28. Bundeswettbewerb Informatik, 2. Runde

Lösungshinweise und Bewertungskriterien

Allgemeines

Es ist immer wieder bewundernswert, wie viele Ideen, wie viel Wissen, Fleiß und Durchhaltevermögen in den Einsendungen zur zweiten Runde eines Bundeswettbewerbs Informatik stecken. Um aber die Allerbesten für die Endrunde zu bestimmen, müssen wir die Arbeiten kritisch begutachten und hohe Anforderungen stellen. Von daher sind Punktabzüge die Regel und Bewertungen über die Erwartungen (5 Punkte) hinaus die Ausnahme. Lassen Sie sich davon nicht entmutigen! Wie auch immer Ihre Einsendung bewertet wurde: Allein durch die Arbeit an den Aufgaben und den Einsendungen hat jede Teilnehmerin und jeder Teilnehmer einiges dazu gelernt; den Wert dieses Effektes sollten Sie nicht unterschätzen.

Bevor Sie sich in die Lösungshinweise vertiefen, lesen Sie doch bitte kurz die folgenden Anmerkungen zu Einsendungen und den beiliegenden Unterlagen durch.

Terminprobleme Einige Einsender gestehen ganz offen, dass Ihnen die Zeit zum Einsendeschluss hin knapp geworden ist, worauf wir bei der Bewertung leider keine Rücksicht nehmen können. Abiturienten macht der Terminkonflikt mit der Abiturvorbereitung Probleme. Der ist für eine erfolgreiche Teilnahme sicher nicht ideal. In der zweiten Jahreshälfte läuft aber die zweite Runde des Mathewettbewerbs, dem wir keine Konkurrenz machen wollen. Also bleibt uns nur die erste Jahreshälfte. Aber: Sie hatten etwa vier Monate Bearbeitungszeit für die zweite BWINF-Runde. Rechtzeitig mit der Bearbeitung der Aufgaben zu beginnen war der beste Weg, Konflikte mit dem Abitur zu vermeiden.

Dokumentation Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und die Umsetzung in Software fließen lassen. Doch ohne eine gute Beschreibung der Lösungsideen, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine gute Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (die die verschiedenen bei der Lösung des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung wenig wert. Bewerterinnen und Bewerter können die Qualität Ihrer Einsendung nur anhand dieser Informationen vernünftig einschätzen. Mängel können nur selten durch gründliches Testen der eingesandten Programme ausgeglichen werden – wenn diese denn überhaupt ausgeführt werden können: Hier gibt es häufig Probleme, die meist vermieden werden könnten, wenn Lösungsprogramme vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner getestet würden. Insgesamt sollte die Erstellung des schriftlichen Materials die Programmierarbeit begleiten oder ihr teilweise sogar vorangehen: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt keine saubere Umsetzung in welche Programmiersprache auch immer hin.

Bewertungsbögen Kein Kreuz in einer Zeile bedeutet, dass die genannte Anforderung den Erwartungen entsprechend erfüllt wurde. Vermerkt wird also in der Regel nur, wenn davon abgewichen wurde – nach oben oder nach unten. Ein Kreuz in der Spalte „+“ bedeutet Zusatzpunkte, ein Kreuz unter „–“ bedeutet Minuspunkte für Fehlendes oder Unzulängliches. Dabei haben die Marken nicht immer den gleichen Einfluss auf die Gesamtbewertung, sondern können je nach Einsendung unterschiedlich gewichtig sein. Die Schattierung eines Feldes bedeutet, dass die entsprechenden Zusatz- bzw. Minuspunkte in der Regel nicht vergeben wurden.

Lösungshinweise Bei den folgenden Erläuterungen handelt es sich um Vorschläge, nicht um die einzigen Lösungswege, die wir gelten ließen. Wir akzeptