11--`“ und „`-0-1`“ können daher nicht zu einem Code zusammen gefasst werden – Eine Kollision findet statt. In solch einem Fall muss der U-Code um einen neuen Code ergänzt werden, und die Länge des U-Codes erhöht sich:

U-Code	Schloss	Ergebnis
`----`	$\Leftarrow$ `11--`	`11--`
`11--`	$\not\Leftarrow$ `10--`	`11--`, `10--`
`11--`, `10--`	$\Leftarrow$ `--01`	`1101`, `10--`
`1101`, `10--`	$\Leftarrow$ `--11`	`1101`, `1011`

Abbildung 4: Kollision von Schloss und Code ($N = 4 \wedge M = 2$)

1.4. Sortierung der Schlösser als relevantes Kriterium

Die Grundlagen des Algorithmus sind nun geklärt: Es werden rekursiv alle Schlösser generiert, und diese werden nacheinander zum U-Code hinzugefügt. Wird das Schloss bereits durch den U-Code abgedeckt, so ist keine Aktion nötig. Findet hingegen eine

Kollision statt, so wird ein neuer Code in dem U-Code hinzugefügt; andernfalls wird der erstbeste Code mit dem Schloss erweitert.

Während dieser Algorithmus schon einen brauchbaren U-Code generiert, kann man ihn noch auf etwa 2/3 seiner ursprünglichen Größe verkleinern indem man die Schlösser entsprechend sinnvoller Kriterien vorsortiert.

Doch warum ist dem so? Dafür gibt es zwei relevante Gründe:

1. Lücken vermeiden:
Wählt man zum einen einfach wahllos verschiedene Schlösserkombinationen aus, so können sich recht schnell Lücken im U-Code bilden. So kann es sein, dass der Code „`1-01`“ entsteht, aber nur noch das Schloss „`-0-0`“ verteilt werden muss. Da der Code und das Schloss nicht ohne einen Konflikt kombinierbar sind, und daher das Schloss zusätzlich zum U-Code hinzugefügt werden muss, wird der U-Code unnötig vergrößert.

Sortiert man die Schlösser hingegen anhand ihrer Position, so ist die Gefahr, dass diese unbeabsichtigten Lücken entstehen, deutlich geringer. So kann „`11--`“ ohne Probleme mit „`-11-`“, und der entstehende Code wiederum mit „`--10`“ erweitert werden ohne das ein zweiter Code notwendig ist.

Die Position der aktiven Bits innerhalb eines Schlosses spielt also eine wichtige Rolle: Ergänzt man einen vorhandenen Code – wie zuvor gezeigt – indem man die Lücken gezielt ausfüllt, so hat man bereits nach einigen Schritten vollständige, lückenlose Codes, welche bereits einen Großteil der möglichen Kombinationen abdecken. Die Wahrscheinlichkeit, dass ein unnötig mit lücken-behafteter U-Code entsteht ist also deutlich geringer.

2. Redundante Teilcodes vermeiden:
Da es $\binom{4}{2} = 6$ Möglichkeiten gibt die aktiven Bits innerhalb des Bitstrings zu positionieren, kann ein einzelner Code im besten Fall also bis zu 6 dieser Möglichkeiten abdecken. Hat man jedoch einen größeren U-Code, so ist es möglich, dass zwei Codes die gleiche Kombination abdecken. Es spielt nämlich nicht nur die Position der aktiven Bits eine Rolle, sondern auch deren Wert. Je nach dem wie Schlösser mit den gleichen aktiven Bispositionen, aber mit unterschiedlichen Werten kombiniert werden, entstehen unterschiedlich effiziente Codes:

Schlösser		Kombination	Redundanz
`00--`	`--00`	`0000`	`-00-`
`10--`	`--01`	`1001`	`1--1`
`11--`	`--11`	`1111`	

Abbildung 5: Unoptimierte Kombination von Schlössern resultiert in Redundanz

Schlösser		Kombination	Redundanz
`00--`	`--00`	`0001`	
`10--`	`--01`	`1011`	`-0-1`
`11--`	`--11`	`1100`	

Abbildung 6: Optimierte Kombinationen sorgen für eine höhere Effizienz

Konfliktpotential

Doch wie vermeidet man diese Redundanz ohne viele aufwändige Berechnungen?

Je größer ein U-Code wird, desto mehr Redundanzen wird er automatisch enthalten - Ab einem gewissen Punkt ist es daher fast unmöglich noch einen neuen Code hinzuzufügen ohne dass er mit den schon vorhandenen Codes Gemeinsamkeiten besitzt, da bereits so viele verschiedene Kombinationen abgedeckt werden.

Erstellt man jedoch gleich zu Beginn des Vorgangs so viele Codes wie unbedingt notwendig sind – also Codes, welche im wahrscheinlichsten Fall gar nicht vermeidbar sind, da sie sich zum Beispiel widersprechen (z. B. „`11--`“ und „`00--`“) – so ist während des Ergänzungsvorgang die Wahrscheinlichkeit, dass ein Schloss bereits durch einen vorhandenen Code schon abgedeckt wird, und deswegen nicht erneut verteilt werden muss, durch die größere Anzahl an Codes höher.

Wenn ein Schloss bereits abgedeckt wird, ist die Möglichkeit einer weiteren Fehlentscheidung nicht mehr möglich. Mein Algorithmus zielt also darauf ab, die Wahrscheinlichkeit für Fehlentscheidungen (also Entscheidungen, welche sich im Nachhinein als schlecht herausstellen würden, und den Code nur unnötig verlängern) zu minimieren, in dem bereits am Anfang der Erstellung schrittweise notwendige und wichtige Codes hinzugefügt und anschließend soweit wie möglich vervollständigt werden.

N, M	Ohne Sortierung	Mit Sortierung
4, 2	6 // 5	6 // 6
9,3	35 // 26	18 // 18
18,4	259 // 93	66 // 65
25,5	1031 // 286	212 // 203

Abbildung 7: Auswirkung der Sortierung auf die Lösungsmenge (Unoptimiert // Optimiert)

1.5. Sortierungskriterien

Durch logische Strukturierung und längere Versuchsreihen haben sich daher folgende Sortierungskriterien – jeweils nach Priorität geordnet – etabliert:

1. Gruppenlänge, absteigend; 2. Position der Bits, aufsteigend

Sortiert man die gegebene Menge an Schlössern nach der Länge der längsten

Gruppe[1], und anschließend anhand der Position der aktiven Bits (Most-Left-Bit, Second MLB, ...), so erhält man bereits eine von der Positionierung her eindeutige Sortierung. Da durch diese Sortierungskriterien Sequenzen wie z. B. „`11--`“ „`-11-`“ „`--11`“ entstehen, wird der Code von links nach rechts aufgebaut bis sich ein vollständiger Code ergibt. Bei jedem Schritt muss im Normalfall nur ein Bit an einem vorhandenen Code hinzugefügt werden. Codes wie „`1-1-`“, welche eine geringere Gruppenlänge als die zu erst erwähnten aufweisen, werden bereits durch den vorher vervollständigten Code abgedeckt.

3. Inverses Element des Schlosses

Neben den Positionen muss auch der Wert der Bits beachtet werden. Das inverse Element eines jeden Schlosses, bei dem jedes Bit einmal negiert wird (z. B. „`00--`“ für „`11--`“), spielt eine gesonderte Rolle, da es sozusagen einen 100% -igen Konflikt mit dem zuvor erstellten Code darstellt, und daher den U-Code um ein unbedingt nötiges Element ergänzt. Dies ist in diesem Fall kein eigentlicher Sortierungsschritt - Stattdessen werden in einem zweiten Schritt jeweils das normale und das inverse Element paarweise in die Schlösserliste eingefügt.

4. Anzahl der Einsen, aufsteigend; 5. Integer-Wert des Schlosses, aufsteigend

Die beiden übrigen Sortierungskriterien sorgen zum einen dafür, die Anzahl der Konflikte gezielt zu Beginn der Generierung in die Höhe zu treiben und diese später zu unterbinden, und dienen zum anderen dem Zweck eine eindeutige bzw. stabile Sortierung zu erreichen.

Durch Anwendung dieser Sortierkriterien kann der U-Code auf bis zu 28% seiner ursprünglichen Größe optimiert werden.

Optimerung des Codes

Der so entstehende U-Code ist zwar schon recht kurz, kann aber teilweise noch weiter optimiert werden. Dazu wird der generierte U-Code zunächst analysiert und überflüssige Codes anschließend entfernt. Ein Code ist genau dann überflüssig, wenn er nicht als einziges ein bestimmtes Schloss öffnen kann, d.h. solange es noch einen weiteren Code gibt, der auch dieses Schloss öffnen kann, kann der erste Code entfernt werden.

Um diese überflüssigen Codes zu finden werden daher in einem dritten Schritt über eine einfache Iteration zu jedem Schloss alle Codes herausgesucht, welche das jeweilige Schloss öffnen können. In einem weiteren Teilschritt werden dann - zunächst unter der Annahme, dass alle Codes überflüssig seien – die Codes, welche als einzige Lösung eines Schlosses existieren, aus dieser Annahme wiederum entfernt. Alle Codes aus der Annahme, welche in diesem Verfahren nicht entfernt wurden, sind potentiell überflüssig.

[1] Eine Gruppe ist in diesem Fall eine ununterbrochene Folge von genutzten Bits

Aus dieser Liste aller überflüssigen Kandidaten kann jeweils aber nur ein Element entfernt werden, da durch dessen Eliminierung die vorgegangene Analyse wieder invalidiert wird. Deshalb wird in jeder Iteration nur ein einzelner Code entfernt, und die verbleibende (potentielle) Annahmemenge solange erneut analysiert, bis sie leer ist.

Wendet man dieses Optimierungsverfahren an, so wird der U-Code erneut um etwa 5% verkürzt, und es entsteht ein zureichend kurzer U-Code.

2. Dokumentation

Das zuvor vorgestellte Verfahren wurde dementsprechend in einer C# Anwendung umgesetzt. Jede Datei entspricht dabei einer Klasse:

Program: Startklasse. Enthält die Startfunktion "Main", Log-Methoden und öffnet das Programmfenster.

MainForm: Programmfenster. Stellt die visuelle Benutzerschnittstelle dar und verbindet diese mit den entsprechenden anderen Klassen.

Util: Stellt einzelne Hilfsmethoden bereit, welche von den anderen Klassen verwendet werden.

UniversalCode: Beschreibt eine Menge von Codes als U-Code anhand der Gesamtanzahl und Anzahl der aktiven Schalter (Containerklasse).

CodeLock: Symbolisiert ein Schloss anhand eines *bool*?-Arrays. Erlaubt die Generierung aller Schlösser für ein gegebenes N, M und bietet Methoden zum Überprüfen eines U-Codes.

Bruteforce: Algorithmus, welcher eine Lösung des Problems anhand eines Bruteforce-Verfahrens (Tiefensuche, Backtracking) berechnet. Nur für sehr kleine N, M's effektiv anwendbar.

Generator: Stellt die Implementierung des zuvor dargestellten Algorithmus (siehe Lösungsidee) dar. Jede Methode dieser Klasse stellt einen Teilschritt des Algorithmus dar.

Zu jeder Klasse und deren Methoden befinden sich dabei in den XML-Kommentaren und im Quelltext weitere Hinweise/Beschreibungen zur Funktionalität und Umsetzung der Lösungsidee.

3. Erweiterungen

Bei der Umsetzung wurden dabei einige, als sinnvoll erscheinende Erweiterungen zu der ursprünglichen Aufgabenstellung betrachtet und umgesetzt. Diese seien kurz vorgestellt:

Beliebiger U-Code

Selbstverständlich funktioniert der Algorithmus nicht nur für den gewünschten Fall ($N = 25 \wedge M = 4$), sondern auch für alle anderen erdenkbaren (und gültigen) Kombinationen. Daher bietet die Benutzerschnittstelle dem Benutzer die Möglichkeit die relevanten Parameter (N und M) zu variieren. Die GUI sorgt dabei dafür, dass nur gültige Werte ($N > 0 \wedge M <= N$) eingegeben werden können, und passt die vorhandenen Schieberegler bei Änderung der Werte automatisch an.

Bruteforce

Um eine Vergleichsmöglichkeit zu haben ist in dem Programm auch ein Bruteforce-Algorithmus implementiert, welcher eine der optimalen Lösungen für ein gegebenes N, M ausrechnet. Die Qualität des verwendeten Algorithmus kann daher so immer in Abhängigkeit von der optimalen Lösung betrachtet werden – Jedoch nur solange N und M nicht größere Werte als 8 annehmen, da ab diesem Zeitpunkt der Rechenaufwand für die Bruteforce-Lösung bereits Zeit im Stundenbereich veranschlagt.

Lösungsüberprüfung

Auch wenn der von mir verwendete Algorithmus auf dem Fakt beruht, dass die von ihm generierten Lösungen automatisch vollständig sind, ist es dennoch ratsam diese Vollständigkeit gelegentlich zu prüfen, da auch hier – wie sonst auch überall – Fehler in der Programmierung nicht ausgeschlossen werden können. Deshalb bietet die GUI dem Nutzer die Möglichkeit den generierten U-Code auf Vollständigkeit zu überprüfen. Wird dabei ein Fehler (ein Schlosskombination, welches durch keinen Code des U-Codes geöffnet werden kann) festgestellt, so wird dies dem Nutzer entsprechend mitgeteilt.

Weiterhin ist es dem Nutzer möglich das Textfeld mit dem U-Code zu editieren und einen eigenen U-Code zum Testen in dieses Feld einzutragen, und diesen anschließend auch auf Vollständigkeit zu überprüfen. Die Testmöglichkeit ist also nicht auf durch das Programm generierte U-Codes beschränkt; benötigt aber einen U-Code in dem entsprechend gleichem Format.

4. Bedienung

Die Bedienung des Programms ist selbsterklärend: Der Nutzer muss nur die Parameter N, M entsprechend seiner Wünsche mithilfe der Schieberegler/Textfelder einstellen, und anschließend auf den „Generieren“-Button klicken um die gewünschte Lösung zu erhalten. Der Berechnungsfortschritt ist dabei im linken Fenster ersichtlich; die Lösung bei Abschluss der Berechnung anschließend in der rechten Textbox. Bei Bedarf kann die Lösung noch anhand des „Überprüfen“-Buttons auf Vollständigkeit überprüft werden.

Da die Berechnung auf einem sekundären Hintergrundsthread abläuft, bleibt die Benutzerschnittstelle während der Berechnung weitestgehend bedienbar, und erlaubt

auch ein Abbrechen der Operation zu jeder Zeit.

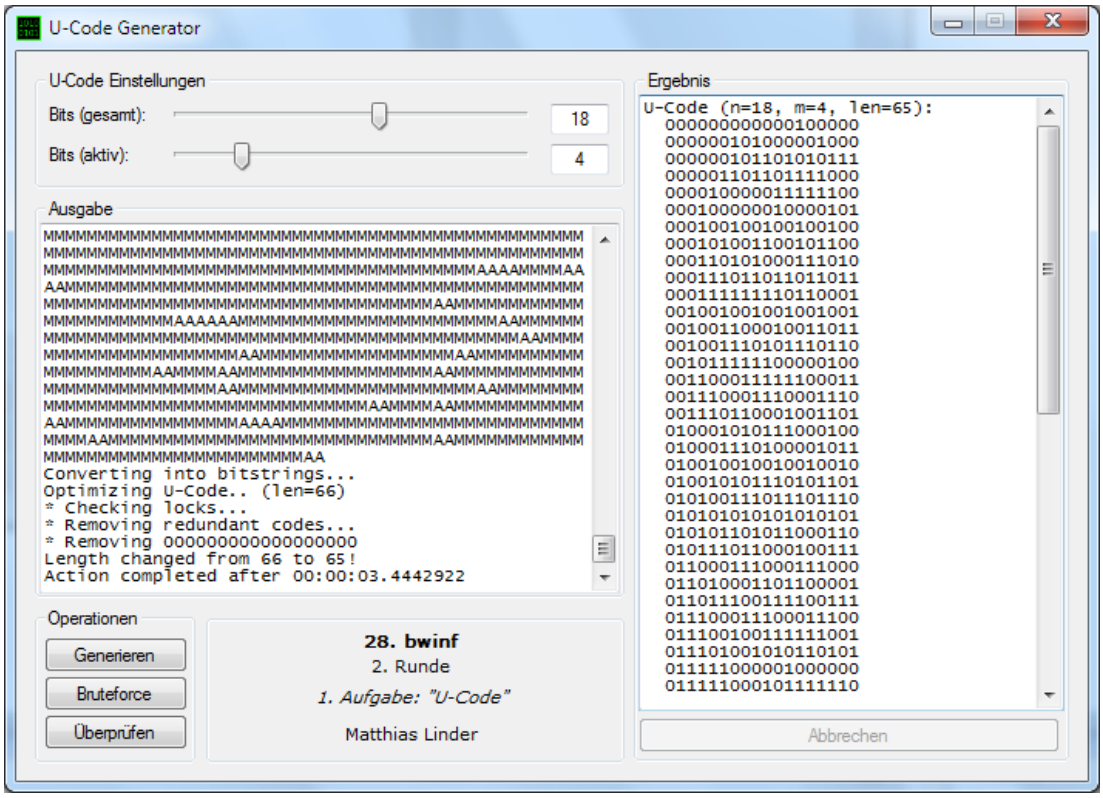


Abbildung 8: GUI

5. Ablaufprotokolle

Auf den folgenden Seiten finden sich einige Ablaufprotokolle zu relevanten/vergleichbaren Werten für N und M. Zu jedem Protokoll wird anschließend ein kurzer Kommentar abgegeben.

5.1. U-Code$_{4,2}$: Vergleich der Implementierungen

Mein Algorithmus

```
Solving problem using Generator!
Retrieving possible locks ..
Sorting by value ...
Removing inverse elements ...
Sorting (main sort) ...
Adding inverse elements ...
Creating final lock list ...
1. Lock: 10--
2. Lock: 01--
3. Lock: 11--
4. Lock: 00--
5. Lock: -10-
6. Lock: -01-
7. Lock: -11-
8. Lock: -00-
9. Lock: --10
10. Lock: --01
11. Lock: --11
12. Lock: --00
13. Lock: 1-0-
14. Lock: 0-1-
15. Lock: 1-1-
16. Lock: 0-0-
17. Lock: 1--0
18. Lock: 0--1
19. Lock: 1--1
20. Lock: 0--0
21. Lock: -1-0
22. Lock: -0-1
23. Lock: -1-1
24. Lock: -0-0
Inserting codes...
M = code modified; A = code added
AAAAMMMMMMMMAAMM
```

```
Converting into bitstrings...
Optimizing U-Code.. (len=6)
* Checking locks...
* Removing redundant codes...
Length changed from 6 to 6!
Action completed after 00:00:00.0894903
```

```
U-Code (n=4, m=2, len=6):
  0000
  0011
  0101
  1010
  1100
  1111
```

Aufgrund der noch geringen Komplexität dieses Beispieles sieht man in diesem Protokoll gut wie die Schlösser entsprechend sortiert werden, und wie der U-Code schrittweise aufgebaut wird. Zunächst werden 4 vollständige Codes erstellt; gegen Ende müssen noch zwei neue hinzugefügt werden, da die vorhandenen Codes die Schlösser nicht mehr abdecken können. Optimiert werden kann erst bei höheren Werten für N und M.

Bruteforce Algorithmus

```
Solving problem using brute force!
Generating all possible combinations...
Generate all possible locks...
Starting recursion!
len=13
len=12
len=11
len=10
len=9
len=8
len=7
len=6
len=5
Action completed after 00:00:00.0289209

U-Code (n=4, m=2, len=5):
  0000
  0011
  0101
  1001
  1110
```

Da hier noch mit recht kleinen N und M's gearbeitet wird, ist es noch möglich eine Lösung mithilfe von Backtracking zu finden. Im Vergleich zu dem Hauptalgorithmus wird hier ein Code weniger benötigt. Dieser Kompromiss ist jedoch akzeptabel, da der erstellte Algorithmus dafür bei hohen Wertebereichen für die beiden Parameter N, M deutlich schneller eine Lösung generieren kann.

5.2. U-Code$_{9,3}$: Vergleich mit der Musterlösung

```
Solving problem using Generator!
Retrieving possible locks ..
Sorting by value ...
Removing inverse elements ...
Sorting (main sort) ...
Adding inverse elements ...
Creating final lock list ...
1. Lock: 100------
2. Lock: 011------
3. Lock: 101------
4. Lock: 010------
5. Lock: 110------
6. Lock: 001------
7. Lock: 111------
8. Lock: 000------
9. Lock: -100-----
10. Lock: -011-----
11. Lock: -101-----
12. Lock: -010-----
13. Lock: -110-----
14. Lock: -001-----
15. Lock: -111-----
16. Lock: -000-----
17. Lock: --100----
18. Lock: --011----
19. Lock: --101----
20. Lock: --010----
21. Lock: --110----
22. Lock: --001----
23. Lock: --111----
24. Lock: --000----
25. Lock: ---100---
26. Lock: ---011---
27. Lock: ---101---
28. Lock: ---010---
29. Lock: ---110---
30. Lock: ---001---
31. Lock: ---111---
32. Lock: ---000---
33. Lock: ----100--
34. Lock: ----011--
35. Lock: ----101--
36. Lock: ----010--
37. Lock: ----110--
38. Lock: ----001--
39. Lock: ----111--
40. Lock: ----000--
41. Lock: -----100-
42. Lock: -----011-
43. Lock: -----101-
44. Lock: -----010-
45. Lock: -----110-
46. Lock: -----001-
47. Lock: -----111-
48. Lock: -----000-
49. Lock: ------100
50. Lock: ------011
51. Lock: ------101
56. Lock: ------000
61. Lock: 11-0-----
66. Lock: 01--1----
71. Lock: 11--1----
76. Lock: 01---0---
81. Lock: 10----0--
86. Lock: 00----1--
91. Lock: 10-----1-
96. Lock: 00-----0-
101. Lock: 11------0
106. Lock: 0-11-----
111. Lock: 1-11-----
116. Lock: 0--10----
121. Lock: 1---00---
126. Lock: 0---01---
131. Lock: 1----01--
136. Lock: 0----00--
141. Lock: 1-----10-
146. Lock: 0------11
151. Lock: 1------11
156. Lock: -01-0----
161. Lock: -10--0---
166. Lock: -00--1---
171. Lock: -10---1--
176. Lock: -00---0--
181. Lock: -11----0-
```

```
186. Lock: -01-----1
191. Lock: -11-----1
196. Lock: -0-10----
201. Lock: -1--00---
206. Lock: -0--01---
211. Lock: -1---01--
216. Lock: -0---00--
221. Lock: -1----10-
226. Lock: -0-----11
231. Lock: -1-----11
236. Lock: --01-0---
241. Lock: --10--0--
246. Lock: --00--1--
251. Lock: --10---1-
256. Lock: --00---0-
261. Lock: --11----0
266. Lock: --0-11---
271. Lock: --1-11---
276. Lock: --0--10--
281. Lock: --1---00-
286. Lock: --0---01-
291. Lock: --1----01
296. Lock: --0----00
301. Lock: ---11-0--
306. Lock: ---01--1-
311. Lock: ---11--1-
316. Lock: ---01---0
321. Lock: ---1-00--
326. Lock: ---0-01--
331. Lock: ---1--01-
336. Lock: ---0--00-
341. Lock: ---1---10
346. Lock: ----01-1-
351. Lock: ----11-1-
356. Lock: ----01--0
361. Lock: ----1-00-
366. Lock: ----0-01-
371. Lock: ----1--01
376. Lock: ----0--00
381. Lock: -----11-0
386. Lock: -----0-11
391. Lock: -----1-11
396. Lock: 0-1-0----
401. Lock: 1-0--0---
406. Lock: 0-0--1---
411. Lock: 1-0---1--
416. Lock: 0-0---0--
421. Lock: 1-1----0-
426. Lock: 0-1-----1
431. Lock: 1-1-----1
436. Lock: 0--1-0---
441. Lock: 1--0--0--
446. Lock: 0--0--1--
451. Lock: 1--0---1-
456. Lock: 0--0---0-
461. Lock: 1--1----0
466. Lock: 0---1-1--
471. Lock: 1---1-1--
476. Lock: 0---1--0-
481. Lock: 1---0---0
486. Lock: 0---0---1
491. Lock: 1----0-1-
496. Lock: 0----0-0-
501. Lock: 1----1--0
506. Lock: 0-----1-1
511. Lock: 1-----1-1
516. Lock: -0-1-0---
521. Lock: -1-0--0--
526. Lock: -0-0--1--
531. Lock: -1-0---1-
536. Lock: -0-0---0-
541. Lock: -1-1----0
546. Lock: -0--1-1--
551. Lock: -1--1-1--
556. Lock: -0--1--0-
561. Lock: -1--0---0
566. Lock: -0--0---1
571. Lock: -1---0-1-
576. Lock: -0---0-0-
581. Lock: -1---1--0
586. Lock: -0----1-1
591. Lock: -1----1-1
596. Lock: --0-1-0--
601. Lock: --1-0--0-
606. Lock: --0-0--1-
611. Lock: --1-0---1
616. Lock: --0-0---0
621. Lock: --1--1-0-
626. Lock: --0--1--1
631. Lock: --1--1--1
636. Lock: --0---1-0
641. Lock: ---1-0-0-
646. Lock: ---0-0-1-
651. Lock: ---1-0--1
656. Lock: ---0-0--0
661. Lock: ---1--1-0
666. Lock: ----0-1-1
671. Lock: ----1-1-1
Inserting codes...
M = code modified; A = code added
AAAAAAAAMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMAAAAMMMMM
MMMMMMMMMMMMMMMMMMAAMMAAMMMMMMM
MMMMMMMMMMMMMAAMMMMMM
Converting into bitstrings...
Optimizing U-Code.. (len=18)
* Checking locks...
* Removing redundant codes...
Length changed from 18 to 18!
Action completed after 00:00:00.7483758

U-Code (n=9, m=3, len=18):
  000000000
  000001111
  000101010
  000111100
  001010101
  001011010
  010011001
  011001100
  011110111
  100001000
  100110011
  101100110
  110100101
  110101010
  111000011
  111010101
  111110000
  111111111
```

Mit 18 benötigten Codes kommt der Algorithmus erstaunlich gut an die Musterlösung heran, welche nur 16 Codes benötigt. Die Sortierung der Schlösser wird nur noch Stufenweise ausgegeben, da sonst das Protokoll zu lang werden würde. Auch hier kann noch nichts optimiert werden.

5.3. U-Code$_{18,4}$: Weniger als 100?

```
Solving problem using Generator!
Retrieving possible locks ..
Sorting by value ...
Removing inverse elements ...
Sorting (main sort) ...
Adding inverse elements ...
Creating final lock list ...
1. Lock: 1000----------
2. Lock: 0111----------
3. Lock: 1001----------
4. Lock: 0110----------
5. Lock: 1010----------
6. Lock: 0101----------
7. Lock: 1100----------
8. Lock: 0011----------
9. Lock: 1011----------
10. Lock: 0100----------
11. Lock: 1101----------
12. Lock: 0010----------
13. Lock: 1110----------
14. Lock: 0001----------
15. Lock: 1111----------
16. Lock: 0000----------
17. Lock: -1000----------
18. Lock: -0111----------
19. Lock: -1001----------
20. Lock: -0110----------
21. Lock: -1010----------
22. Lock: -0101----------
23. Lock: -1100----------
24. Lock: -0011----------
25. Lock: -1011----------
26. Lock: -0100----------
27. Lock: -1101----------
28. Lock: -0010----------
29. Lock: -1110----------
30. Lock: -0001----------
31. Lock: -1111----------
32. Lock: -0000----------
33. Lock: --1000----------
34. Lock: --0111----------
35. Lock: --1001----------
36. Lock: --0110----------
37. Lock: --1010----------
38. Lock: --0101----------
39. Lock: --1100----------
40. Lock: --0011----------
41. Lock: --1011----------
42. Lock: --0100----------
43. Lock: --1101----------
44. Lock: --0010----------
45. Lock: --1110----------
46. Lock: --0001----------
47. Lock: --1111----------
48. Lock: --0000----------
49. Lock: ---1000----------
50. Lock: ---0111----------
51. Lock: ---1001----------
433. Lock: 100----------0-
815. Lock: -111--------1-----
1197. Lock: --111------0------
1579. Lock: ---110------1-----
1961. Lock: ----101--------1--
2343. Lock: -----1---100------
2725. Lock: -------101--0-----
3107. Lock: --------1------001
3489. Lock: ----------1---000
3871. Lock: 11--1--1----------
4253. Lock: 11----1-0---------
4635. Lock: 11------0----1----
5017. Lock: 10---------1----1-
5399. Lock: 1-10--------0-----
```

```
5781. Lock: 1--01------0------   26409. Lock: ---------1---01--1  47037. Lock: ------1-1-1------0    001011111100000100
6163. Lock: 1---00------1-----   26791. Lock: ----------1--1--00  47419. Lock: ------1--1--0----1    001100011111100011
6545. Lock: 1----00--------0--   27173. Lock: 1-0-1----0--------  47801. Lock: ------1-----0-1--1    001110001110001110
6927. Lock: 1-----1--11-------   27555. Lock: 1-0---0-----1-----  48183. Lock: -------1--1--0---0    001110110001001101
7309. Lock: 1-------11-0------   27937. Lock: 1-0------0--0-----  48565. Lock: --------1--0-1-0--    010001010111000100
7691. Lock: 1--------1-----01-   28319. Lock: 1-1----------1---1  48947. Lock: -----------1-0-0-1    010001110100001011
8073. Lock: 1-----------0--11-   28701. Lock: 1--1--1----------0  Inserting codes...               010010010010010010
8455. Lock: -11--0---0--------   29083. Lock: 1--1-----0-------1  M = code modified; A = code added  010010101110101101
8837. Lock: -10----1----0-----   29465. Lock: 1---0-1---1-------  AAAAAAAAAAAAAAAAAMMMMMMMMMMMMMMMMM 010100111011101110
9219. Lock: -10-------0-1-----   29847. Lock: 1---1---0--------0  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM 010101010101010101
9601. Lock: -10------------00-   30229. Lock: 1---0--------1--0-  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM 010101101011000110
9983. Lock: -1-1---------11---   30611. Lock: 1----0---0---1----  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM 010111011000100111
10365. Lock: -1--1----------10-  30993. Lock: 1----0--------0--0  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM 011000111000111000
10747. Lock: -1----10---1------  31375. Lock: 1-----1----1-1----  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM 011010001101100001
11129. Lock: -1-----0-11-------  31757. Lock: 1------1--1---0---  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM 011011100111100111
11511. Lock: -1-------10---0---  32139. Lock: 1-------1-0------1  MMAAAAMMMMAAAAAMMMMMMMMMMMMMMMMMM 011100011100011100
11893. Lock: -1---------0-10---  32521. Lock: 1--------0--1----1  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM 011100100111111001
12275. Lock: --10-0--------1---  32903. Lock: 1----------1-0--0  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMAAMM 011101001010110101
12657. Lock: --10---0---------0  33285. Lock: -1-0---1-0--------  MMMMMMMMMMMMMMMMMMMMMAAAAAAAMMMMMM 011111000001000000
13039. Lock: --11------1-----1-  33667. Lock: -1-0------0-1-----  MMMMMMMMMMMMMMMMMAAMMMMMMMMMMMMMM 011111000101111110
13421. Lock: --1-11--0---------  34049. Lock: -1--0-0----0------  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM 011111101110010000
13803. Lock: --1--10----1------  34431. Lock: -1--1---1--------1  MMAAMMMMMMMMMMMMMMMMMMMMMAAMMMMMM 100000010001101111
14185. Lock: --1---01--------1-  34813. Lock: -1--1--------1--0-  MMMMMMMMMMMMAAMMMMMMMMMMMMMMMMMM 100000111010000001
14567. Lock: --1----1------00--  35195. Lock: -1---1---0---1----  AAMMMMAAMMMMMMMMMMMMMMMMAAMMMMMM 100000111110111111
14949. Lock: --1------0--10----  35577. Lock: -1---0--------1--1  MMMMMMMMMMMMMMMMMMMMAAMMMMMMMMMM 100010110101001010
15331. Lock: --1---------00--1-  35959. Lock: -1-----1----0--0---  MMMMMMMMMMMMAAMMMMMMMMMMMMMMMMMM 100011011000000110
15713. Lock: ---10--0--0-------  36341. Lock: -1------0--1----0--  MMMMMMMMMMMMMMMMMAAMMMAAMMMMMMMM 100011100011100011
16095. Lock: ---11----1------1-  36723. Lock: -1------0--0-1----  MMMMAAMMMMMMMMMMMMMMMAAAAMMMMMMM 100100011000011000
16477. Lock: ---11---------10--  37105. Lock: -1-------0---0-0--  MMMMMMMMMMMMMMMMMMMMMMAAMMMMMMMM 100101110010011110
16859. Lock: ---1-1--------01-   37487. Lock: -1----------1-1--1  MMMMMMMMMMMMMMMMMMMMMAAMMMMMMMMM 100111000111000111
17241. Lock: ---1---01-----1---  37869. Lock: --1-1---1---0-----  MMMMMMMMMMMMMMMMMMMMMMMMMAA       101000100111011000
17623. Lock: ---1----1------00-  38251. Lock: --1-1-------0-1---  Converting into bitstrings...     101010010100111001
18005. Lock: ---1--------01-0---  38633. Lock: --1--0--1------1--  Optimizing U-Code.. (len=66)      101010101010101010
18387. Lock: ----10-0----1-----  39015. Lock: --1--1------0----0  * Checking locks...              101011000100010001
18769. Lock: ----10----0-0-----  39397. Lock: --1---0---1---0---  * Removing redundant codes...     101101010001010010
19151. Lock: ----11--------1--1  39779. Lock: --1----0-0------1-  * Removing 000000000000000000     101101101101101101
19533. Lock: ----1--11---0-----  40161. Lock: --1-----0-0--0----  Length changed from 66 to 65!     101110001011110100
19915. Lock: ----1---1----01---  40543. Lock: --1------1-1----1-  Action completed after 00:00:03.4183696 101001001110110010
20297. Lock: ----1-----0----11-  40925. Lock: --1--------1-1---0                                   110001110001110001
20679. Lock: -----11-0--0------  41307. Lock: ---1-1--0-------1  U-Code (n=18, m=4, len=65):       110011100000011100
21061. Lock: -----10----1--0---  41689. Lock: ---1-0-------1--1-    000000000000100000            110100000011111011
21443. Lock: -----1-00------1--  42071. Lock: ---1--1---0-----0-    000000101000001000            110110001010001001
21825. Lock: -----1--0-------00  42453. Lock: ---1---0--1-0-----    000000101101010111            110110011101100100
22207. Lock: -----1-----11-1---  42835. Lock: ---1----0-0----1--    000001101101111000            110110110110110110
22589. Lock: ------11-1----0---  43217. Lock: ---1-----0--0--0--    000010000011111100            111000000001001110
22971. Lock: ------11-----0--1-  43599. Lock: ---1-------1--1--1    000100000010000101            111000100100100100
23353. Lock: ------1--01---1---  43981. Lock: ----1-1---1----0--    000100100100100100            111010110011010011
23735. Lock: ------1----1-00---  44363. Lock: ----1--1-0-------1    000101001100101100            111011011011011011
24117. Lock: -------10--1-0----  44745. Lock: ----1---0-1---1---    000110101000111010            111101111100000011
24499. Lock: -------1-00-----1-  45127. Lock: ----1----1--0-0---    000111011011011011            111111010010101000
24881. Lock: -------1---0---00-  45509. Lock: ----1------0--1--0    000111111110110001            111111111111111111
25263. Lock: --------11--1---1-  45891. Lock: -----1-0----0-1---    001001001001001001
25645. Lock: --------1--11---0-  46273. Lock: -----1--0---0----0    001001100010011011
26027. Lock: ---------11--0-1--  46655. Lock: -----1----1-1--1--    001001110101110110
```

In diesem Fall kann das erste mal ein Code wegoptimiert werden. Während in den ersten Versionen des Algorithmus der Optimierungsteil noch deutlich mehr Codes wegoptimieren konnte, sorgt die gewählte Sortierung schon dafür, dass dies nur noch seltener notwendig ist. In Anbetracht der Tatsache, dass die Optimierungsfunktion bis zu $O(n!)$ Schlösser überprüfen muss, ist dieses Verhalten wünschenswert.

Der U-Code benötigt mit 65 Codes „deutlich weniger als 100“ und erfüllt damit auch dieses Kriterium; ist also – soweit ersichtlich – akzeptabel.

5.4. U-Code$_{25,5}$: Geforderte Lösung

```
Solving problem using Generator!
Retrieving possible locks ..
Sorting by value ...
Removing inverse elements ...
Sorting (main sort) ...
Adding inverse elements ...
Creating final lock list ...
1. Lock: 10000--------------------
2. Lock: 01111--------------------
3. Lock: 10001--------------------
4. Lock: 01110--------------------
5. Lock: 10010--------------------
6. Lock: 01101--------------------
7. Lock: 10100--------------------
8. Lock: 01011--------------------
9. Lock: 11000--------------------
10. Lock: 00111--------------------
11. Lock: 10011--------------------
12. Lock: 01100--------------------
13. Lock: 10101--------------------
14. Lock: 01010--------------------
15. Lock: 10110--------------------
16. Lock: 01001--------------------
17. Lock: 11001--------------------
18. Lock: 00110--------------------
19. Lock: 11010--------------------
20. Lock: 00101--------------------
21. Lock: 11100--------------------
22. Lock: 00011--------------------
23. Lock: 10111--------------------
24. Lock: 01000--------------------
25. Lock: 11011--------------------
26. Lock: 00100--------------------
27. Lock: 11101--------------------
28. Lock: 00010--------------------
29. Lock: 11110--------------------
30. Lock: 00001--------------------
31. Lock: 11111--------------------
32. Lock: 00000--------------------
33. Lock: -10000-------------------
34. Lock: -01111-------------------
35. Lock: -10001-------------------
36. Lock: -01110-------------------
37. Lock: -10010-------------------
38. Lock: -01101-------------------
39. Lock: -10100-------------------
40. Lock: -01011-------------------
41. Lock: -11000-------------------
42. Lock: -00111-------------------
43. Lock: -10011-------------------
44. Lock: -01100-------------------
45. Lock: -10101-------------------
46. Lock: -01010-------------------
47. Lock: -10110-------------------
48. Lock: -01001-------------------
49. Lock: -11001-------------------
50. Lock: -00110-------------------
51. Lock: -11010-------------------
13333. Lock: ---------------1-1100----
26615. Lock: 1-----0------111---------
39897. Lock: -11------011-------------
53179. Lock: --111----0--------1------
66461. Lock: --1-------------1-----110
79743. Lock: ---1----------1--111-----
93025. Lock: ----1-----------0--000---
106307. Lock: ------100----0-------1---
119589. Lock: -------1---0-010---------
132871. Lock: ---------1-0-------100---
146153. Lock: ------------11-----000---
159435. Lock: 10-0--------11-----------
172717. Lock: 10---1-----------0------1
185999. Lock: 10------1----------1--0--
199281. Lock: 11------------0-0----1---
212563. Lock: 1-1---01------------0----
225845. Lock: 1--11-------------0----0-
239127. Lock: 1---01-------1-----1-----
252409. Lock: 1----10-------1-1--------
265691. Lock: 1-----1-10--1------------
278973. Lock: 1------1-----1-10--------
292255. Lock: 1--------1-11----------1-
305537. Lock: 1----------0-----00---0--
318819. Lock: 1-----------------00-01--
332101. Lock: -10---0--1------0--------
345383. Lock: -10------1---0---------0-
358665. Lock: -11------------0----0---0
371947. Lock: -1-0------01-----------1-
385229. Lock: -1--0-----10-1-----------
398511. Lock: -1---0-----1-----10------
411793. Lock: -1----1---------0-----01-
425075. Lock: -1------1-01------0------
438357. Lock: -1--------11-------0---0-
451639. Lock: -1-----------01------1-1-
464921. Lock: --11-0-------------1---1-
478203. Lock: --11----1-------------01-
491485. Lock: --11----------1--1------0
504767. Lock: --1-1-----1-11-----------
518049. Lock: --1--0------00-------0---
531331. Lock: --1---0-----------00--1--
544613. Lock: --1-----0-0-------10-----
557895. Lock: --1-------0-10--0--------
571177. Lock: --1----------1-0--00-----
584459. Lock: ---10--0-1----1----------
597741. Lock: ---10-----1--------0-1---
611023. Lock: ---1-01----1----0--------
624305. Lock: ---1--10--------0----1---
637587. Lock: ---1---1---0--------10---
650869. Lock: ---1-----11------0----0--
664151. Lock: ---1-------0--1---11-----
677433. Lock: ---1------------1--01-1--
690715. Lock: ----11----10------1------
703997. Lock: ----11------------1--10--
717279. Lock: ----1--11------1---1-----
730561. Lock: ----1---0------00--0-----
743843. Lock: ----1-----0------00-----1
757125. Lock: ----1---------0-0--10----
770407. Lock: -----10---1--0-------0---
783689. Lock: -----1-10-0----0---------
796971. Lock: -----1--0--0-------11----
810253. Lock: -----1----0--10-----1----
823535. Lock: -----1-------0---1-10----
836817. Lock: ------11---00-1----------
850099. Lock: ------1-10-------10------
863381. Lock: ------1---11---00--------
876663. Lock: ------1-----0--------11-1
889945. Lock: -------11--0---1---1-----
903227. Lock: -------1-1-1----------01-
916509. Lock: -------1---1------11---0-
929791. Lock: -------1----------1--11-1
943073. Lock: --------1-00--------00---
956355. Lock: --------1----00-------01-
969637. Lock: ---------10---0-------10-
982919. Lock: ---------1---0-10---0----
996201. Lock: ----------11-----0-00----
1009483. Lock: ----------1------00--1-1-
1022765. Lock: -----------1----01---0--1
1036047. Lock: -------------10--1---1--0
1049329. Lock: 1-1-0-0-----------1------
1062611. Lock: 1-1----0--1---0----------
1075893. Lock: 1-1---------1------0--0--
1089175. Lock: 1--0---1---1----1--------
1102457. Lock: 1--1---------0-1------1--
1115739. Lock: 1---1---1--------0------1
1129021. Lock: 1----1-1---1---------0---
1142303. Lock: 1----1-------1--1---1----
1155585. Lock: 1-----0-----0--0--------0
1168867. Lock: 1------0-----0----0--1---
1182149. Lock: 1--------0-0----1-----0--
1195431. Lock: 1----------0--1------0-0-
1208713. Lock: -1-1--0-0-----------0----
1221995. Lock: -1-0------0----1--1------
1235277. Lock: -1--0--1----0--1---------
1248559. Lock: -1--0--------1---1-0-----
1261841. Lock: -1---1----0----0-------1-
1275123. Lock: -1----1--0----------1---0
1288405. Lock: -1-----1--1------0---0---
1301687. Lock: -1------0----1--1---1----
1314969. Lock: -1--------1---0-1-----1--
1328251. Lock: --1-1-1-0------1---------
1341533. Lock: --1-1-----1---1------0---
1354815. Lock: --1--1--1-----1-----1----
1368097. Lock: --1---0-0-0------------0-
1381379. Lock: --1---0---------0--0--1--
1394661. Lock: --1-----0-0---1--0-------
1407943. Lock: --1------0----1---0---0--
1421225. Lock: --1---------1---0----0-0-
1434507. Lock: ---1-0---0---------1---1-
1447789. Lock: ---1--0--1---0--------1--
1461071. Lock: ---1---0--1-1----0-------
1474353. Lock: ---1----1---0--0---1-----
1487635. Lock: ---1------1-0-------1---0
1500917. Lock: ---1----------1----1-0-0-
1514199. Lock: ----1-0-------1-1------1-
1527481. Lock: ----1--1--------0--1---1-
1540763. Lock: ----1----1---1-0--1------
1554045. Lock: ----1-------1-1--1-----0-
1567327. Lock: -----1-1----1---1-1------
1580609. Lock: -----1--0--------0-0---0-
1593891. Lock: -----1-----0-0--0-----1--
1607173. Lock: ------1-0--0----1-----0--
1620455. Lock: ------1---0-1-0-------0--
1633737. Lock: ------1------1-----0-0-0-
1647019. Lock: -------1--0----0---1--1--
1660301. Lock: --------1-0-1-------0---1
1673583. Lock: --------1-----0----1-1-0-
1686865. Lock: ----------1-1--0-----0-1-
1700147. Lock: ----------------1-1-0-1-0
Inserting codes...
M = code modified; A = code added
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMAAAAAAAA
AAAAMMMMMMMMMMMMMMMMMMMMMMMMAAAAMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMAAMMAAMMMMMMMMAAMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMAAMMAAMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMAAMMMMMMMMMM
MMMMMMAAAAMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMAAMMAAMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMAAMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMAAMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMAAMMMMMMMMMMAAMAAAAMMMMMMMAAMMAAMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMAAMMMMMMMM
AAMMMMMMMMMMMMMMMMMMMMMMMMMMMMAAMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMAAAAMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMAAMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMAAAAMMMMAAMMMMMMMMMMMMMMMMMMMMMMMMAA
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMAAMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMAAMMMMAAMMMMMMAAMMMMMMMMMAAMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMAAMMMMMMAAMMMMMMMMMMMMAA
```

```
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMAAMMMMMMMMAA
MMMMMMAAMMAAMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMAAMMMMMMMMMMMMMMMMMM
MMMMMMMMAAMMAAMMMMMMMMMMMMMMMMMMMMMMMMMM
MMAAMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMAAAAMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMAAMMMMMMMMMMMMMMMMMMMM
MMMMAAMMMMMMMMAAMMMMMMMMMMMMMMMMMMMMMMMM
AAMMMMMMMMMMMMMMAAMMMMMMMMMMMMMMMMMMMMMM
AAMMMMMMAAMMMMMMMMMMMMMMMMMMMMAAMMMMMMMM
MMMMMMMMMMMMMMAAAAMMMMMMMMAAMMMMMMMMMMMM
MMMMMMAAMMMMMMMMMMMMMMMMAAMMMMMMMMAAMMMM
MMMMMMMMMMMMMMAAMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMAAMMMMMMMMMMMMMMMMMMMMMMMM
MMAAAAMMMMMMMMMMMMMMMMAAAAMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMAAMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMAAMM
MMMMMMMMMMMMMMMMAAMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMAAMMMMAAMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMAAMMMMMMMM
MMMMAAMMMMMMMMMMMMMMMMAAMMMMMMAAMMMMMMMM
MMMMMMMMMMMMMMMMMMAAMMAAMMMMMMMMMMAAMMMM
MMMMMMMMAAMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMAAMMMMAAMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMAA
Converting into bitstrings...
Optimizing U-Code.. (len=212)
* Checking locks...
* Removing redundant codes...
* Removing 0000000000000000000000000
* Removing 0001000100010001000100010
* Removing 0011110000111100001111000
* Removing 0111000011110000111100001
* Removing 0111100001111000011110000
* Removing 1000011110000111100001111
* Removing 1100001111000011110000111
* Removing 1100110011001100110011001
* Removing 1110011001100110011001100
Length changed from 212 to 203!
Action completed after 00:01:25.6577842

U-Code (n=25, m=5, len=203):
  0000000000000000000010000
  0000000000000101010000001
  0000000000100000010000101
  0000000110100110110110011
  0000001001001101100011011
  0000001111011111001100011
  0000001111111100111111000
  0000010100111010111100010
  0000011001000010111011110
  0000011010101111110100100
  0000100010001000100010001
  0000100110100001000101000
  0000110001110011100011111
  0000110111000110001110010
  0000111100001000110001111
  0000111101110111011101110
  0001000011010100101010010
  0001001100100001000111001
  0001001100110100010010110
  0001001111101110000001101
  0001010111111100000000100
  0001011000001000001110111
  0001011001100110011001100
  0001011100010111000001101
  0001100110010110010010111
  0001100110011001100110011
  0001101100111010011001011
  0001110011110010000110001
  0001110110110101010010100
  0001111111001000001100100
  0010000011101111011011110
  0010000110010011010000101
  0010001000100010001000100
  0010001001000010000101111
  0010010101010101010101010
  0010010101011100010010101
  0010011001110110001011000
  0010011110000111111001000
  0010100100010011101011000
  0010100101010101110010110
  0010100111011000100101111
  0010101010101010101010101
  0010111111110001001101001
  0011000000100101100010000
  0011000011001011110001000
  0011001000010101100101011
  0011001100110011001100110
  0011010111000010011111011
  0011011001010101001111111
  0011011011100110010000010
  0011011100101110111100101
  0011100000111101111100010
  0011100010001011011101101
  0011110000101100110100110
  0011110101000111001100001
  0011111011111000110011000
  0011111111000011110000111
  0100000001001101100001000
  0100000001011101000000101
  0100010001000100010001000
  0100010011111011001110110
  0100010111110011001011111
  0100011000010001100101100
  0100011010010100111000101
  0100101101001011010010110
  0100110010001110100100100
  0100110010110100101101001
  0100111110001001011111111
  0101000000101110010101010
  0101001011010010110100101
  0101010100101001001010101
  0101100001000110111101011
  0101100010010111011101110
  0101101001011010010110100
  0101101101011111101000000
  0101101111100100000110110
  0101110110101101101011011
  0101110110111101111010000
  0101111001111011110001001
  0110000001101101000100101
  0110000011101000111110010
  0110000111100001111000011
  0110001101110111110110101
  0110001110111110100110011
  0110011000011010000110011
  0110100010111100001011101
  0110100101101001011010010
  0110101001000010011010101
  0110101110110100011101100
  0110110100001101001011101
  0110110100011110010000001
  0110111010001101010111001
  0110111010011110100111101
  0110111010100001001000010
  0111000100000010111111000
  0111000101100000100101110
  0111001011011100001000011
  0111001110010001110110001
  0111010010011001000111110
  0111010100100010101001010
  0111011110001111100011111
  0111011111011000010110000
  0111100100011110000010100
  0111110011110111111110111
  1000000000001100001010000
  1000001100001000000001000
  1000010000010001011110010
  1000011011100001111101011
  1000100000100111101001111
  1000100001110000011100000
  1000101011011101010110101
  1000101101100110111000001
  1000110001101110001001110
  1000110100100011110111100
  1000110100100100011101011
  1000111010011111011010001
  1000111011111101000000111
  1000111100001111000011110
  1001000101011110110111101
  1001000101100001011000010
  1001000101110010101000110
  1001001011100001101111110
  1001001011110010110100010
  1001010001001011100010011
  1001010110111101100101010
  1001011010010110100101101
  1001011101000011110100010
  1001100111100101111001100
  1001110001000001011001100
  1001110010001000001001010
  1001111000011110000111100
  1001111110010010111011010
  1010000110000100001110110
  1010001001010010010100100
  1010010000011011111001001
  1010010010100000010111111
  1010010110100101101001011
  1010011101101000100010001
  1010011110111001000010100
  1010101011010110110101010
  1010110100101101001011010
  1010111111010001101010101
  1011000001110110100000000
  1011001101001011010010110
  1011001101110001011011011
  1011010010110100101101001
  1011100101101011000111010
  1011100111101110011010011
  1011101000011001101000000
  1011101100000100110001001
  1011101110111011101110111
  1011111101000101111110100
  1100000000111100001111000
  1100000100000111001100111
  1100000101011010011101101
  1100001010111000110011110
  1100001111010011001011001
  1100011101110100100010010
  1100011111000010000011101
  1100100011010001000011010
  1100100100011001101111101
  1100100110101010110000000
  1100101000111101100000100
  1100110111101010011010100
  1100111100110100001110111
  1101000000001110110010110
  1101010101010101010101010
  1101011000100111011010000
  1101011010101010001101001
  1101011011101100010100111
  1101100001111000000110111
  1101100110001001110100111
  1101101010100011101101010
  1101101010101010101010101
  1101110111011101110111011
  1101111001101100101111010
  1101111100010000100100001
  1110000000110111110011011
  1110001001001010101101011
  1110001100001101111001110
  1110010011000101100110100
  1110011001101001101101000
  1110100110011001100110011
  1110100111110111110001000
  1110101000000011111111011
  1110101001011001001001100
  1110110011001011101101001
  1110110011011110111000110
```

```
11101110111011010111011101    11110011100011000011100000    11111010110001010000011101
11101111001010110101011101    11110110010111101111010111    11111100000000011000000111
11110000100010001000100010001 11110111011101110111011110    11111100001000000110011100
11110000111101110011110000    11111001010100000001011011    11111101101100100111001000
11110010001110011110001101    11111001101111010001000001    11111111111111111111111111
```

Da zu dem Fall keine Vergleichsmöglichkeiten existieren, ist es schwierig die Lösung zu beurteilen. Der Algorithmus arbeitet zumindest recht schnell, und die Optimierungsfunktion konnte einige Codes wegoptimieren. Da die Komplexität exponential zunimmt, ist die Lösung mit 203 Codes vermutlich auch nicht all zu weit von der optimalen Lösung entfernt.

6. Auswertung

Während es aufgrund fehlender Referenzen insgesamt schwierig ist ein eindeutiges Urteil zu der Effektivität des entwickelten Algorithmus abzugeben, so stellt die Lösung – in Anbetracht der Relation „Lösungsgröße : Berechnungsdauer “ – meiner Meinung nach einen passenden Kompromiss dar. Es wäre möglich, die Anzahl der benötigten Codes weiterhin zu vereinfachen, doch würde dies gleichzeitig einen deutlich größeren Zeitaufwand mit sich ziehen.

Da die Codes vermutlich später auch zum Öffnen eines Schlosses genutzt werden sollen, muss bedacht werden, dass die Zeit zur Berechnung eines einzelnen Codes innerhalb dieses U-Codes nicht länger als die zur Eingabe benötigte Zeit sein sollte. Dauert das Generieren eines um einen Code kürzeren U-Codes nämlich länger als diese Eingabe, so ist die Lösung – wenn sie denn in einem größeren Maßstab gesehen wird – schlechter als die Alternative, obwohl der generierte U-Code eigentlich kürzer wäre.

Daher ist es in diesem Fall auch nicht zwingend sinnvoll, einen öptimalenÜ-Code zu bilden. Im Rahmen dieses Kontextes erreicht der Algorithmus passable Ergebnisse; auch wenn diese vermutlich teilweise noch um ein paar Codes gekürzt werden könnten.

7. Quellcode

Program.cs

```
1  using System;
2  using System.Windows.Forms;
3
4  namespace UCode
5  {
6    internal static class Program
7    {
8      /// <summary>
9      /// Log Message Event
10     /// </summary>
11     public static event EventHandler OnLogMessage;
12
13     /// <summary>
14     /// Loggt eine Nachricht
15     /// </summary>
16     /// <param name="msg"></param>
17     public static void Log(string msg)
```

```
18      {
19        if (OnLogMessage != null)
20          OnLogMessage(msg, EventArgs.Empty);
21        Console.Write(msg);
22      }
23
24      /// <summary>
25      /// Loggt eine Nachricht (+Zeilenumbruch)
26      /// </summary>
27      /// <param name="msg"></param>
28      public static void LogLine(string msg)
29      {
30        Log(msg + Environment.NewLine);
31      }
32
33      /// <summary>
34      /// Der Haupteinstiegspunkt fuer die Anwendung.
35      /// </summary>
36      [STAThread]
37      private static void Main()
38      {
39        Application.EnableVisualStyles();
40        Application.SetCompatibleTextRenderingDefault(false);
41        Application.Run(new MainForm());
42      }
43    }
44  }
```

MainForm.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Diagnostics;
5  using System.Threading;
6  using System.Windows.Forms;
7
8  namespace UCode
9  {
10   internal delegate UniversalCode GeneratorDelegate();
11
12   internal partial class MainForm : Form
13   {
14     private int _activeSwitches = 5;
15     private Thread _currentThread;
16     private int _totalSwitches = 25;
17
18     public MainForm()
19     {
20       InitializeComponent();
21
22       //Ausgabe umleiten
23       Program.OnLogMessage += (str, e) => LogMessage((string) str);
24     }
25
26     /// <summary>
27     /// Zeigt eine Fehlermeldung an
28     /// </summary>
29     /// <param name="err"></param>
30     private static void ShowError(string err)
31     {
32       MessageBox.Show(err, @"Fehler", MessageBoxButtons.OK, MessageBoxIcon.Error
             );
33     }
34
35     /// <summary>
36     /// Fuehrt die angegebenen Methoden in einem zweiten (Hintergrunds-)Thread
           aus.
```

```
37      /// </summary>
38      /// <param name="doWork"></param>
39      private void RunThreaded(ThreadStart doWork)
40      {
41        var t = new Thread(() =>
42                           {
43                             try
44                             {
45                               var watch = new Stopwatch();
46                               watch.Start();
47                               doWork();
48                               LogMessage("Action␣completed␣after␣" + watch.
                                     Elapsed);
49                             }
50                             catch (ThreadAbortException) //Abbruch
51                             {
52                               LogMessage("Action␣aborted␣by␣user.");
53                             }
54                             catch (InvalidOperationException) //Abbruch @
                                   Sortierung
55                             {
56                               LogMessage("Action␣aborted␣by␣user.");
57                             }
58                             _currentThread = null;
59                           }) {IsBackground = true};
60        t.Start();
61        _currentThread = t;
62      }
63
64      /// <summary>
65      /// Schreibt eine Meldung in das Log
66      /// </summary>
67      /// <param name="msg"></param>
68      private void LogMessage(string msg)
69      {
70        if (InvokeRequired)
71        {
72          BeginInvoke(new EventHandler((obj, e) => LogMessage(msg)));
73          return;
74        }
75
76        outputBox.AppendText(msg);
77        outputBox.Select(outputBox.Text.Length, 0);
78        outputBox.ScrollToCaret();
79      }
80
81      /// <summary>
82      /// Aktiviert/Deaktiviert die OperationsButtons
83      /// Deaktiviert/Aktiviert den CancelButton
84      /// </summary>
85      /// <param name="enabled"></param>
86      private void ToggleButtons(bool enabled)
87      {
88        if (InvokeRequired)
89        {
90          Invoke(new EventHandler((obj, e) => ToggleButtons(enabled)));
91          return;
92        }
93
94        if (!enabled)
95          outputBox.Text = "";
96
97        generateButton.Enabled = enabled;
98        bruteforceButton.Enabled = enabled;
99        validateButton.Enabled = enabled ? (resultBox.Text.Length > 0) : false;
100       cancelButton.Enabled = !enabled;
101     }
102
103     /// <summary>
```

```
104     /// ueberprueft den U-Code
105     /// </summary>
106     /// <param name="code"></param>
107     private void ValidateCode(string code)
108     {
109       //1. In U-Code umwandeln
110       var ucode = new List<string>();
111
112       foreach (string str in code.Split('\r', '\n'))
113       {
114         string trim = str.Trim();
115
116         if (string.IsNullOrEmpty(trim))
117           continue; //leer
118
119         //Pruefen, ob der String ungueltige Zeichen enthaelt
120         bool valid = true;
121
122         foreach (char c in trim)
123         {
124           if (c != '1' && c != '0' && c != '-')
125           {
126             valid = false;
127             break;
128           }
129         }
130
131         if (!valid)
132           continue;
133
134         ucode.Add(trim);
135       }
136
137       //U-Code pruefen.
138       CodeLock.TestUCode(_totalSwitches, _activeSwitches, ucode.ToArray());
139       ToggleButtons(true);
140     }
141
142     private void GenerateUCode(GeneratorDelegate method)
143     {
144       resultBox.Text = "";
145       ToggleButtons(false);
146       RunThreaded(() => OnResultArrived(method()));
147     }
148
149     private void OnResultArrived(UniversalCode result)
150     {
151       if (InvokeRequired)
152       {
153         Invoke(new EventHandler((obj, e) => OnResultArrived(result)));
154         return;
155       }
156
157       resultBox.Text = result.ToString();
158       ToggleButtons(true);
159     }
160
161     #region Parameter (Gesamt + Aktive Bits)
162
163     private void BitSliderScroll(object sender, EventArgs e)
164     {
165       bitBox.Text = bitSlider.Value.ToString();
166       BitBoxValidated(sender, e);
167     }
168
169     private void ActiveSliderScroll(object sender, EventArgs e)
170     {
171       activeBox.Text = activeSlider.Value.ToString();
172       ActiveBoxValidated(sender, e);
```

```
173     }
174
175     private void BitBoxValidating(object sender, CancelEventArgs e)
176     {
177       int val;
178
179       //Keine oder ungueltige Zahl?
180       if (!int.TryParse(bitBox.Text, out val) || val < 1)
181       {
182         ShowError("Bitte einen gueltigen Integer groeßer als 0 eingeben");
183         e.Cancel = true;
184         return;
185       }
186     }
187
188     private void BitBoxValidated(object sender, EventArgs e)
189     {
190       _totalSwitches = int.Parse(bitBox.Text);
191       bitSlider.Value = _totalSwitches;
192
193       activeSlider.Maximum = _totalSwitches;
194
195       if (_activeSwitches > _totalSwitches)
196         activeBox.Text = (_activeSwitches = _totalSwitches).ToString();
197     }
198
199     private void ActiveBoxValidating(object sender, CancelEventArgs e)
200     {
201       int val;
202
203       //Keine oder ungueltige Zahl?
204       if (!int.TryParse(activeBox.Text, out val) || val < 1 || val >
              _totalSwitches)
205       {
206         ShowError("Bitte einen gueltigen Integer groeßer als 0 und kleiner
              gleich der Maximalen Schalteranzahl eingeben");
207         e.Cancel = true;
208         return;
209       }
210     }
211
212     private void ActiveBoxValidated(object sender, EventArgs e)
213     {
214       _activeSwitches = int.Parse(activeBox.Text);
215       activeSlider.Value = _activeSwitches;
216     }
217
218     #endregion
219
220     #region Buttons
221
222     private void GenerateButtonClick(object sender, EventArgs e)
223     {
224       GenerateUCode(() => Generator.GenerateUCode(_totalSwitches,
              _activeSwitches));
225     }
226
227     private void BruteforceButtonClick(object sender, EventArgs e)
228     {
229       GenerateUCode(() => new Bruteforce(_totalSwitches, _activeSwitches).
              GenerateUCode());
230     }
231
232     private void ValidateButtonClick(object sender, EventArgs e)
233     {
234       //Im Hintergrund verarbeiten
235       string text = resultBox.Text;
236       ToggleButtons(false);
237
```

```
238         RunThreaded(() => ValidateCode(text));
239       }
240
241       private void CancelButtonClick(object sender, EventArgs e)
242       {
243         if (_currentThread == null)
244           return;
245
246         cancelButton.Enabled = false;
247         _currentThread.Abort();
248         ToggleButtons(true);
249       }
250
251       private void ResultBoxTextChanged(object sender, EventArgs e)
252       {
253         validateButton.Enabled = (resultBox.Text.Length > 0);
254       }
255
256       #endregion
257     }
258 }
```

Util.cs

```
1  using System;
2
3  namespace UCode
4  {
5    /// <summary>
6    /// Helfer Klasse
7    /// Enthaelt nuetzliche Methoden
8    /// </summary>
9    internal static class Util
10   {
11     /// <summary>
12     /// Fuegt ein Element an das Array an
13     /// </summary>
14     /// <typeparam name="T"></typeparam>
15     /// <param name="arr"></param>
16     /// <param name="app"></param>
17     /// <returns></returns>
18     public static T[] Append<T>(T[] arr, T app)
19     {
20       var res = new T[arr.Length + 1];
21       Array.Copy(arr, res, arr.Length);
22       res[arr.Length] = app;
23       return res;
24     }
25
26     /// <summary>
27     /// Vergleicht zwei Elemente mit dem
28     /// angegebenen Kriterium
29     /// Gibt Null zurueck, wenn beide Objekte identisch sind
30     /// </summary>
31     /// <typeparam name="T"></typeparam>
32     /// <param name="a"></param>
33     /// <param name="b"></param>
34     /// <param name="conv"></param>
35     /// <returns></returns>
36     public static int? Compare<T>(T a, T b, Converter<T, int> conv)
37     {
38       int res = conv(a).CompareTo(conv(b));
39       return res == 0 ? (int?) null : res;
40     }
41
42     /// <summary>
43     /// Vergleicht zwei Elemente mit dem
44     /// angegebenen Vergleich
```

```
45      /// Gibt Null zurueck, wenn beide Objekte identisch sind
46      /// </summary>
47      /// <typeparam name="T"></typeparam>
48      /// <param name="a"></param>
49      /// <param name="b"></param>
50      /// <param name="comp"></param>
51      /// <returns></returns>
52      public static int? Compare<T>(T a, T b, Comparison<T> comp)
53      {
54        int res = comp(a, b);
55        return res == 0 ? (int?) null : res;
56      }
57
58      /// <summary>
59      /// Gibt den Code als Bitstring wieder
60      /// </summary>
61      /// <param name="code"></param>
62      /// <returns>String aus 0/1/-</returns>
63      public static string GetBitstring(bool?[] code)
64      {
65        string str = "";
66
67        foreach (var e in code)
68          str += (e == null ? "-" : (e.Value ? "1" : "0"));
69
70        return str;
71      }
72    }
73  }
```

CodeLock.cs

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace UCode
5  {
6    /// <summary>
7    /// Schloss
8    /// Stellt ein wie in der Dokumentation beschriebenes Schloss zur Verfuegung
       .
9    /// Enthaelt N Kippschalter, von denen nur M aktiv sind.
10   /// </summary>
11   internal class CodeLock
12   {
13     //---------- Felder & Eigenschaften ----------
14
15     #region Delegates
16
17     /// <summary>
18     /// Delegate fuer die "GetAllLocks"-Methode
19     /// </summary>
20     /// <param name="s">Erzeugtes Schloss</param>
21     public delegate void LockCreated(CodeLock s);
22
23     #endregion
24
25     /// <summary>
26     /// Enthaelt die Kombination des Schlosses
27     /// null entspricht unverkabeltem Schalter
28     /// </summary>
29     private readonly bool?[] _combination;
30
31     /// <summary>
32     /// Konstruktor
33     /// </summary>
34     /// <param name="komb"></param>
35     private CodeLock(bool?[] komb)
```

```
36      {
37        _combination = komb;
38      }
39
40      //---------- Instanzierte Methoden ----------
41
42      /// <summary>
43      /// Versucht das Schloss zu oeffnen
44      /// </summary>
45      /// <param name="code">Kombination aus 1'en oder 0'en</param>
46      /// <returns>True wenn die Kombination das Schloss geoeffnet hat</returns>
47      public bool Unlock(string code)
48      {
49        if (code.Length != _combination.Length)
50          return false; //Code zu kurz/lang
51
52        for (int i = 0; i < _combination.Length; i++)
53        {
54          bool? template = _combination[i];
55          bool? example = null;
56
57          if (code[i] == '0') example = false;
58          else if (code[i] == '1') example = true;
59
60          //Schalter ist inaktiv -> ignorieren
61          if (template == null) continue;
62
63          //Richtige Position
64          if (template == example) continue;
65
66          //Falsche Position
67          return false;
68        }
69
70        //Nicht ausgeschieden -> Richtige Loesung
71        return true;
72      }
73
74      /// <summary>
75      /// Versucht das Schloss mit den gegebenen Codes zu oeffnen
76      /// </summary>
77      /// <param name="code">Sammlung von Codes</param>
78      /// <param name="usedCode">Gibt den zum oeffnen verwendeten Code wieder</
            param>
79      /// <returns></returns>
80      public bool Unlock(IEnumerable<string> code, out string usedCode)
81      {
82        foreach (string c in code)
83        {
84          if (Unlock(c))
85          {
86            usedCode = c;
87            return true;
88          }
89        }
90        usedCode = null;
91        return false;
92      }
93
94      /// <summary>
95      /// Versucht das Schloss mit den gegebenen Codes zu oeffnen
96      /// </summary>
97      /// <param name="ucode">U-Code</param>
98      /// <param name="usedCode">Gibt den zum oeffnen verwendeten Code wieder</
            param>
99      /// <returns></returns>
100     public bool Unlock(UniversalCode ucode, out string usedCode)
101     {
102       return Unlock(ucode.Codes, out usedCode);
```

```
103     }
104
105     /// <summary>
106     /// Gibt die Loesungskombination fuer dieses Schloss
107     /// (Backdoor! ^^)
108     /// </summary>
109     /// <returns></returns>
110     public bool?[] GetCombination()
111     {
112       return _combination;
113     }
114
115     public override string ToString()
116     {
117       return Util.GetBitstring(_combination);
118     }
119
120     //---------- Statische Methoden ----------
121
122     /// <summary>
123     /// Erstellt ein zufaelliges neues Schloss
124     /// </summary>
125     /// <param name="schalter">Anzahl der Schalter</param>
126     /// <param name="aktiv">Anzahl der aktiven Schalter</param>
127     public static CodeLock GetRandomLock(int schalter, int aktiv)
128     {
129       //--> Fehlerbehandlung
130       if (aktiv > schalter) throw new ArgumentOutOfRangeException("aktiv", @"
              Aktive Schalter > Anzahl Schalter");
131
132
133       //--> Kombination erzeugen
134       var rand = new Random();
135       var elem = new List<bool?>();
136
137       //1. Schalter generieren
138       for (int i = 1; i <= schalter; i++)
139       {
140         if (i > aktiv) elem.Add(null); //Inaktiv
141         else elem.Add(rand.Next(0, 2) == 0 ? false : true);
142       }
143
144       //2. Schalter verteilen
145       var komb = new bool?[schalter];
146       for (int i = 0; i < schalter; i++)
147       {
148         //Zufaelligen Schalter suchen
149         int j = rand.Next(0, elem.Count);
150         komb[i] = elem[j];
151         elem.RemoveAt(j);
152       }
153
154       return new CodeLock(komb);
155     }
156
157     /// <summary>
158     /// Erzeugt rekursiv alle moeglichen Schloesser aus den gegebenen
          Parametern
159     /// Ruft fuer jedes erzeugte Schloss die uebergebene Methode auf
160     /// </summary>
161     /// <param name="switches">Anzahl der (verbleibenden) Schalter</param>
162     /// <param name="active">Anzahl der (verbleibenden) aktiven Schalter</
          param>
163     /// <param name="call">Methode, welche aufgerufen werden soll</param>
164     /// <param name="komb">ggf. Vorgegebene Prefixkombination</param>
165     public static void GetAllLocks(int switches, int active, LockCreated call
          , params bool?[] komb)
166     {
167       if (active > switches || active < 0)
```

```
168         return; //Unmoegliche Kombination verwerfen
169
170
171       //Rekursiv neue Kombinationen erzeugen (d.h. Schalter hinzufuegen)
172       if (switches > 0)
173       {
174         //3 Moegliche Kombinationen: Inaktiver Schalter, Aktiver (true) und
              Aktiver (false)
175         GetAllLocks(switches - 1, active, call, Util.Append(komb, null));
176         if (active > 0)
177         {
178           GetAllLocks(switches - 1, active - 1, call, Util.Append(komb, true));
179           GetAllLocks(switches - 1, active - 1, call, Util.Append(komb, false));
180         }
181       }
182       else
183       {
184         //Fertige Kombination (alle Schalter belegt)
185         //Switches == 0, Active <= Switches && Active >= 0 -> Active=0
186         call(new CodeLock(komb));
187       }
188     }
189
190     /// <summary>
191     /// Testet einen UCode auf Vollstaendigkeit
192     /// </summary>
193     /// <param name="ucode">UCode</param>
194     /// <param name="active">Aktive Schalter</param>
195     /// <param name="switches">Gesamte Anzahl der Schalter</param>
196     /// <returns>True falls der Test erfolgreich verlaufen ist</returns>
197     public static bool TestUCode(int switches, int active, string[] ucode)
198     {
199       try
200       {
201         Program.LogLine("Testing U-Code (N=" + switches + ", M=" + active + ",
              cnt=" + ucode.Length + ") ..");
202         Program.LogLine("-----------------------------------");
203
204         int kombinationen = 0;
205         GetAllLocks(switches, active, schloss =>
206                                       {
207                                         kombinationen++;
208
209                                         string code;
210                                         bool result = schloss.Unlock(ucode,
                                                out code);
211
212                                         if (switches < 10) //Zuviel Spam
213                                           Program.LogLine(schloss + ": " + (
                                                code ?? ""));
214
215                                         if (!result)
216                                         {
217                                           throw new ArgumentException("U-Code
                                                incomplete -> " + schloss);
218                                         }
219                                       });
220         Program.LogLine("Test successful! " + kombinationen + " combinations
              checked.");
221         Program.LogLine("");
222       }
223       catch (ArgumentException e)
224       {
225         Program.LogLine("Test failed: " + e.Message);
226         Program.LogLine("");
227         return false;
228       }
229       return true;
230     }
```

```
231
232     /// <summary>
233     /// Testet einen UCode auf Vollstaendigkeit
234     /// </summary>
235     /// <param name="ucode">UCode</param>
236     /// <returns>True, falls der Test erfolgreich verlaufen ist</returns>
237     public static bool TestUCode(UniversalCode ucode)
238     {
239       return TestUCode(ucode.Switches, ucode.ActiveSwitches, ucode.Codes);
240     }
241   }
242 }
```

Generator.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Threading;
4
5  namespace UCode
6  {
7    /// <summary>
8    /// U-Code Generator
9    /// Versucht einen moeglichst kurzen U-Code zu generieren
10   /// </summary>
11   internal static class Generator
12   {
13     /// <summary>
14     /// Erzeugt einen U-Code
15     /// </summary>
16     /// <param name="switches">Gesamtanzahl der Schalter</param>
17     /// <param name="active">Anzahl der aktiven Schalter</param>
18     /// <returns>U-Code</returns>
19     public static UniversalCode GenerateUCode(int switches, int active)
20     {
21       Program.LogLine("Solving problem using Generator!");
22
23       //Alle moeglichen Schloesser speichern
24       var locks = new List<bool?[]>();
25       Program.LogLine("Retrieving possible locks..");
26       CodeLock.GetAllLocks(switches, active, schloss => locks.Add(schloss.
             GetCombination()));
27
28       //Elemente sortieren
29       SortLocks(locks);
30
31       //Einige Elemente der Sortierung zur Beobachtung ausgeben
32       for (int i = 0; i < locks.Count; i += i < 50 ? 1 : (Math.Max(locks.Count
             /128, 1)))
33         Program.LogLine((i + 1) + ". Lock: " + Util.GetBitstring(locks[i]));
34
35       //U-Code generieren
36       var ucode = new List<bool?[]>();
37
38       Program.LogLine("Inserting codes...");
39       Program.LogLine("M = code modified; A = code added");
40       foreach (var l in locks)
41         AddCode(l, ucode);
42       Program.LogLine("");
43
44       //Den UCode in die Bitstrings umwandeln
45       Program.LogLine("Converting into bitstrings...");
46       List<string> bitstrings = ConvertUCode(ucode);
47
48       //Optimieren
49       Program.LogLine("Optimizing U-Code.. (len=" + bitstrings.Count + ")");
50       Optimize(switches, active, bitstrings);
51       Program.LogLine("Length changed from " + ucode.Count + " to " + bitstrings
```

```
            .Count + "!");
52
53          //Ausgabe
54          return new UniversalCode(switches, active, bitstrings.ToArray());
55      }
56
57      /// <summary>
58      /// Sortiert die Schloesser
59      /// </summary>
60      /// <param name="locks"></param>
61      private static void SortLocks(List<bool?[]> locks)
62      {
63          #region Suchkriterien (Bitstring -> Vergleichbarer Int-Wert)
64
65          //Laengste Gruppierung im U-Code (111- wird bevorzugt gegenueber 11-1)
66          Converter<bool?[], int> getGroups = a =>
67                                              {
68                                                  int maxLength = 0;
69                                                  int groupLen = 0;
70
71                                                  foreach (var bit in a)
72                                                  {
73                                                      if (bit.HasValue)
74                                                      {
75                                                          groupLen++;
76
77                                                          if (groupLen > maxLength)
78                                                              maxLength = groupLen;
79                                                      }
80                                                      else groupLen = 0;
81                                                  }
82
83                                                  return maxLength;
84                                              };
85
86          //Vergleicht die Position der aktiven Bits (von links nach rechts)
87          Comparison<bool?[]> getPosition = (a, b) =>
88                                            {
89                                                //Zuerst gefundene aktive
90                                                //Bits betrachten
91                                                int nA = 0;
92                                                int nB = 0;
93
94                                                for (int i = 0; i < a.Length; i++)
95                                                {
96                                                    if (a[i].HasValue) nA++;
97                                                    if (b[i].HasValue) nB++;
98
99                                                    if (nA < nB) return 1;
100                                                   if (nA > nB) return -1;
101                                               }
102
103                                               return 0; //Gleichwertig
104                                           };
105
106         //Bevorzugt 1'en und erhoeht damit das Konfliktpotenzial
107         Converter<bool?[], int> getOnes = a =>
108                                           {
109                                               int ones = 0;
110
111                                               foreach (var b in a)
112                                                   if (b == true) ones++;
113
114                                               return ones;
115                                           };
116
117         //Bewertet den gegebenen Bitarray anhand der Werte der aktiven Bits (
                rechts -> links, aufsteigend)
118         Converter<bool?[], int> getValue = a =>
```

```
119                                         {
120                                           int res = 0;
121                                           int activeBits = 0;
122
123                                           for (int i = a.Length - 1; i >= 0;
                                                  i--)
124                                           {
125                                             bool? val = a[i];
126
127                                             if (val.HasValue)
128                                             {
129                                               if (val.Value)
130                                                 res += 1 << activeBits;
131                                               activeBits++;
132                                             }
133                                           }
134
135                                           return res;
136                                         };
137
138       #endregion
139
140       //Eintraege nach Wert sortieren um dann die inversen Elemente entfernen
141       //(vereinfacht die Suchmenge und erlaubt genauere Kontrolle des
              Konfliktverhaltens)
142       Program.LogLine("Sorting by value ...");
143       locks.Sort((a, b) => Util.Compare(a, b, getValue) ?? 0);
144
145       //Inverse Elemente entfernen
146       //(da wir nach Wert sortiert haben, enthaelt die zweite Haelfte
147       // automatisch die zu entfernenden inversen Elemente)
148       Program.LogLine("Removing inverse elements ...");
149       locks.RemoveRange(0, locks.Count/2);
150
151       //Suchkriterien zu einer eindeutigen Suchabfrage kombinieren
152       Comparison<bool?[]> search = (a, b) => -Util.Compare(a, b, getGroups) ??
              //Gruppenlaenge, absteigend
153                                             Util.Compare(a, b, getPosition) ??
                                                  //Position, aufsteigend
154                                             Util.Compare(a, b, getOnes) ?? //
                                                  Einsen, aufsteigend
155                                             Util.Compare(a, b, getValue) ?? //
                                                  Wert, aufsteigend
156                                             0; //irrelevant, Suche ist schon
                                                  eindeutig.
157       //Schloesser nach Suchabfrage sortieren
158       Program.LogLine("Sorting (main sort) ...");
159       locks.Sort(search); //nutzt Quicksort
160
161       //Element und inverses Element jeweils Paarweise hinzufuegen..
162       Program.LogLine("Adding inverse elements ...");
163       var result = new List<bool?[]>(locks.Count*2);
164
165       foreach (var l in locks)
166       {
167         result.Add(l); //Element
168         result.Add(Invert(l)); //Inverses Element
169       }
170
171       //Ergebnisse uebertragen
172       Program.LogLine("Creating final lock list ...");
173       locks.Clear();
174       locks.AddRange(result);
175     }
176
177     /// <summary>
178     /// Konvertiert einen U-Code aus Bitarrays in Bitstrings
179     /// </summary>
180     /// <param name="ucode"></param>
```

```
181     /// <returns></returns>
182     private static List<string> ConvertUCode(IEnumerable<bool?[]> ucode)
183     {
184       //U-Code in Bitstrings umwandeln
185       var result = new List<string>();
186 
187       //Unbenutzte Zeichen durch 0 ersetzen;
188       //dadurch kann spaeter besser optimiert werden
189       foreach (var code in ucode)
190         result.Add(Util.GetBitstring(code).Replace('-', '0'));
191 
192       //Ergebnisse sortieren und zurueckgeben
193       result.Sort();
194       return result;
195     }
196 
197     private static void Optimize(int switches, int active, ICollection<
          string> ucode)
198     {
199       //Enthaelt alle Schloesser, und die Kombination, welche diese oeffnen
200       var lockCodes = new List<List<string>>();
201       Program.LogLine("* Checking locks...");
202       CodeLock.GetAllLocks(switches, active, curLock =>
203                                              {
204                                                var openers = new List<string
                                                       >();
205 
206                                                foreach (string code in ucode)
207                                                  if (curLock.Unlock(code))
208                                                    openers.Add(code);
209 
210                                                lockCodes.Add(openers);
211                                              });
212 
213       //Solange Codes entfernen bis Schloesser jeweils von nur
214       //genau einem Schluessel geoeffnet werden koennen
215       Program.LogLine("* Removing redundant codes...");
216       bool actionTaken;
217       var removeable = new List<string>(ucode);
218 
219       do
220       {
221         actionTaken = false;
222 
223         //Pruefen, welche Codes notwendig sind
224         foreach (var l in lockCodes)
225         {
226           if (l.Count == 1)
227           {
228             string code = l[0];
229 
230             //Code ist notwendig -> Nicht entfernbar
231             if (removeable.Contains(code))
232               removeable.Remove(code);
233           }
234         }
235 
236         if (removeable.Count > 0)
237         {
238           string toRemove = removeable[0];
239           removeable.RemoveAt(0);
240           ucode.Remove(toRemove);
241           Program.LogLine("* Removing " + toRemove);
242 
243           //Code entfernen..
244           foreach (var codeList in lockCodes)
245           {
246             if (codeList.Contains(toRemove))
247               codeList.Remove(toRemove);
```

```
248             }
249
250             actionTaken = true;
251           }
252         } while (actionTaken);
253       }
254
255       /// <summary>
256       /// Invertiert den Code
257       /// </summary>
258       /// <param name="code"></param>
259       /// <returns></returns>
260       private static bool?[] Invert(IList<bool?> code)
261       {
262         var res = new bool?[code.Count];
263
264         for (int i = 0; i < res.Length; i++)
265           res[i] = (code[i] == null ? null : !code[i]);
266
267         return res;
268       }
269
270       /// <summary>
271       /// Adds the given lockcode to the U-Code
272       /// </summary>
273       /// <param name="lockCode">Source UCode</param>
274       /// <param name="ucode">Target Collection</param>
275       private static void AddCode(bool?[] lockCode, ICollection<bool?[]> ucode)
276       {
277         //Besten Match merken
278         Thread.Sleep(0); //Zeit zum Interrupten geben
279         bool?[] match = null;
280
281         foreach (var code in ucode)
282         {
283           CodeType res = CompareCode(lockCode, code);
284           if (res == CodeType.Working)
285             return; //done
286
287           //Falls wir keinen funktionierenden Code finden,
288           //Erstbesten nehmen
289           if (match == null && res == CodeType.Possible)
290             match = code;
291         }
292
293         //Kein passender Code gefunden; neuen generieren
294         if (match == null)
295         {
296           Program.Log("A"); //"Added"
297           ucode.Add(lockCode);
298         }
299           //Ergaenzen
300         else
301         {
302           Program.Log("M"); //"Modified"
303           for (int i = 0; i < lockCode.Length; i++)
304           {
305             bool? bit = lockCode[i];
306
307             if (bit == null)
308               continue; //nicht relevant
309
310             //Bit im Code setzen
311             match[i] = bit;
312           }
313         }
314       }
315
316       /// <summary>
```

```
317     /// Vergleicht eine Schlosskombination mit einem Code
318     /// </summary>
319     /// <param name="lockCombination">Schloss</param>
320     /// <param name="code">Code</param>
321     /// <returns>Code passt/Code koennte passen/Code passt nicht</returns>
322     private static CodeType CompareCode(bool?[] lockCombination, bool?[] code
            )
323     {
324       if (lockCombination.Length != code.Length)
325         return CodeType.Invalid;
326
327       int notMatched = 0;
328
329       for (int i = 0; i < code.Length; i++)
330       {
331         bool? t = lockCombination[i];
332
333         //Bit ist im Schloss nicht gesetzt
334         if (t == null) continue;
335
336         bool? e = code[i];
337
338         //Bit ist noch nicht gesetzt
339         if (e == null)
340         {
341           notMatched++;
342           continue;
343         }
344
345         //Bits stimmen nicht ueberein
346         if (e != t) return CodeType.Invalid;
347       }
348
349       return notMatched > 0 ? CodeType.Possible : CodeType.Working;
350     }
351
352     #region Nested type: CodeType
353
354     private enum CodeType
355     {
356       Working,
357       Possible,
358       Invalid,
359     }
360
361     #endregion
362   }
363 }
```

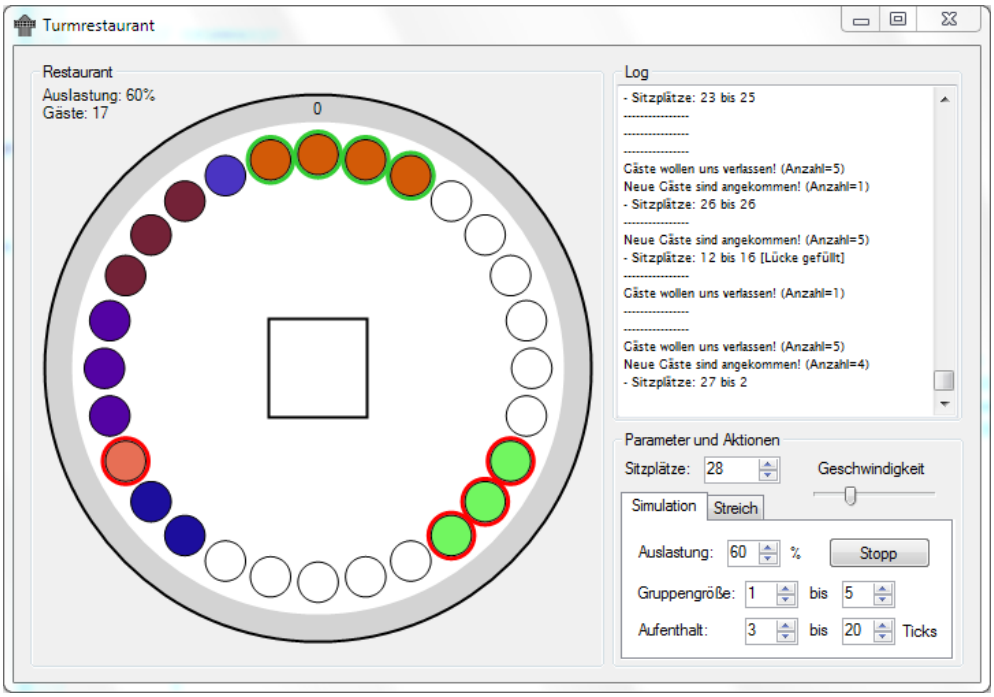
Bruteforce.cs

```
1  using System.Collections.Generic;
2
3  namespace UCode
4  {
5    /// <summary>
6    /// Bruteforce U-Code Generator
7    /// </summary>
8    internal class Bruteforce
9    {
10     private readonly int _active;
11     private readonly string[] _combinations;
12     private readonly CodeLock[] _locks;
13     private readonly int _switches;
14     private string[] _best;
15
16     /// <summary>
17     /// Bruteforce UCode Generator
```

```
18      /// </summary>
19      /// <param name="m">Switches</param>
20      /// <param name="n">Active Switches</param>
21      public Bruteforce(int m, int n)
22      {
23        Program.LogLine("Solving problem using brute force!");
24        _switches = m;
25        _active = n;
26
27        //1. Generiere alle moeglichen Kombinationen
28        Program.LogLine("Generating all possible combinations...");
29        var kombs = new List<string>();
30        GenerateElements(kombs, m, "");
31        _combinations = kombs.ToArray();
32
33        //2. Generiere alle moeglichen Schloesser
34        Program.LogLine("Generate all possible locks...");
35        var locks = new List<CodeLock>();
36        CodeLock.GetAllLocks(m, n, locks.Add);
37        _locks = locks.ToArray();
38      }
39
40
41      /// <summary>
42      /// Generate Code
43      /// </summary>
44      /// <returns>Best moeglicher UCode</returns>
45      public UniversalCode GenerateUCode()
46      {
47        Program.LogLine("Starting recursion!");
48        GenerateCombinations(new string[0], 0);
49        return new UniversalCode(_switches, _active, _best);
50      }
51
52      private void GenerateCombinations(string[] ucode, int ind)
53      {
54        if (_best != null && ucode.Length >= _best.Length)
55          return; //Loesung verwerfen; zu schlecht
56
57        //Funktioniert der U-Code?
58        if (IsWorking(ucode))
59        {
60          Program.LogLine("len=" + ucode.Length);
61          _best = ucode;
62          return;
63        }
64
65        if (ind >= _combinations.Length)
66          return; //no elements left
67
68        //Weitere Codes zum U-Code hinzupacken (rekursion)
69        for (int i = ind; i < _combinations.Length; i++)
70          GenerateCombinations(Util.Append(ucode, _combinations[i]), i + 1);
71      }
72
73      private static void GenerateElements(ICollection<string> list, int max,
            string cur)
74      {
75        if (cur.Length == max)
76          list.Add(cur);
77        else
78        {
79          GenerateElements(list, max, cur + "0");
80          GenerateElements(list, max, cur + "1");
81        }
82      }
83
84      private bool IsWorking(IEnumerable<string> ucode)
85      {
```

```
86        foreach (CodeLock l in _locks)
87        {
88          string str;
89          if (!l.Unlock(ucode, out str))
90            return false;
91        }
92        return true;
93      }
94    }
95  }
```

C. Aufgabe 2: Das Turmrestaurant



Das Turmrestaurant stellt einen besonderen Fall der Fragmentierungs- und Defragmentierungsproblematik einer Festplatte dar. Gäste kommen zu unterschiedlichen Zeiten in immer variierenden Mengen in das Restaurant, und bleiben dort auch für eine variable Zeit. Dadurch bilden sich während des Restaurantbetriebs Lücken und kleinere Blöcke. Während dies in einem Restaurant mit genügend Sitzplätzen kein Problem darstellt, gelangt man bei einer geringeren Anzahl an Sitzplätzen schnell an einem Punkt, an dem eine Gruppe keine kontinuierlich freie Sitzreihe findet. Bei Festplatten würde der Datenblock in solch einem Fall in mehrere kleine Stücke fragmentiert werden und daher in längeren Zugriffszeiten resultieren; in diesem Beispiel stellt solch eine Fragmentierung jedoch ein Abbruchskriterium des Programms dar.

Obwohl das Turmrestaurant sich also prinzipiell mit dem Fragmentierungsverhalten von Festplatten vergleichen lässt, sind die Schwerpunkte beider Problematiken sehr unterschiedlich: Bei Dateisystemen muss das Datenplatzierungsverfahren mit deutlich größeren Datenmengen und Transferraten arbeiten - Die Geschwindigkeit spielt dort teilweise also eine wichtigere Rolle als die Fragmentationsvermeidung selber. Bei dem Turmrestaurant wird man stattdessen nie mehr als 1000 Sitzplätze haben, weshalb die Performance des Algorithmus selber keine besondere Rolle spielt. Hier liegt der Fokus aufgrund des K.O.-Kriteriums komplett auf der Vermeidung einer Fragmentation.

Die Anzahl der Sitzplätze des Turmrestaurants ist dabei in dem Programm frei einstellbar. In der weiteren Dokumentation wird jedoch vorranging der Fall $n_{max} = 28$ betrachtet.

1. Lösungsidee

Aufgrund der vorgegebenen Aufgabenstellungen liegt es nah für dieses Problem eine Art Simulator zu entwickeln. Dabei wird für die erste Teilaufgabe zunächst eine Restaurant-Simulation verwirklicht, welche versucht anhand diverser einstellbarer Parameter einen realistischen Ablauf und Besucherfluss eines Restaurants darzustellen.

In der zweiten Teilaufgabe werden das gleiche System und die vorhandenen Klassen zu der Realisation des Streiches der Töchter adaptiert. Im Gegensatz zu der ersten Teilaufgabe übernimmt hier aber nicht der Restaurant-Simulator den Besucherfluss. Stattdessen werden für diese Teilaufgabe Gäste (oder hier: Schüler) gezielt über eine neu geschaffene Klasse explizit verwaltet.

1.1. Die Simulation

Die Simulation erfolgt anhand eines rundenbasiertes System: Pro „Tick“ wird jeweils die verbleibende Aufenthaltsdauer einer Gruppe um eins dekrementiert. Sobald die verbleibende Aufenthaltsdauer den Wert 0 erreicht, verlässt die Gruppe das Restaurant. Anschließend wird die aktuelle Auslastung $Usage = Guests/Seats$ berechnet und – falls diese unterhalb der vorher eingestellten Mindestauslastung liegt – wird eine[2] neue Gruppe hinzugefügt.

Ich habe die Auslastung als relevanten Parameter anerkannt, da die Chance das K.O.-Kriterium „Ober ärgert sich“ auszulösen direkt von der Auslastung abhängt: Bei niedriger Auslastung ist dies fast unmöglich; bei höherer Auslastung sehr wahrscheinlich. Weiterhin kann der Benutzer die möglichen Gruppengrößen und die Aufenthaltsdauer anpassen, da auch diese Parameter die Chance einer Fragmentierung signifikant beeinflussen.

Der Simulator selber wird dabei anschaulich durch eine grafische Benutzeroberfläche dargestellt und ermöglichst es die Simulation schrittweise (mit wählbarer Geschwindigkeit) zu beobachten. Dabei wird jeweils eine grafische Repräsentation des Restaurants zu dem aktuellen Zeitpunkt als auch eine Ereignisanzeige angezeigt. Da beide Systeme die gleiche Grundlage benutzen, finden sie sich auch beide in dem gleichen Programm wieder. Der Benutzer kann über einen Registerreiter den jeweiligen Modus auswählen.

1.2. Der Ober: Platzierung von neuen Gruppen

Die erste Teilaufgabe greift die schon zuvor beschriebene Fragmentierungsproblematik auf. Im Gegensatz zu Dateisystemen finden sich hier jedoch einige markante Unterschiede, welche die Suche nach einem passenden Algorithmus vereinfachen. Im folgenden seien daher einige normative Annahmen eines realistischen Restaurants dargelegt:

[2] Es wird maximal eine neue Gruppe hinzugefügt, damit die Neuankünfte für den Benutzer besser ersichtlich sind

First in, First out

Auch wenn jede Gruppe von Gästen zu einer beliebigen Zeit an- und abreisen kann, so kann im Durchschnitt davon ausgegangen werden, dass Gruppen, welche als erstes ankommen, auch als erstes wieder das Restaurant verlassen werden. Abgesehen von der Varianz in der Aufenthaltsdauer bildet sich generell also ein FiFo-Prinzip aus.

Da es sich bei dem Turmrestaurant um einen Kreis (d. h. effektiv eine Menge von Sitzplätzen ohne spezifischen Anfangs- und Endpunkt) handelt, und wir eine Art FiFo-Verhalten der Gäste haben, wird eine zu hohe Fragmentation schon durch den Aufbau des Restaurants selbst verhindert: Bei geringer Auslastung würde sich durch eine abreisende Gruppe der imaginäre Startpunkt – und damit gleichzeitig der Endpunkt – des Arrays im Kreis verschieben. Setzt man in solch einem Fall also jeweils neue Gruppen an das aktuelle Ende der belegten Sitzplätze, so hat man – solange keine weitere Fragmentation auftritt und generell genug freie Sitzplätze vorhanden sind – immer genügend aneinandergereihte, freie Sitzgelegenheiten.

Abbildung 9: Kreisverhalten

Leider funktioniert dieses Schema nur bei einer geringen Auslastung (d. h. bei wenig belegten Plätzen) – Je mehr Gäste da sind, desto kleiner wird nämlich die Anzahl der freien Plätze zwischen End- und Startpunkt.

Best Fit

Weiterhin bilden sich aufgrund der zufälligen bzw. verschiedenen Aufenthaltsdauer der einzelnen Gruppen Lücken innerhalb der Strecke der belegten Plätze. Es kann passieren, dass eine zweite Gruppe, welche später als die erste ankommt, dennoch früher das Restaurant verlässt.

Vor allem bei einer höheren Auslastung des Restaurants muss dann eine Gruppe innerhalb dieser Lücken platziert werden, weil sonst nicht genug Platz frei ist. Doch welche Lücke soll gewählt werden? Die kleinere oder die größere? Oder gar eine komplett andere?

Im simpelsten Fall hat eine Lücke genau die Größe der Gruppe. Wenn die Gruppe eine Lücke komplett ausfüllen kann, so soll sie auch dort platziert werden – Es gibt keine andere Möglichkeit, wie diese Lücke besser ausgefüllt werden könnte, und die übrigen Lücken können noch von anderen Gruppen belegt werden.

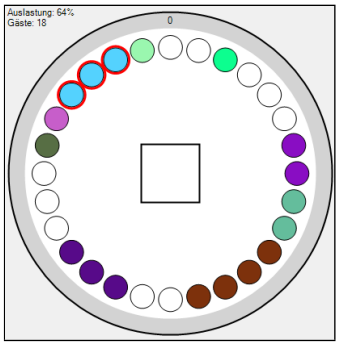


Abbildung 10: Fragmentierung

In den meisten Fällen gibt es aber keinen solchen „Perfect Fit“– Es muss also nach einem anderen Kriterium vorgegangen werden. Soll man also nun die größte oder kleinste Lücke auswählen? Dazu sei hypothetisch eine Mengenverteilung zur Gruppengröße der Gäste gegeben. Diese könnte dann entsprechend der Abbildung aussehen:

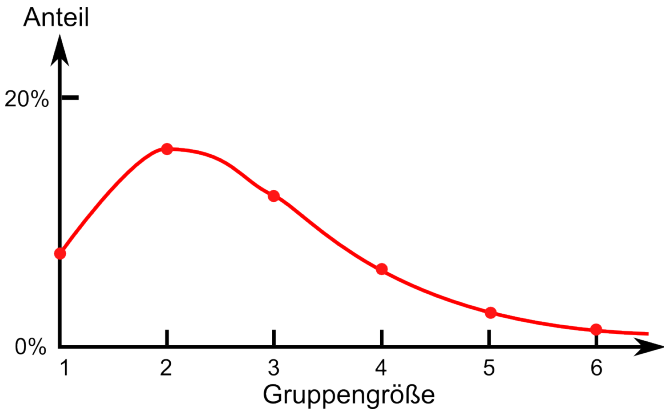


Abbildung 11: Anteilige Verteilung auf die jeweiligen Gruppengrößen

Betrachtet man diese Verteilung genauer, so erhält man folgende Gleichung:

$$\lim_{Groupsize \to \infty} Probability(Groupsize) = 0 \tag{2}$$

Je größer also die Gruppengröße ist, desto unwahrscheinlicher tritt dieser Fall auf. Dies spielt bei der Lückenwahl eine signifikante Rolle: Wählen wir nämlich zunächst die größere der Lücken, so fällt durch deren Verkleinerung nur der rechte Abschnitt der Verteilung weg – also nur ein insignifikanter Teil. Die meist-auftretenden Fälle werden immer noch abgedeckt. Würde man stattdessen die kleinere Lücke benutzen, so würde ein prozentual größerer Anteil von Gruppen ausgeschlossen.

Daher ist es – falls nötig – empfehlenswert immer den größten freien Block aufzuteilen. Teilt man einen größeren Block, so bleibt auch noch ein großer Block übrig, welcher deutlich mehr Fälle abdeckt als nur noch ein einzelner freier Sitzplatz.

Anmerkung: In der Aufgabenstellung ist nicht klar festgelegt, ob man eine einzelne Person auch als mögliche Gruppe ansehen kann. Da die „Group of one“ jedoch in der Programmierung genauso wie alle anderen Gruppen gehandhabt wird, und in einem Restaurant auch Einzelpersonen schon einmal anzutreffen sind, wird in meiner Umsetzung der Aufgabe dieser Fall erlaubt bzw. als gültige Gästemenge betrachtet.

Zusammenfassung

Fasst man die zuvor gewonnenen Erkenntnisse kurz zusammen, so kommt man auf einen einfachen Algorithmus für den Ober:

1. Der Ober muss jederzeit wissen, wo sich freie Plätze befinden und wie viele davon jeweils in Folge angereiht sind.
2. Falls möglich, so sollte der Ober die neu ankommende Gruppe in eine Lücke, welche genau der Gruppengröße entspricht, ansiedeln.
3. Ansonsten soll der Ober die größte freie Lücke suchen und dort die Gruppe hinzubitten.
4. Innerhalb dieser Lücke soll die neue Gruppe immer vom unteren Feldrand an gesetzt werden

Werden diese Regeln beachtet, so wird das K.O.-Kriterium so lang wie möglich vermieden. Bei geringer Auslastung ist die größte freie Lücke automatisch die Begrenzung zwischen End- und Startpunkt; bei hoher Auslastung werden die am wahrscheinlichsten auftretenden Fälle so lang wie möglich als abdeckbare Optionen offen gehalten.

1.3. Ein Streich: Das System aushebeln

Der von den Töchtern geplante Streich zielt darauf ab, das so eben erschaffene System auszuhebeln. Es geht also darum gezielt eine „Worst-Case“-Situation auszulösen - Und das mit sowenig Schülern wie möglich. Mein Algorithmus lässt sich dabei in zwei Teile gliedern:

Die benötigte Anzahl an Schülern kalkulieren: Trial and Error

Die benötigte Anzahl an Schülern hängt dabei von zwei Faktoren ab: Zum einen von der Anzahl der Sitzplätze in dem Restaurant, und zum anderen von der Qualität des verwendeten „Streich-Algorithmus“. Da dieser Zusammenhang nicht linear verläuft (vgl: Ablaufsprotokoll), wird die jeweils benötigte Anzahl von Schülern über ein „Trial-and-Error“-Verfahren bestimmt. Dabei sind für den Benutzer drei auswählbare Modi vorgesehen:

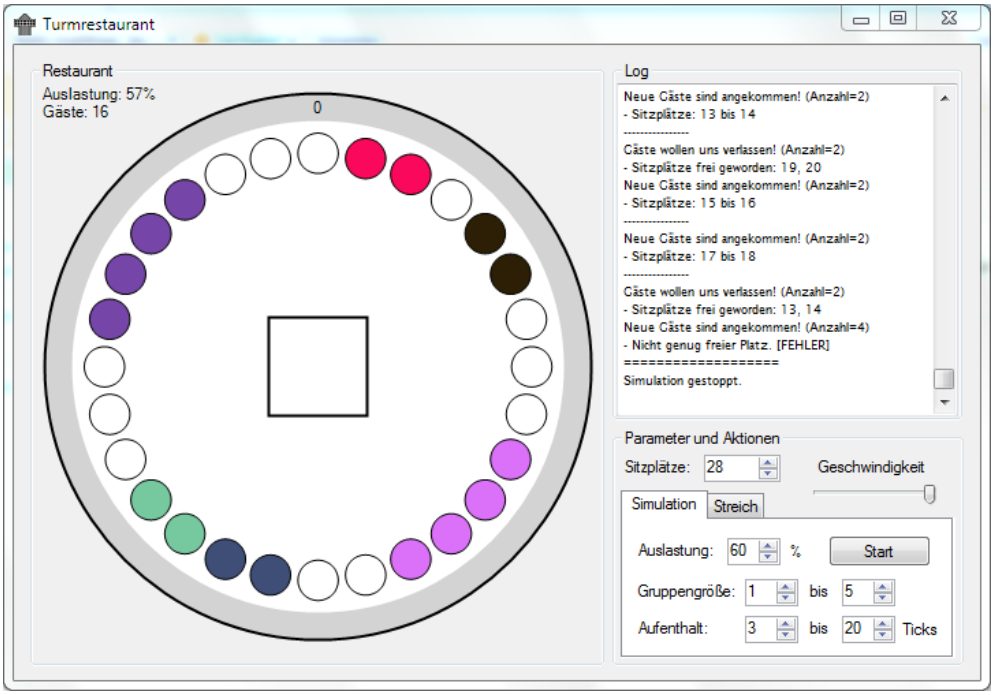


Abbildung 12: K.O.-Kriterium

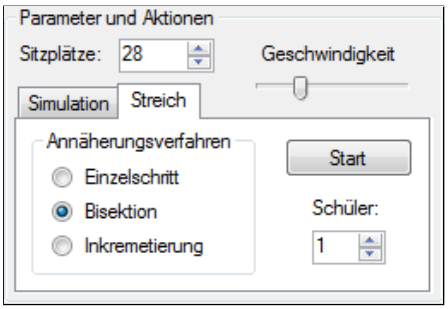


Abbildung 13: Annäherungsmodi

Einzelschritt: Es wird nur die vom Benutzer angegebene Anzahl von Schülern benutzt um dem Ober einen Streich zu spielen. Dabei wird am Ende im Log anzeigt, ob der Streich geglückt ist oder nicht.

Bisektion: Anhand eines bisektionalen Intervallschachtelungsverfahrens wird die benötigte Anzahl der Schüler bestimmt. Schlägt ein Streich fehl, wird die Anzahl der verfügbaren Schüler um d erhöht; ansonsten um d verringert. Anschließend wird d halbiert. Das Verfahren wird solange wiederholt bis $d = 0$. Als Startwert gilt $d > Sitze$. Dieses Verfahren bestimmt die Lösung am schnellsten und eignet sich daher zur genauen Bestimmung der notwendigen Schüler.

Inkrementierung: Da die Bisektion für den Benutzer nicht unbedingt intuitiv verständlich ist, habe ich auch ein einfaches Inkrementierungsverfahren implementiert: Die Anzahl der verfügbaren Schüler wird solange um jeweils eins erhöht bis der Streich das erste Mal glückt. Durch dieses Vorgehen wird die Abhängigkeit des „Streich-Algorithmus“ von der Anzahl der verfügbaren Schüler deutlich und der Algorithmus allgemein besser verständlich.

Der eigentliche Streich

Doch wie führt man nun einen Streich mit m gegebenen Schülern durch? Um das K.O.-Kriterium gezielt auszulösen, muss man zunächst für eine Fragmentierung des Restaurants sorgen. Sobald anschließend der größte, freie Block kleiner als die Anzahl der verbleibenden Schüler ist, kann man zuschlagen: Der Ober kann die Gruppe dann nämlich nicht mehr nebeneinander platzieren und ärgert sich.

Insgesamt hängt der Algorithmus stark von den verfügbaren Schülern ab. Es wäre unklug alle Schüler zur Fragmentierung einzusetzen, da man so am Ende keine Schüler mehr für den „finalen Angriff“ übrig hätte. Auch kann die Fragmentierung nicht willkürlich erfolgen, da der Ober die Schüler sonst wieder gezielt an gut passende Stellen setzen kann.

Als erstes sollte jedoch auf jeden Fall ein einzelner Schüler in das Restaurant geschickt werden, welcher dort permanent sitzen bleibt. Dieser einzelne Schüler sorgt nämlich dafür, dass das Restaurant kein Endlosband mehr darstellt, sondern eine Art Array mit klarem Anfangs- und Endpunkt.

Da wir die Absicht haben am Ende einen (im metaphorischem Sinne) vernichtenden Schlag gegen den Ober auszuführen, müssen wir dafür sorgen, dass es keinen freien Block mehr gibt, welcher größer als die Angriffsmenge ist. Dies erreicht man, indem man sich zu Beginn eine bestimmte Teilmenge der verfügbaren Schüler (etwa $m_T = \frac{1}{2}m$) für diesen finalen Schlag aufhebt, und anschließend nur Lücken kleiner als diese Teilmenge generiert.

Lücken selber werden dabei generiert, indem wir zunächst eine große Menge an Schülern in das Restaurant schicken, und anschließend einen einzelnen Schüler hinterher,

welcher die Randmarkierung dieser Lücke darstellt. Die größere, erste Gruppe verlässt das Restaurant nun wieder und kann erneut verwendet werden; der einzelne Schüler bleibt sitzen – Ein Fragment mit der Größe der ersten Gruppe entsteht.

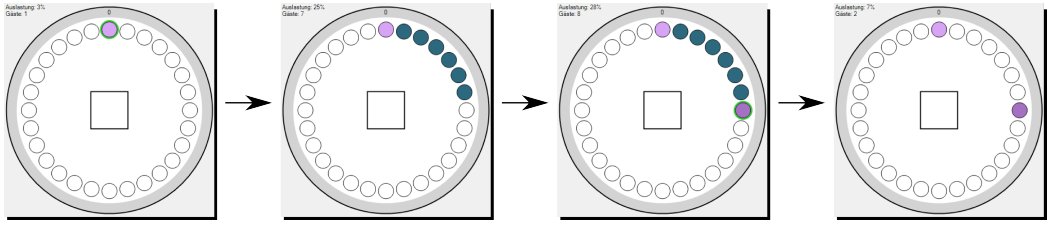


Abbildung 14: Das Prinzip des Streiches: Gezielte Fragmentierung

In den weiteren Schritten darf jedoch die Gruppengröße, welche zur Fragmentierung eingesetzt wird, nicht identisch bleiben. Wäre dies der Fall, so würde der Ober die Gruppe entsprechend immer in der ersten Lücke platzieren. Variiert man sie jedoch, so hat er keine Wahl als sie wo anders zu platzieren.

Ob man dabei mit aufsteigender oder absteigender Gruppengröße arbeitet hängt dabei vom verwendeten Oberalgorithmus ab. Ich habe mich für eine absteigende Gruppengröße entschieden, da der Algorithmus so einfacher zu implementieren war und die Platzierung stabiler verläuft, da am Ende jeweils nur mit kleinen Gruppen gearbeitet wird.

Diese Fragmentierung muss nun nur solange fortgesetzt werden, bis es keine Lücke mehr gibt, welche die verbleibende Anzahl an Schülern – also sowohl solche, die vorher aufgehoben wurden, als auch diejenigen, welche von der Fragmentierung noch übrig sind – aufnehmen kann. Sobald dieser Zustand eintritt ist das Ziel erreicht. Der Ober ärgert sich.

Der Algorithmus sei dabei noch einmal grafisch mit einigen weiteren Schritten dargelegt:

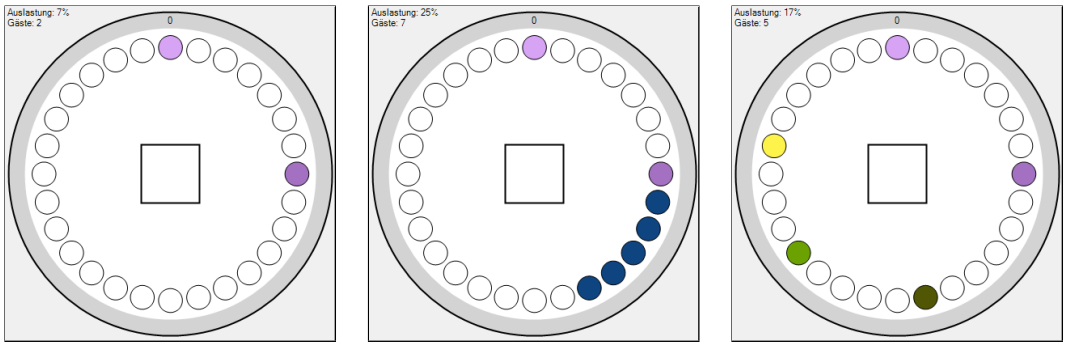


Abbildung 15: Weitere Schritte des Streiches

2. Dokumentation

Die Umsetzung der Lösungsidee erfolgt analog in mehreren Klassen, welche ich im folgenden vorstellen werde. Die Algorithmen wurden dabei identisch der Beschreibung umgesetzt und sind genauer im Quellcode ersichtlich, sollten aber auch so für jeden verständlich sein. Das Programm besteht aus folgenden Klassen:

Program.cs: Programmeinstiegspunkt; Öffnet die MainForm.

MainForm.cs: Stellt die grafische Schnittstelle dar. Ermöglicht das Anpassen einzelner Parameter und das Initiieren von Aktionen. Enthält einen einfachen Zeitgeber.

RestaurantControl.cs: Sorgt für die visuelle Repräsentation des Restaurant. Zeichnet den aktuellen Status des Restaurants samt Sitzbelegung (vgl. Screenshots). Frei skalierbar.

Utils.cs: Enthält diverse Hilfsmethoden (Zufall, . . .), welche von verschiedenen Klassen genutzt werden.

Group.cs: Stellt eine Gruppe von Besuchern dar. Einfache Containerklasse, welche die Attribute der Gruppe sammelt (Verbleibende Aufenthaltsdauer und farbliche Darstellung).

Restaurant.cs: Sorgt für die Verwaltung des Restaurants (Ober). Enthält die aktuelle Sitzbelegung und weist neuen Gruppen jeweils die optimalen Plätze zu. (Umsetzung der ersten Teilaufgabe.)

Simulation.cs: Liefert anhand der zuvor eingestellten Parameter und anhand der Auslastung neue, zufällige Besuchergruppen, welche das Restaurant füllen. (Erweiterung der ersten Teilaufgabe.)

Streich.cs: Umsetzung der zweiten Aufgabe und zugleich alternative Besucherquelle. Versucht dem Ober gezielt einen Streich zu spielen. (Umsetzung der zweiten Teilaufgabe.)

Einstellbare Parameter

Aufgrund des möglichen Freiraumes der Aufgabe habe ich mich für einige, als sinnvoll erscheinende Parameter entschieden:

Generell

- **Sitzplätze:** Stellt die Anzahl der Sitzplätze (1-999) im Restaurant ein.

- **Geschwindigkeit:** Geschwindigkeit, mit welcher das Restaurant abläuft. [Stillstand bis „as fast as possible“]

Simulation

- **Auslastung (%):** Richtwert für die Simulation. Sobald die Auslastung kleiner als der Richtwert ist, wird eine neue Gruppe in das Restaurant geschickt.
- **Gruppengröße (min. und max.):** Möglicher Größenbereich für neue Gruppen in der Simulation.
- **Aufenthaltsdauer (min. und max.):** Bereich der möglichen Aufenthaltsdauer in Ticks (Simulationsschritten).

Streich

- **Anzahl der Schüler für den Streich:** Bestimmt, wie viele Schüler den Töchtern für den Streich zur Verfügung stehen. Wird je nach gewähltem Annäherungsverfahren automatisch verändert.
- **Annäherungsverfahren:** Einzelschritt, Bisektion oder Inkrementierung (vgl. Lösungsidee).

Die Beschreibung der einzelnen Methoden und Funktionsweisen kann dabei im Quellcode in den XML-Kommentaren genauer eingesehen werden.

3. Erweiterungen

Auch wenn die meisten Erweiterungen schon ihren Fluss in die Dokumentation gefunden haben oder in dem Kapitel Lösungsidee bereits erwähnt wurden, seien sie der Vollständigkeit halber noch einmal kurz aufgelistet:

Visualisierung: Da die Visualisierung bei solchen Algorithmen einen besonderen Rolle spielt, wurde auch dementsprechend viel Wert auf die passende und ersichtliche Darstellung derselben gelegt. Deshalb wird das ganze Restaurant in seinem aktuellen Zustand auch jeweils grafisch dargestellt. Jede Gruppe erhält dabei ihre eigene Farbe; Neuankömmlinge werden grün; bald abreisende Gäste rot umrandet.

Simulation: In der ersten Teilaufgabe wird explizit nur ein Programm zur Platzierung neuer Gäste bzw. Gruppen gesucht, aber keine Umgebung, in welcher dieses Programm angewendet wird. Daher habe ich mich dazu entschlossen eine kleine Restaurantsimulation zu erstellen, welche anhand realitätsnaher Werte regelmäßig Gruppen zu dem Restaurant hinzufügt und entfernt, und damit die Fragmentierungsproblematik sowie die Strategie des Obers darstellt.

Parameter: Um den Nutzer alle Einstellungsmöglichkeiten einzuräumen wurden viele, aber sinnvolle verstellbare Parameter hinzugefügt, welche es ermöglichen (fast) alle erdenklichen Situationen zu erzeugen.

Approximation: Da in der Aufgabenstellung nach der Anzahl der notwendigen Schüler für den Streich gefragt wurde, bat es sich an zur schnellen Annäherung derselben einen kleinen Algorithmus auf Basis der Intervallschachtelung zu implementieren.

4. Bedienung

Die Bedienung des Programs ist intuitiv. Auf der linken Seite des Fensters findet sich die grafische Darstellung des aktuellen Restaurantzustandes; auf der rechten Seite sich eine Ereignisanzeige (Log). Im unteren rechten Bereich sind alle Elemente zur Steuerung durch den Nutzer angelegt: Der Benutzer kann durch den jeweiligen Registerreiter auswählen, ob er sich mit der ersten Teilaufgabe (Simulation) oder mit der zweiten Teilaufgabe (Streich) beschäftigen will, und kann anschließend verschiedene Parameter variieren und die Simulation dadurch gezielt steuern. Oberhalb der Registerreiter finden sich dahingegen die Elemente, welche beiden Teilaufgaben angehören oder das Restaurant selber beeinflussen (Anzahl der verfügbaren Sitzplätze und die Geschwindigkeit der Simulation).

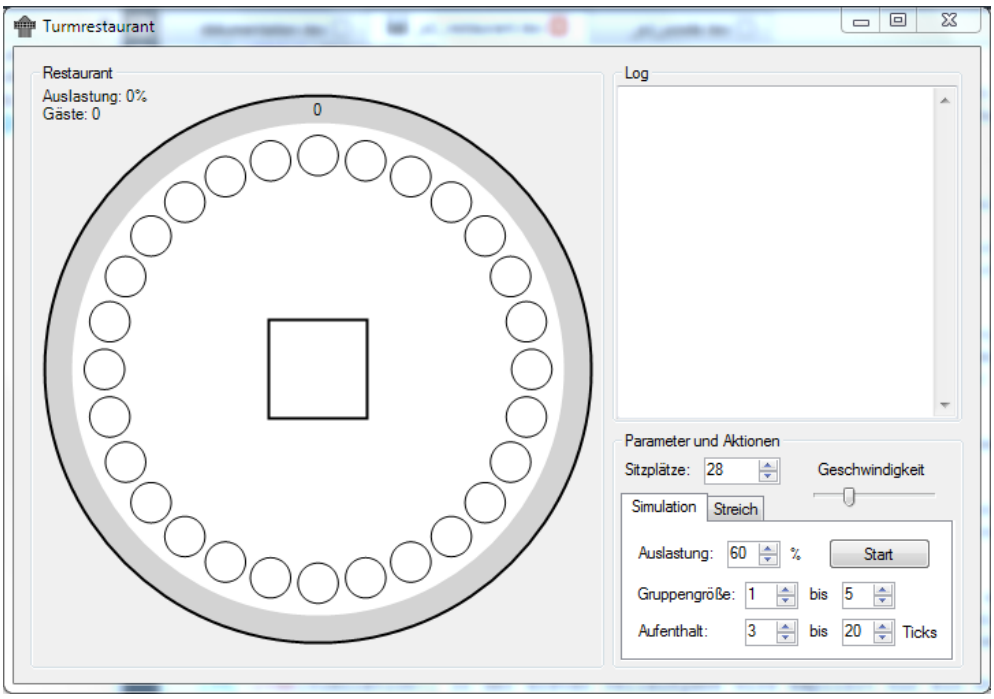


Abbildung 16: Benutzeroberfläche

5. Ablaufprotokolle

In diesem Abschnitt werde ich einige Ablaufprotokoll der beiden Teilaufgaben darstellen. Dabei werde ich jeweils eine für den jeweiligen Fall repräsentative Darstellung des Restaurants aufzeigen, und anschließend das Ereignisprotokoll abdrucken.

5.1. Restaurant-Simulation

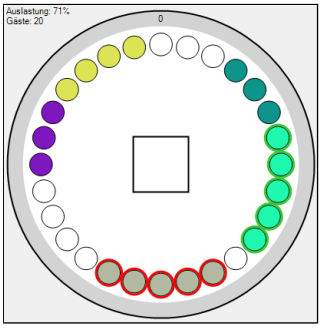
Modus: Simulation

Sitzplätze: 28 Sitze

Auslastung (Richtwert): 60%

Gruppengröße: 1 bis 5 Gäste

Aufenthaltsdauer: 3 bis 20 Ticks




```
Starte Simulation...
----------------
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 0 bis 4
----------------
Neue Gäste sind angekommen! (Anzahl=2)
- Sitzplätze: 5 bis 6
----------------
Neue Gäste sind angekommen! (Anzahl=2)
- Sitzplätze: 7 bis 8
----------------
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 9 bis 13
----------------
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 14 bis 16
----------------
----------------
----------------
----------------
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden: 0, 1, 2, 3, 4
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 17 bis 21
----------------
----------------
[..]
----------------
Gäste wollen uns verlassen! (Anzahl=2)
- Sitzplätze frei geworden: 10, 11
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden: 0, 1, 2, 3, 4
Neue Gäste sind angekommen! (Anzahl=2)
- Sitzplätze: 10 bis 11 [Lücke gefüllt]
----------------
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden: 12, 13, 14, 15, 16
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 12 bis 16
----------------
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 0 bis 4
----------------
----------------
----------------
----------------
----------------
----------------
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden: 0, 1, 2, 3, 4
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 0 bis 0
----------------
Gäste wollen uns verlassen! (Anzahl=2)
- Sitzplätze frei geworden: 10, 11
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 17 bis 20
----------------
----------------
----------------
Gäste wollen uns verlassen! (Anzahl=3)
- Sitzplätze frei geworden: 7, 8, 9
Gäste wollen uns verlassen! (Anzahl=4)
- Sitzplätze frei geworden: 24, 25, 26, 27
Neue Gäste sind angekommen! (Anzahl=2)
- Sitzplätze: 1 bis 2
----------------
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 3 bis 5
----------------
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 21 bis 23
----------------
Gäste wollen uns verlassen! (Anzahl=2)
- Sitzplätze frei geworden: 1, 2
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 24 bis 27 [Lücke gefüllt]
----------------
----------------
----------------
Gäste wollen uns verlassen! (Anzahl=1)
- Sitzplätze frei geworden: 0
Gäste wollen uns verlassen! (Anzahl=4)
- Sitzplätze frei geworden: 17, 18, 19, 20
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 6 bis 10
```

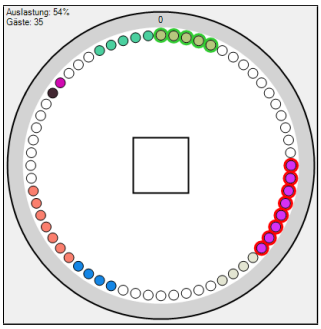
Modus: Simulation

Sitzplätze: 64 Sitze

Auslastung (Richtwert): 60%

Gruppengröße: 1 bis 8 Gäste

Aufenthaltsdauer: 5 bis 30 Ticks




```
Starte Simulation...
----------------
Neue Gäste sind angekommen! (Anzahl=8)
- Sitzplätze: 0 bis 7
----------------
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 8 bis 8
----------------
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 9 bis 9
----------------
Neue Gäste sind angekommen! (Anzahl=7)
- Sitzplätze: 10 bis 16
----------------
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 17 bis 20
----------------
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 21 bis 21
----------------
[..]
----------------
Gäste wollen uns verlassen! (Anzahl=1)
- Sitzplätze frei geworden: 9
----------------
Gäste wollen uns verlassen! (Anzahl=3)
- Sitzplätze frei geworden: 30, 31, 32
Neue Gäste sind angekommen! (Anzahl=2)
- Sitzplätze: 40 bis 41
----------------
Neue Gäste sind angekommen! (Anzahl=7)
- Sitzplätze: 42 bis 48
----------------
Gäste wollen uns verlassen! (Anzahl=6)
- Sitzplätze frei geworden: 22, 23, 24, 25, 26, 27
----------------
----------------
Gäste wollen uns verlassen! (Anzahl=2)
- Sitzplätze frei geworden: 40, 41
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 49 bis 52
----------------
Gäste wollen uns verlassen! (Anzahl=1)
- Sitzplätze frei geworden: 21
----------------
Gäste wollen uns verlassen! (Anzahl=1)
- Sitzplätze frei geworden: 8
----------------
Gäste wollen uns verlassen! (Anzahl=7)
- Sitzplätze frei geworden: 10, 11, 12, 13, 14, 15, 16
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 30 bis 32 [Lücke gefüllt]
----------------
Gäste wollen uns verlassen! (Anzahl=4)
- Sitzplätze frei geworden: 17, 18, 19, 20
Gäste wollen uns verlassen! (Anzahl=2)
- Sitzplätze frei geworden: 33, 34
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden: 35, 36, 37, 38, 39
Gäste wollen uns verlassen! (Anzahl=4)
- Sitzplätze frei geworden: 49, 50, 51, 52
Neue Gäste sind angekommen! (Anzahl=8)
- Sitzplätze: 8 bis 15
----------------
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 49 bis 53
----------------
Neue Gäste sind angekommen! (Anzahl=8)
- Sitzplätze: 16 bis 23
----------------
Gäste wollen uns verlassen! (Anzahl=8)
- Sitzplätze frei geworden: 0, 1, 2, 3, 4, 5, 6, 7
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 54 bis 54
----------------
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 55 bis 55
----------------
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 24 bis 27 [Lücke gefüllt]
----------------
Gäste wollen uns verlassen! (Anzahl=2)
- Sitzplätze frei geworden: 28, 29
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 56 bis 58
----------------
Gäste wollen uns verlassen! (Anzahl=3)
- Sitzplätze frei geworden: 30, 31, 32
Neue Gäste sind angekommen! (Anzahl=8)
- Sitzplätze: 28 bis 35
----------------
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden: 49, 50, 51, 52, 53
----------------
----------------
Gäste wollen uns verlassen! (Anzahl=8)
- Sitzplätze frei geworden: 8, 9, 10, 11, 12, 13, 14, 15
Neue Gäste sind angekommen! (Anzahl=7)
- Sitzplätze: 59 bis 1
----------------
----------------
Gäste wollen uns verlassen! (Anzahl=7)
- Sitzplätze frei geworden: 42, 43, 44, 45, 46, 47, 48
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 36 bis 39
----------------
Gäste wollen uns verlassen! (Anzahl=3)
- Sitzplätze frei geworden: 56, 57, 58
Gäste wollen uns verlassen! (Anzahl=8)
- Sitzplätze frei geworden: 28, 29, 30, 31, 32, 33, 34, 35
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 0 bis 3
----------------
Neue Gäste sind angekommen! (Anzahl=7)
- Sitzplätze: 40 bis 46
----------------
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 0 bis 4
```

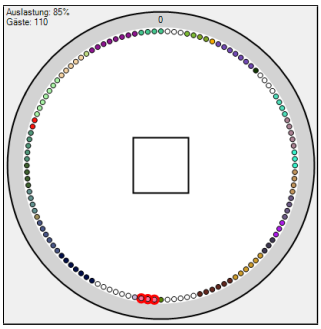
Modus: Simulation

Sitzplätze: 128 Sitze

Auslastung (Richtwert): 90%

Gruppengröße: 1 bis 8 Gäste

Aufenthaltsdauer: 20 bis 50 Ticks




```
Starte Simulation...
----------------
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 0 bis 0
----------------
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 1 bis 3
----------------
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 4 bis 7
----------------
Neue Gäste sind angekommen! (Anzahl=6)
- Sitzplätze: 8 bis 13
----------------
[..]
----------------
Gäste wollen uns verlassen! (Anzahl=8)
- Sitzplätze frei geworden: 76, 77, 78, 79, 80, 81, 82, 83
Gäste wollen uns verlassen! (Anzahl=2)
- Sitzplätze frei geworden:
Neue Gäste sind angekommen! (Anzahl=6)
- Sitzplätze: 58 bis 63
----------------
Gäste wollen uns verlassen! (Anzahl=1)
- Sitzplätze frei geworden:
Gäste wollen uns verlassen! (Anzahl=1)
- Sitzplätze frei geworden:
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 64 bis 64
----------------
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 65 bis 67
----------------
Gäste wollen uns verlassen! (Anzahl=1)
- Sitzplätze frei geworden:
----------------
[..]
----------------
Gäste wollen uns verlassen! (Anzahl=3)
- Sitzplätze frei geworden:
Gäste wollen uns verlassen! (Anzahl=6)
- Sitzplätze frei geworden:
Gäste wollen uns verlassen! (Anzahl=8)
- Sitzplätze frei geworden: 6, 7
Neue Gäste sind angekommen! (Anzahl=2)
- Sitzplätze: 6 bis 7 [Lücke gefüllt]
----------------
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden:
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 97 bis 101
----------------
[..]
----------------
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 37 bis 40
----------------
Gäste wollen uns verlassen! (Anzahl=1)
- Sitzplätze frei geworden: 64
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 111 bis 115
----------------
Gäste wollen uns verlassen! (Anzahl=8)
- Sitzplätze frei geworden:
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 41 bis 43
----------------
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 44 bis 46
----------------
----------------
Gäste wollen uns verlassen! (Anzahl=1)
- Sitzplätze frei geworden:
----------------
Gäste wollen uns verlassen! (Anzahl=6)
- Sitzplätze frei geworden: 4, 5
Neue Gäste sind angekommen! (Anzahl=2)
- Sitzplätze: 4 bis 5 [Lücke gefüllt]
----------------
Gäste wollen uns verlassen! (Anzahl=4)
- Sitzplätze frei geworden:
Gäste wollen uns verlassen! (Anzahl=4)
- Sitzplätze frei geworden:
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 64 bis 64 [Lücke gefüllt]
----------------
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 116 bis 116
----------------
Neue Gäste sind angekommen! (Anzahl=8)
- Sitzplätze: 117 bis 124
----------------
Neue Gäste sind angekommen! (Anzahl=6)
- Sitzplätze: 47 bis 52
----------------
Gäste wollen uns verlassen! (Anzahl=4)
- Sitzplätze frei geworden: 0, 1, 2, 3
Gäste wollen uns verlassen! (Anzahl=6)
- Sitzplätze frei geworden: 58, 59, 60, 61, 62, 63
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden: 16, 17, 18, 19, 20
Neue Gäste sind angekommen! (Anzahl=6)
- Sitzplätze: 53 bis 58
----------------
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 125 bis 0
----------------
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 16 bis 16
----------------
----------------
----------------
----------------
Gäste wollen uns verlassen! (Anzahl=6)
- Sitzplätze frei geworden: 69, 70, 71, 72, 73, 74
Neue Gäste sind angekommen! (Anzahl=8)
- Nicht genügend freie Plätze. [FEHLER]
====================
Simulation gestoppt.
```

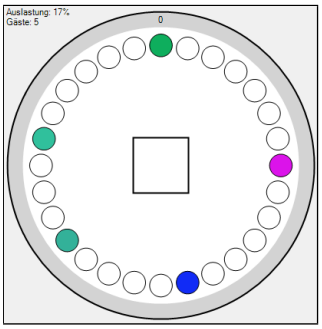
5.2. Streich der Töchter

Modus: Streich

Sitzplätze: 28 Sitze

Verfahren: Bisektion

Benötigte Schüler: 12 Schüler



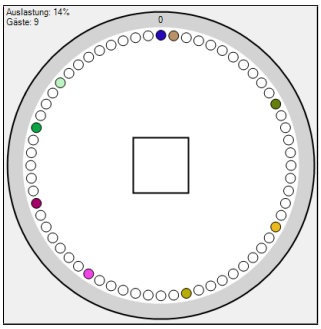

```
Streich mit 12 Schülern...!
====================
1. Startmarkierung setzen
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 0 bis 0
------------------------------
2. Wir heben uns 4 Schüler für das Ende auf.
------------------------------
3. Fragmentierung verursachen ...
.............................
Neue Gäste sind angekommen! (Anzahl=6)
- Sitzplätze: 1 bis 6
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 7 bis 7
Gäste wollen uns verlassen! (Anzahl=6)
- Sitzplätze frei geworden: 1, 2, 3, 4, 5, 6
.............................
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 8 bis 12
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 13 bis 13
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden: 8, 9, 10, 11, 12
.............................
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 14 bis 17
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 18 bis 18
Gäste wollen uns verlassen! (Anzahl=4)
- Sitzplätze frei geworden: 14, 15, 16, 17
.............................
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 19 bis 21
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 22 bis 22
Gäste wollen uns verlassen! (Anzahl=3)
- Sitzplätze frei geworden: 19, 20, 21
.............................
------------------------------
4. Mit allen Schülern 'angreifen'!
Neue Gäste sind angekommen! (Anzahl=7)
- Nicht genügend freie Plätze. [FEHLER]
====================
Erfolg: Ja!
12 Schüler benötigt!
Approximation vollständig.
==================
Streich gestoppt.
```

Modus: Streich

Sitzplätze: 64 Sitze

Verfahren: Bisektion

Benötigte Schüler: 19 Schüler



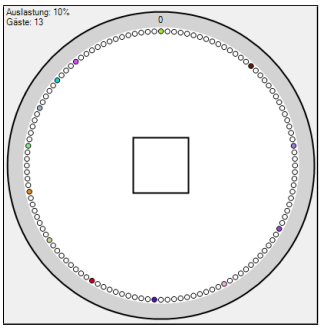

```
Streich mit 19 Schülern...!
====================
1. Startmarkierung setzen
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 0 bis 0
------------------------------
2. Wir heben uns 7 Schüler für das Ende auf.
------------------------------
3. Fragmentierung verursachen ...
.............................
Neue Gäste sind angekommen! (Anzahl=10)
- Sitzplätze: 1 bis 10
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 11 bis 11
Gäste wollen uns verlassen! (Anzahl=10)
- Sitzplätze frei geworden: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
.............................
Neue Gäste sind angekommen! (Anzahl=9)
- Sitzplätze: 12 bis 20
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 21 bis 21
Gäste wollen uns verlassen! (Anzahl=9)
- Sitzplätze frei geworden: 12, 13, 14, 15, 16, 17, 18, 19, 20
.............................
Neue Gäste sind angekommen! (Anzahl=8)
- Sitzplätze: 22 bis 29
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 30 bis 30
Gäste wollen uns verlassen! (Anzahl=8)
- Sitzplätze frei geworden: 22, 23, 24, 25, 26, 27, 28, 29
.............................
Neue Gäste sind angekommen! (Anzahl=7)
- Sitzplätze: 31 bis 37
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 38 bis 38
Gäste wollen uns verlassen! (Anzahl=7)
- Sitzplätze frei geworden: 31, 32, 33, 34, 35, 36, 37
.............................
Neue Gäste sind angekommen! (Anzahl=6)
- Sitzplätze: 39 bis 44
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 45 bis 45
Gäste wollen uns verlassen! (Anzahl=6)
- Sitzplätze frei geworden: 39, 40, 41, 42, 43, 44
.............................
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 46 bis 50
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 51 bis 51
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden: 46, 47, 48, 49, 50
.............................
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 52 bis 55
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 1 bis 1
Gäste wollen uns verlassen! (Anzahl=4)
- Sitzplätze frei geworden: 52, 53, 54, 55
.............................
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 52 bis 54
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 55 bis 55
Gäste wollen uns verlassen! (Anzahl=3)
- Sitzplätze frei geworden: 52, 53, 54
.............................
------------------------------
4. Mit allen Schülern 'angreifen'!
Neue Gäste sind angekommen! (Anzahl=10)
- Nicht genügend freie Plätze. [FEHLER]
====================
Erfolg: Ja!
19 Schüler benötigt!
Approximation vollständig.
====================
Streich gestoppt.
```

Modus: Streich

Sitzplätze: 128 Sitze

Verfahren: Bisektion

Benötigte Schüler: 28 Schüler




```
Streich mit 28 Schülern...!
====================
1. Startmarkierung setzen
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 0 bis 0
------------------------------
2. Wir heben uns 12 Schüler für das Ende auf.
------------------------------
3. Fragmentierung verursachen ...
.............................
Neue Gäste sind angekommen! (Anzahl=14)
- Sitzplätze: 1 bis 14
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 15 bis 15
Gäste wollen uns verlassen! (Anzahl=14)
- Sitzplätze frei geworden: 1, 2, 3, 4, 5, 6, 7,
8, 9, 10, 11, 12, 13, 14
.............................
Neue Gäste sind angekommen! (Anzahl=13)
- Sitzplätze: 16 bis 28
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 29 bis 29
Gäste wollen uns verlassen! (Anzahl=13)
- Sitzplätze frei geworden: 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28
.............................
Neue Gäste sind angekommen! (Anzahl=12)
- Sitzplätze: 30 bis 41
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 42 bis 42
Gäste wollen uns verlassen! (Anzahl=12)
- Sitzplätze frei geworden: 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41
.............................
Neue Gäste sind angekommen! (Anzahl=11)
- Sitzplätze: 43 bis 53
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 54 bis 54
Gäste wollen uns verlassen! (Anzahl=11)
- Sitzplätze frei geworden: 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53
.............................
Neue Gäste sind angekommen! (Anzahl=10)
- Sitzplätze: 55 bis 64
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 65 bis 65
Gäste wollen uns verlassen! (Anzahl=10)
- Sitzplätze frei geworden: 55, 56, 57, 58, 59,
60, 61, 62, 63, 64
.............................
Neue Gäste sind angekommen! (Anzahl=9)
- Sitzplätze: 66 bis 74
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 75 bis 75
Gäste wollen uns verlassen! (Anzahl=9)
- Sitzplätze frei geworden: 66, 67, 68, 69, 70,
71, 72, 73, 74
.............................
Neue Gäste sind angekommen! (Anzahl=8)
- Sitzplätze: 76 bis 83
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 84 bis 84
Gäste wollen uns verlassen! (Anzahl=8)
- Sitzplätze frei geworden: 76, 77, 78, 79, 80,
81, 82, 83
.............................
Neue Gäste sind angekommen! (Anzahl=7)
- Sitzplätze: 85 bis 91
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 92 bis 92
Gäste wollen uns verlassen! (Anzahl=7)
- Sitzplätze frei geworden: 85, 86, 87, 88, 89,
90, 91
.............................
Neue Gäste sind angekommen! (Anzahl=6)
- Sitzplätze: 93 bis 98
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 99 bis 99
Gäste wollen uns verlassen! (Anzahl=6)
- Sitzplätze frei geworden: 93, 94, 95, 96, 97,
98
.............................
Neue Gäste sind angekommen! (Anzahl=5)
- Sitzplätze: 100 bis 104
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 105 bis 105
Gäste wollen uns verlassen! (Anzahl=5)
- Sitzplätze frei geworden: 100, 101, 102, 103,
104
.............................
Neue Gäste sind angekommen! (Anzahl=4)
- Sitzplätze: 106 bis 109
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 110 bis 110
Gäste wollen uns verlassen! (Anzahl=4)
- Sitzplätze frei geworden: 106, 107, 108, 109
.............................
Neue Gäste sind angekommen! (Anzahl=3)
- Sitzplätze: 111 bis 113
Neue Gäste sind angekommen! (Anzahl=1)
- Sitzplätze: 114 bis 114
Gäste wollen uns verlassen! (Anzahl=3)
- Sitzplätze frei geworden: 111, 112, 113
.............................
------------------------------
4. Mit allen Schülern 'angreifen'!
Neue Gäste sind angekommen! (Anzahl=15)
- Nicht genügend freie Plätze. [FEHLER]
====================
Erfolg: Ja!
28 Schüler benötigt!
Approximation vollständig.
====================
Streich gestoppt.
```

6. Auswertung

In diesem Abschnitt möchte ich auf die dritte Teilaufgabe der 2. Aufgabe eingehen. Dazu werde ich jeweils zu den ersten beiden Teilaufgaben eine Bewertung abgeben, und anschließend noch die für den Streich benötigten Schülerzahlen visualisieren.

6.1. Restaurant-Simulation

Die Effizienz von dem Ober-Algorithmus hängt sehr von der Auslastung ab. Bei Auslastungen $< 60\%$ ärgert er sich während praktisch nie. Wird die Auslastung jedoch höher, erhöht sich zugleich auch stetig die Wahrscheinlichkeit, dass das K.O.-Kriterium eintritt.

Insgesamt wird die Fragmentierungsproblematik jedoch gut verhindert. Der Algorithmus kommt ohne Probleme mit einer standardmäßigen, zufälligen Gruppenlogik klar und versucht möglichst lange einen großen, freien Block freizuhalten. Würde es möglich sein Gruppen während ihres Aufenthalts zu relokalisieren (Defragmentation) wäre das K.O.-Kriterium vollständig ausgeschlossen.

Die Simulation selber zeigt deutlich wie der Algorithmus arbeitet und erlaubt es dem Nutzer nahezu spielend die Auswirkungen der einzelnen Faktoren auf den Algorithmus auszuprobieren; die Visualisierung ist in diesem Fall aber auch dementsprechend wichtig und kann möglicherweise nicht innerhalb dieses Dokumentes perfekt wiedergegeben werden. Anzuraten ist es daher, das Programm selber einmal auszuführen und sich die Ergebnisse anzuschauen.

6.2. Streich der Töchter

Lässt man beide Algorithmen gegeneinander antreten, so gilt es zu beachten, dass der zweite Algorithmus auf dem Wissen des ersten Algorithmus basiert, und daher gezielt in der Lage ist diesen zu kontern. In einem normalen Dateisystem/Restaurant wäre dies vermutlich nicht so einfach möglich. Dementsprechend ist der Streich-Algorithmus in seiner jetzigen Form aber auch nur auf genau diesen Oberalgorithmus anwendbar. Während er in den meisten Fällen vermutlich auch gegen andere Oberalgorithmen antreten könnte, so kann die Anzahl der notwendigen Schüler bis sich der Ober das erste Mal ärgert durchaus unterscheiden. Aber da der Streich von den Töchtern des Obers durchgeführt wird, liegt es nahe, dass sie auch so Zugang zu dem Restaurant haben und ihn genauer beobachten konnten.

Aufgrund der Aufgabenstellung wurde der Streichalgorithmus so entwickelt, dass er auf die Frage „Kann man mit M Schülern den Ober ärgern?“ klar mit „Ja“ oder „Nein“ antworten kann. Das gegebene Schülerlimit wird daher auch strikt eingehalten und nie überschritten. Während dies zur genauen Bestimmung der notwendigen Schülerzahl zu einigen Versuchen mit Niederlagen führt, ermöglicht es dafür auf der anderen Seite klare Aussagen über die Grenzen – Also ab wann sich der Ober überhaupt ärgert.

Anzumerken sei, dass der Algorithmus nur auf ein Restaurant mit mehr als drei Sitzplätzen ausgelegt ist. Während er unter dieser Grenze funktionieren mag, ist er dort nicht mehr effizient anwendbar bzw. nicht mehr sinnvoll. Dieser Fall ist jedoch zu vernachlässigen, da ein Restaurant mit weniger als vier Sitzgelegenheiten auch nicht mehr als normales Restaurant betrachtet werden kann.

Der Algorithmus arbeitet dabei in keinem Fall linear. Während er für weniger Sitzplätze noch recht viele Schüler benötigt, sinkt das Verhältnis bei steigender Sitzanzahl deutlich:

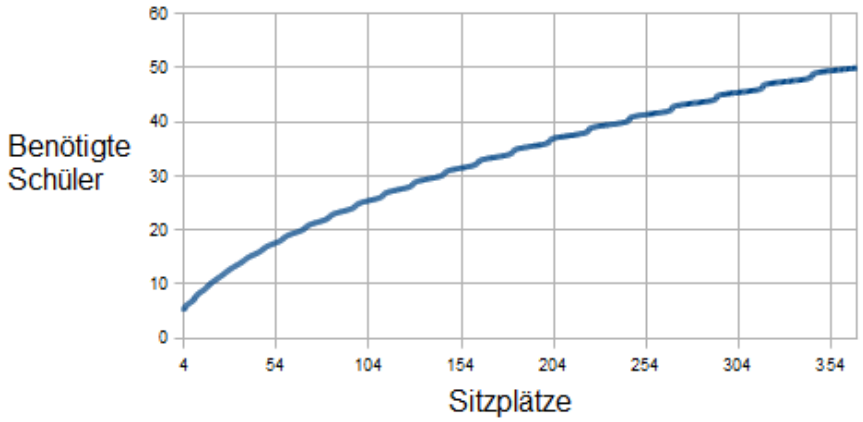


Abbildung 17: Anzahl der für den Streich notwendigen Schüler in Abhängigkeit von den Sitzgelegenheiten

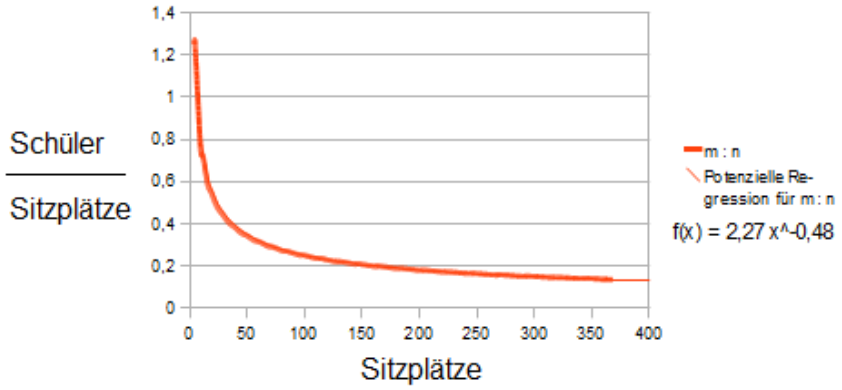


Abbildung 18: Verhältnis „Schüler : Sitzgelegenheiten“

Die Anzahl der notwendigen Schüler wird dabei durch eine mathematische Folge beschrieben:

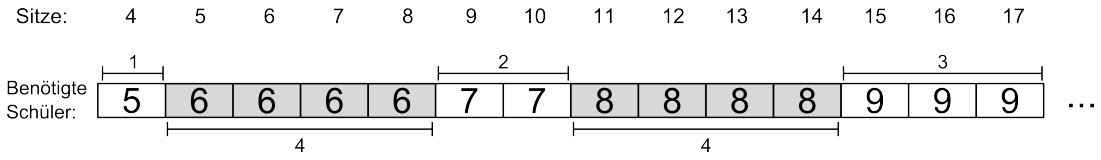


Abbildung 19: Genaue Bestimmung der notwendigen Schüler

Die Gültigkeitsgrenzen der jeweiligen benötigten Schüler werden dabei ersichtlich immer länger $(i, i+1, i+2, ...)$. Dadurch lässt sich auch erklären, dass der Algorithmus für mehr Sitzgelegenheiten auch verhältnismäßig weniger Schüler benötigt.

Aufgrund der Komplexität der Reihe lässt sich daher jedoch nicht ohne weiteres eine geschlossene Darstellung derselben finden. Stattdessen kann die Anzahl der notwendigen Schüler über eine potenzielle Regression des Verhältnisses „Schüler : Sitzgelegenheiten“ angenähert werden:

$$Schueler \approx 2,27 \cdot Sitzgelegenheiten^{-0,48} \cdot Sitzgelegenheiten \tag{3}$$

Befindet sich die Anzahl der verwendeten Schüler unterhalb der jeweiligen Grenze, so ärgert sich der Ober nie; befindet sie sich in bzw. über der Grenze, so ärgert er sich bei jedem versuchten Streich.

Abschließend will ich noch einige exakte Beispieldaten darlegen:

Sitzplätze	4	8	16	28	64	128	256	512	999
notwendige Schüler	5	6	9	12	19	28	41	59	85

7. Quellcode

Program.cs

```
1  using System;
2  using System.Windows.Forms;
3
4  namespace Turmrestaurant
5  {
6    /// <summary>
7    /// Program
8    /// </summary>
9    internal static class Program
10   {
11     /// <summary>
12     /// Der Haupteinstiegspunkt fuer die Anwendung.
13     /// </summary>
14     [STAThread]
15     private static void Main()
16     {
17       Application.EnableVisualStyles();
18       Application.SetCompatibleTextRenderingDefault(false);
19       Application.Run(new MainForm());
```

```
20      }
21    }
22  }
```

MainForm.cs

```
1  using System;
2  using System.Windows.Forms;
3
4  namespace Turmrestaurant
5  {
6    /// <summary>
7    /// Main Form
8    /// </summary>
9    internal partial class MainForm : Form
10   {
11     private Restaurant _restaurant;
12     private Timer _timer;
13
14     public MainForm()
15     {
16       InitializeComponent();
17       _restaurant = new Restaurant(28, Log);
18       cRestaurant.SetRestaurant(_restaurant);
19     }
20
21     /// <summary>
22     /// Log Nachricht
23     /// </summary>
24     /// <param name="msg"></param>
25     public void Log(string msg)
26     {
27       if (InvokeRequired)
28         BeginInvoke(new EventHandler((o1, o2) => Log(msg)));
29       else
30         cLog.AppendText(msg + Environment.NewLine);
31     }
32
33     /// <summary>
34     /// Clear Log
35     /// </summary>
36     public void ClearLog()
37     {
38       //Invoke fuer MultiThreading
39       if (InvokeRequired) Invoke(new EventHandler((o1, o2) => ClearLog()));
40       else cLog.Clear();
41     }
42
43     private void cSeats_ValueChanged(object sender, EventArgs e)
44     {
45       _restaurant = new Restaurant((int) cSeats.Value, Log);
46       cRestaurant.SetRestaurant(_restaurant);
47     }
48
49     private void cSpeed_Scroll(object sender, EventArgs e)
50     {
51       if (_timer == null)
52         return;
53
54       if (cSpeed.Value == 0)
55       {
56         _timer.Interval = int.MaxValue; //"Pause"
57         return;
58       }
59
60       _timer.Interval = 3000/cSpeed.Value;
61
62       //Stufenskalierung
```

```
63        if (cSpeed.Value >= 6)
64          _timer.Interval /= 2;
65        if (cSpeed.Value >= 12)
66          _timer.Interval /= 2;
67        if (cSpeed.Value >= 15)
68          _timer.Interval = 1;
69
70        //Simulation oder Streich?
71        if (tabStreich.Enabled)
72          _timer.Interval *= 2; //verlangsamen
73      }
74
75      private void bSimulation_Click(object sender, EventArgs e)
76      {
77        //Timer initialisieren?
78        if (_timer == null)
79        {
80          //Zuruecksetzen
81          foreach (Group g in _restaurant.Groups)
82            _restaurant.RemoveGroup(g);
83          cLog.Clear();
84
85          Log("Starte Simulation...");
86          bSimulation.Text = @"Stopp";
87
88          _timer = new Timer();
89          _timer.Tick += OnTick;
90          cSpeed_Scroll(sender, e); //Update speed
91          _timer.Start();
92
93          tabStreich.Enabled = false;
94        }
95        else
96        {
97          _timer.Stop();
98          _timer = null;
99          Log("====================");
100         Log("Simulation gestoppt.");
101         bSimulation.Text = @"Start";
102
103         tabStreich.Enabled = true;
104       }
105     }
106
107     private void bStreich_Click(object sender, EventArgs e)
108     {
109       //Timer initialisieren?
110       if (_timer == null)
111       {
112         //Zuruecksetzen
113         _restaurant.Clear();
114         cLog.Clear();
115
116         Log("Starte Streich...");
117         bStreich.Text = @"Stopp";
118
119         Streich.Start(_restaurant, this);
120         _timer = new Timer();
121         _timer.Tick += (o1, o2) =>
122                        {
123                          Streich.Tick();
124                          if (Streich.IsFinished)
125                            bStreich_Click(sender, e);
126                        };
127         cSpeed_Scroll(sender, e); //Update speed
128         _timer.Start();
129
130         tabSimulation.Enabled = false;
131         cSchueler.Enabled = false;
```

```
132         }
133         else
134         {
135           Streich.Stop();
136           _timer.Stop();
137           _timer = null;
138           Log("====================");
139           Log("Streich gestoppt.");
140           bStreich.Text = @"Start";
141 
142           tabSimulation.Enabled = true;
143           cSchueler.Enabled = true;
144         }
145       }
146 
147       /// <summary>
148       /// Simulations Tick
149       /// </summary>
150       private void OnTick(object sender, EventArgs args)
151       {
152         Log("--------------------");
153 
154         //Simulations-Tick
155         if (!Simulation.Tick(_restaurant, this))
156           bSimulation_Click(sender, args); //Bei Fehler stoppen
157 
158         //Darstellung
159         cRestaurant.Refresh();
160       }
161 
162       private void cMinStay_ValueChanged(object sender, EventArgs e)
163       {
164         if (cMinStay.Value > cMaxStay.Value)
165           cMinStay.Value = cMaxStay.Value;
166       }
167 
168       private void cMaxStay_ValueChanged(object sender, EventArgs e)
169       {
170         if (cMaxStay.Value < cMinStay.Value)
171           cMaxStay.Value = cMinStay.Value;
172       }
173 
174       private void cMinGroupSize_ValueChanged(object sender, EventArgs e)
175       {
176         if (cMinGroupSize.Value > cMaxGroupSize.Value)
177           cMinGroupSize.Value = cMaxGroupSize.Value;
178       }
179 
180       private void cMaxGroupSize_ValueChanged(object sender, EventArgs e)
181       {
182         if (cMaxGroupSize.Value < cMinGroupSize.Value)
183           cMaxGroupSize.Value = cMinGroupSize.Value;
184       }
185 
186       private void MainForm_Resize(object sender, EventArgs e)
187       {
188         cRestaurant.Refresh();
189       }
190     }
191 }
```

RestaurantControl.cs

```
1 using System;
2 using System.Drawing;
3 using System.Drawing.Drawing2D;
4 using System.Windows.Forms;
5 
```

```
 6 namespace Turmrestaurant
 7 {
 8   /// <summary>
 9   /// Restaurant Control
10   /// Stellt ein Restaurant grafisch dar.
11   /// </summary>
12   internal sealed partial class RestaurantControl : Control
13   {
14     private Restaurant _restaurant;
15
16     public RestaurantControl()
17     {
18       InitializeComponent();
19       DoubleBuffered = true;
20     }
21
22     /// <summary>
23     /// Setz das Restaurant, welches dargestellt werden soll
24     /// </summary>
25     /// <param name="r"></param>
26     public void SetRestaurant(Restaurant r)
27     {
28       _restaurant = r;
29       Refresh();
30     }
31
32     protected override void OnPaint(PaintEventArgs pe)
33     {
34       base.OnPaint(pe);
35
36       Graphics g = pe.Graphics;
37       g.InterpolationMode = InterpolationMode.HighQualityBicubic;
38       g.SmoothingMode = SmoothingMode.HighQuality;
39
40       var outline = new Pen(Color.Black, 2f);
41
42       //1. Kreis (Rand & Tisch)
43       int cX = Width/2; //center X
44       int cY = Height/2; //center Y
45       int outerRadius = Math.Min(Width, Height)/2 - 4;
46       g.FillEllipse(Brushes.LightGray, cX - outerRadius, cY - outerRadius, 2*
             outerRadius, 2*outerRadius);
47       g.DrawEllipse(outline, Width/2 - outerRadius, Height/2 - outerRadius, 2*
             outerRadius, 2*outerRadius);
48
49       //2. Kreis (Sitzbereich)
50       int innerRadius = outerRadius*9/10;
51       g.FillEllipse(Brushes.White, cX - innerRadius, cY - innerRadius, 2*
             innerRadius, 2*innerRadius);
52
53       //Zentrum/Saeule
54       int innerRect = innerRadius*2/10;
55       g.DrawRectangle(outline, cX - innerRect, cY - innerRect, 2*innerRect, 2*
             innerRect);
56
57       if (_restaurant == null)
58         return; //We are done here...
59
60       //Sitzplaetze
61       int members = 0;
62       int seats = _restaurant.Seats;
63
64       float seatRadius = innerRadius*23f/10f/seats;
65       if (seats < 16) seatRadius *= (Math.Max(seats - 8, 0) + 8)/16f;
66
67       float seatMidDist = innerRadius - (seatRadius*16/10);
68       double teilWinkel = 2*Math.PI/seats;
69
70       var newArrival = new Pen(Color.LimeGreen, 3f);
```

```
71        var isLeaving = new Pen(Color.Red, 3f);
72
73        //0-Markierung
74        SizeF measure = g.MeasureString("0", SystemFonts.DefaultFont);
75        g.DrawString("0", SystemFonts.DefaultFont, Brushes.Black, cX - measure.
              Width/2,
76                    cY - (innerRadius + outerRadius)/2 - measure.Height/2 + 1);
77
78        for (int i = 0; i < _restaurant.Seats; i++)
79        {
80          double winkel = i*teilWinkel;
81          float seatx = cX + (float) (Math.Sin(winkel)*seatMidDist);
82          float seaty = cY - (float) (Math.Cos(winkel)*seatMidDist);
83
84          //Fuellung?
85          Brush b = Brushes.White;
86          Pen p = null;
87
88          Group seating = _restaurant.Seating[i];
89          //Besetzte Sitze entsprechend einfaerben
90          if (seating != null)
91          {
92            b = new SolidBrush(seating.Color);
93
94            //Rand gibt den Status an
95            if (seating.RemainingLengthOfStay <= 1)
96              p = isLeaving;
97            else if (seating.NewlyAdded)
98            {
99              p = newArrival;
100             //Benoetigt fuer den Streich (nutzt nicht das Tick-System)
101             if (seating.RemainingLengthOfStay == int.MaxValue)
102               seating.NewlyAdded = false;
103           }
104           members++;
105         }
106
107         var seatRect = new RectangleF(seatx - seatRadius, seaty - seatRadius, 2*
              seatRadius, 2*seatRadius);
108         g.FillEllipse(b, seatRect);
109
110         //Rand
111         if (p != null)
112           g.DrawEllipse(p, RectangleF.Inflate(seatRect, 2, 2));
113         g.DrawEllipse(Pens.Black, seatRect);
114       }
115
116       //Auslastung
117       g.DrawString("Auslastung: " + (members*100/seats) + "%" + Environment.
              NewLine + "Gaeste: " + members,
118                   SystemFonts.DefaultFont, Brushes.Black, 1, 1);
119     }
120   }
121 }
```

Utils.cs

```
1  using System;
2
3  namespace Turmrestaurant
4  {
5    /// <summary>
6    /// Helferklasse
7    /// </summary>
8    internal static class Utils
9    {
10     static Utils()
11     {
```

```
12          Random = new Random();
13        }
14
15        /// <summary>
16        /// RNG
17        /// </summary>
18        public static Random Random { get; private set; }
19      }
20  }
```

Group.cs

```
1   using System.Drawing;
2
3   namespace Turmrestaurant
4   {
5     /// <summary>
6     /// Gruppe aus Gaesten
7     /// </summary>
8     internal class Group
9     {
10      /// <summary>
11      /// Gruppe aus Gaesten
12      /// </summary>
13      /// <param name="members">Anzahl der Gaeste</param>
14      /// <param name="stayDuration">Aufenthaltsdauer</param>
15      public Group(int members, int stayDuration)
16      {
17        MemberCount = members;
18        RemainingLengthOfStay = stayDuration;
19        NewlyAdded = true;
20
21        //Zufaellige Farbe ohne Alpha generieren
22        Color = Color.FromArgb((int) (Utils.Random.Next() | 0xFF000000));
23      }
24
25      /// <summary>
26      /// Gruppe aus Gaesten (ohne bekannte Abreisezeit)
27      /// </summary>
28      /// <param name="members"></param>
29      public Group(int members) : this(members, int.MaxValue)
30      {
31      }
32
33      /// <summary>
34      /// Farbe der Gruppe
35      /// (nur zu anschaulichen Zwecken)
36      /// </summary>
37      public Color Color { get; private set; }
38
39      /// <summary>
40      /// Anzahl der Gruppenmitglieder
41      /// </summary>
42      public int MemberCount { get; private set; }
43
44      /// <summary>
45      /// Verbleibende Aufenthaltsdauer in Ticks
46      /// </summary>
47      public int RemainingLengthOfStay { get; set; }
48
49      /// <summary>
50      /// Neue Gaeste?
51      /// </summary>
52      public bool NewlyAdded { get; set; }
53    }
54  }
```

Restaurant.cs

```
1  using System.Collections.Generic;
2
3  namespace Turmrestaurant
4  {
5    /// <summary>
6    /// Restaurant
7    /// Stellt die Sitzlogik zur Verfuegung
8    /// </summary>
9    internal class Restaurant
10   {
11     #region Generelles Layout
12
13     #region Delegates
14
15     /// <summary>
16     /// Log Delegate
17     /// </summary>
18     /// <param name="msg"></param>
19     public delegate void LogDelegate(string msg);
20
21     #endregion
22
23     private readonly List<Group> _currentGroups = new List<Group>();
24     private readonly LogDelegate _log;
25
26     /// <summary>
27     /// Restaurant
28     /// </summary>
29     /// <param name="seats">Sitzplaetze</param>
30     /// <param name="log">Log-Ausgabe Methode</param>
31     /// <returns></returns>
32     public Restaurant(int seats, LogDelegate log)
33     {
34       _log = log;
35       Seats = seats;
36       Seating = new Group[seats];
37     }
38
39     /// <summary>
40     /// Seats (Sitzplaetze)
41     /// </summary>
42     public int Seats { get; private set; }
43
44     /// <summary>
45     /// Sitzverteilung
46     /// </summary>
47     public Group[] Seating { get; private set; }
48
49     /// <summary>
50     /// Gibt die im Restaurant vertretenen Gruppen zurueck
51     /// </summary>
52     public Group[] Groups
53     {
54       get { return _currentGroups.ToArray(); }
55     }
56
57     /// <summary>
58     /// Log
59     /// </summary>
60     /// <param name="msg"></param>
61     public void Log(string msg)
62     {
63       _log(msg);
64     }
65
66     #endregion
67
```

```
68      #region Ober
69
70      /// <summary>
71      /// Ruft die freien Sitzplaetze in Form verbundener Grenzen ab
72      /// </summary>
73      /// <returns></returns>
74      public IEnumerable<Bounding> GetFreeSeats()
75      {
76        //Nach "linker" Grenze suchen
77        //Da es sich hier um einen Kreis handelt, verknuepfen wir dem Bereich
78        //kleiner als 0 mit dem Ende des Arrays
79        int mostLeft = Seats - 1;
80        while (mostLeft >= 0 && Seating[mostLeft] == null)
81          mostLeft--;
82
83        if (mostLeft < 0) //Sonderfall: Alle Plaetze frei
84          return new[] {new Bounding {Left = 0, Size = Seats}};
85
86        //Grenze um einen nach rechts verschieben
87        mostLeft = (mostLeft + 1)%Seats;
88
89        //Gruppen bilden
90        var grps = new List<Bounding>();
91        var curBounding = new Bounding {Left = mostLeft};
92
93        int i = mostLeft;
94
95        //Durch den Kreis iterieren
96        do
97        {
98          if (Seating[i] == null)
99          {
100           if (curBounding.Left == int.MinValue)
101             curBounding.Left = i;
102           curBounding.Size++;
103         }
104         else if (curBounding.Size > 0)
105         {
106           //Ende der freien Sitzreihe
107           grps.Add(curBounding);
108           curBounding = new Bounding {Left = int.MinValue};
109         }
110
111         //Naechster Platz
112         i = (i + 1)%Seats;
113       } while (i != mostLeft);
114
115       if (curBounding.Size > 0)
116         grps.Add(curBounding);
117
118       return grps.ToArray();
119     }
120
121     /// <summary>
122     /// Sucht fuer die Gruppe die beste Platzierungsmoeglichkeit
123     /// </summary>
124     /// <param name="g"></param>
125     /// <returns></returns>
126     private Bounding GetBestSeating(Group g)
127     {
128       Bounding largest = null;
129       foreach (Bounding b in GetFreeSeats())
130       {
131         if (b.Size < g.MemberCount)
132           continue; //zu klein
133         if (b.Size == g.MemberCount)
134           return b; //ideal
135         if (largest == null || b.Size > largest.Size)
136           largest = b; //groeÃŸter Freiraum
```

```
137         }
138
139         //Falls keine Ideale Loesung gefunden wird, beste (ggf. auch null)
                zurueck geben
140         return largest;
141       }
142
143       /// <summary>
144       /// Versucht fuer die gegebene Gruppe einen Platz im Restaurant zu finden
145       /// </summary>
146       /// <param name="p">Zu hinzufuegende Gruppe</param>
147       /// <returns>Erfolg?</returns>
148       public bool AddGroup(Group p)
149       {
150         if (p.MemberCount == 0)
151           return true;
152
153         _log("Neue Gaeste sind angekommen! (Anzahl=" + p.MemberCount + ")");
154         Bounding dest = GetBestSeating(p);
155
156         if (dest == null)
157         {
158           _log("- Nicht genuegend freie Plaetze. [FEHLER]");
159           return false; //Kein Platz. Ober aergert sich.
160         }
161
162         //Sonst: Von Links an die Plaetze "bestuecken"
163         int cur = dest.Left;
164         for (int i = 0; i < p.MemberCount; i++)
165         {
166           Seating[cur] = p;
167           cur = (cur + 1)%Seats;
168         }
169         _log("- Sitzplaetze: " + dest.Left + " bis " + ((dest.Left + p.MemberCount
                - 1)%Seats) +
170             (dest.Size == p.MemberCount ? " [Luecke gefuellt]" : ""));
171         _currentGroups.Add(p);
172         return true;
173       }
174
175       /// <summary>
176       /// Entfernt die Gruppe aus dem Restaurant
177       /// </summary>
178       /// <param name="p">Zu entfernende Gruppe</param>
179       /// <returns>Erfolg?</returns>
180       public bool RemoveGroup(Group p)
181       {
182         if (!_currentGroups.Contains(p))
183           return false;
184
185         _log("Gaeste wollen uns verlassen! (Anzahl=" + p.MemberCount + ")");
186         _currentGroups.Remove(p);
187
188         string frei = "";
189         for (int i = 0; i < Seats; i++)
190         {
191           if (Seating[i] == p)
192           {
193             frei += i + ", ";
194             Seating[i] = null; //remove.
195           }
196         }
197         _log("- Sitzplaetze frei geworden: " + frei.Trim(' ', ','));
198
199         return true;
200       }
201
202       /// <summary>
203       /// Entfernt alle Gaeste aus dem Restaurant
```

```
204       /// </summary>
205       public void Clear()
206       {
207         foreach (Group g in Groups)
208           RemoveGroup(g);
209       }
210
211       /// <summary>
212       /// Grenze
213       /// </summary>
214       public class Bounding
215       {
216         /// <summary>
217         /// Linker Rand
218         /// </summary>
219         public int Left { get; set; }
220
221         /// <summary>
222         /// GroeÃŸe
223         /// </summary>
224         public int Size { get; set; }
225       }
226
227       #endregion
228     }
229 }
```

Simulation.cs

```
 1 namespace Turmrestaurant
 2 {
 3   /// <summary>
 4   /// Restaurant Simulation
 5   /// </summary>
 6   internal static class Simulation
 7   {
 8     /// <summary>
 9     /// Tick
10     /// </summary>
11     /// <param name="r">Restaurant</param>
12     /// <param name="form">MainForm</param>
13     public static bool Tick(Restaurant r, MainForm form)
14     {
15       //1. Pruefen, ob Gaeste das Restaurant verlassen
16       int members = 0;
17       foreach (Group g in r.Groups)
18       {
19         g.NewlyAdded = false;
20         g.RemainingLengthOfStay--;
21
22         if (g.RemainingLengthOfStay == 0)
23           r.RemoveGroup(g);
24         else members += g.MemberCount;
25       }
26
27       //2. Auslastung berechnen und ggf. neue Gruppen hinzufuegen
28       //(nur eine Gruppe/tick max.)
29       int usage = members*100/r.Seats;
30
31       if (usage < form.cUsage.Value)
32       {
33         //Zufaellige Werte bilden
34         int size = Utils.Random.Next((int) form.cMinGroupSize.Value,
35                                      (int) form.cMaxGroupSize.Value + 1); //
                                          exclusive rand
36
37         var ng = new Group(size, Utils.Random.Next((int) form.cMinStay.Value,
38                                                    (int) form.cMaxStay.Value));
```

```
39
40          //Neue Gruppe hinzufuegen
41          if (!r.AddGroup(ng))
42            return false; //Bei Fehler stoppen.
43        }
44        return true;
45      }
46    }
47 }
```

Streich.cs

```
1  using System;
2  using System.ComponentModel;
3  using System.Threading;
4
5  namespace Turmrestaurant
6  {
7    /// <summary>
8    /// Streich
9    /// </summary>
10   internal static class Streich
11   {
12     private static BackgroundWorker _worker;
13     private static volatile bool _waiting = true;
14     private static Restaurant _restaurant;
15     private static MainForm _form;
16
17     /// <summary>
18     /// Ist die Arbeit abgeschlossen?
19     /// </summary>
20     public static bool IsFinished
21     {
22       get { return _worker == null; }
23     }
24
25     /// <summary>
26     /// Streich beginnen
27     /// </summary>
28     public static void Start(Restaurant r, MainForm form)
29     {
30       if (_worker != null) Stop();
31
32       r.Clear();
33       _restaurant = r;
34       _form = form;
35
36       _worker = new BackgroundWorker();
37       _worker.DoWork += (o1, o2) => RunStreich();
38       _worker.WorkerSupportsCancellation = true;
39       _worker.RunWorkerAsync();
40     }
41
42     /// <summary>
43     /// Streich abbrechen
44     /// </summary>
45     public static void Stop()
46     {
47       if (_worker != null)
48         _worker.CancelAsync();
49     }
50
51     /// <summary>
52     /// Laesst einen Tick vergehen..
53     /// </summary>
54     public static void Tick()
55     {
56       _waiting = false;
```

```
57      }
58
59      /// <summary>
60      /// Wartet einen Tick
61      /// </summary>
62      private static void WaitTick()
63      {
64        _form.Invoke(new EventHandler((o1, o2) => _form.cRestaurant.Refresh()));
65        _waiting = true;
66        while (_waiting && !_worker.CancellationPending)
67          Thread.Sleep(1);
68      }
69
70      /// <summary>
71      /// Streich Logik
72      /// </summary>
73      private static void RunStreich()
74      {
75        Restaurant r = _restaurant;
76
77        var pupils = (int) _form.cSchueler.Value;
78        int fac = 1;
79
80        while (fac < r.Seats)
81          fac <<= 1; //Passenden Startwert suchen (Power of Two)
82
83        do
84        {
85          r.Log("Streich mit " + pupils + " Schuelern...!");
86          r.Log("".PadLeft(20, '='));
87
88          bool success = TryStreich(pupils);
89
90          r.Log("".PadLeft(20, '='));
91          r.Log("Erfolg: " + (success ? "Ja!" : "Nein..."));
92
93          if (success)
94            r.Log(pupils + " Schueler benoetigt!");
95
96          //a) Binaere Approximation (geht einfach schneller...)
97          if (_form.cBisection.Checked)
98          {
99            if (success)
100           {
101             if (fac == 0)
102             {
103               r.Log("Approximation vollstaendig.");
104               break; //We are done!
105             }
106             pupils -= fac;
107           }
108           else pupils += Math.Max(fac, 1);
109           fac /= 2;
110         }
111           //b) Normale Inkrementierung
112         else if (!success && _form.cIncrement.Checked)
113           pupils++;
114         else //Single Step oder Ziel erreicht
115           break; //Abbruch
116
117         //Anzeige Updaten
118         pupils = Math.Max(pupils, 1);
119         int pupilsSecure = pupils; //Verhindert access-to-modified-closure
120         _form.Invoke(new EventHandler((o1, o2) => _form.cSchueler.Value =
              pupilsSecure));
121
122         for (int i = 0; i < 3; i++)
123           WaitTick();
124         _restaurant.Clear();
```

```
125           _form.ClearLog();
126         } while (!_worker.CancellationPending);
127
128         _worker = null;
129       }
130
131       /// <summary>
132       /// Versucht einen Angriff mit der angegebenen Anzahl an Schuelern
133       /// </summary>
134       /// <param name="pupils"></param>
135       private static bool TryStreich(int pupils)
136       {
137         Restaurant r = _restaurant;
138
139         //Startmarkierung setzen
140         r.Log("1. Startmarkierung setzen");
141         r.AddGroup(new Group(1));
142         WaitTick();
143         pupils--;
144
145         //Äœberflutungs-/Fluch-Gruppe schon aufsparen; verbleibende Schueler aus
                dem
146         //Fragmentierungsschritt kommen hinzu. Wir nehmen etwas weniger als die
                Haelfte
147         //(-2 hat sich als am besten passend herausgestellt) um bereits mit dem
                jeweiligen
148         //Fragmentierungsrest das Limit zu uebertreffen. So koennen wir schon
                einen Schritt
149         //vorher aufhoeren und brauchen weniger Schueler
150         int uf = (int) Math.Ceiling(pupils/2f) - 2;
151         r.Log("".PadLeft(30, '-'));
152         r.Log("2. Wir heben uns " + uf + " Schueler fuer das Ende auf.");
153         pupils -= uf;
154
155         //Mit den verbleibenden Schuelern jeweils fuer eine Fragmentierung
                sorgen
156         r.Log("".PadLeft(30, '-'));
157         r.Log("3. Fragmentierung verursachen ...");
158         while (pupils > 0 && !_worker.CancellationPending)
159         {
160           r.Log("".PadLeft(30, '.'));
161
162           //Pruefen, ob sich ein "Äœberfluten" lohnt
163           int max = -1;
164           foreach (Restaurant.Bounding b in r.GetFreeSeats())
165           {
166             if (b.Size > max)
167               max = b.Size;
168           }
169
170           if (max < pupils + uf) //..der wird sich aergern!
171             break; //Zeit zum Stuermen!
172
173           //Temp Gruppe
174           var temp = new Group(pupils - 1);
175           if (!r.AddGroup(temp)) return true;
176           WaitTick();
177
178           //Permanenter Marker
179           if (!r.AddGroup(new Group(1))) return true;
180           WaitTick();
181           pupils--;
182
183           //Temp Gruppe entfernen
184           r.RemoveGroup(temp);
185           WaitTick();
186         }
187
188         r.Log("".PadLeft(30, '-'));
```

```
189         r.Log("4. Mit allen Schuelern 'angreifen'!");
190         bool result = !r.AddGroup(new Group(pupils + uf));
191         WaitTick();
192         return result;
193       }
194     }
195 }
```

D. Anhang

In diesem Teil des Dokumentes befinden sich einige weitere Informationen und Erklärungen zu den einzelnen Aufgaben sowie der generellen Bewerkstelligung der Arbeit.

Abbildungsverzeichnis

Verwendete Programme

- Microsoft Visual Studio 2008 C# Express Edition (Entwicklungsumgebung)
- Inkscape Vector Graphics Editor (Diagramme)
- GIMP - GNU Image Manipulation Program (Grafik)
- Texmaker (Latex Entwicklungsumgebung)

Ordnerstruktur der CD

Die CD besteht aus drei einzelnen Ordnern:
/U-Code/.. - 1. Aufgabe (Universeller Öffnungscode)
/Turmrestaurant/.. - 2. Aufgabe (Turmrestaurant)
/tex/.. - Dokumentation (.pdf und .tex)

In den Verzeichnissen der einzelnen Aufgaben finden sich jeweils drei Unterordner:
./Bin/: Enthält die ausführbaren Dateien
./Source/: Enthält den Quelltext
./Material/: Enthält Screenshots und Ablaufsprotokolle zum jeweiligen Programm

Einverständniserklärung

Ich erkläre, dass ich diese Arbeit (Dokumentation und Quelltexte) ohne fremde Hilfe angefertigt und nur die angegebenen Hilfsmittel benutzt habe.

Duisburg, 7. April 2010 _____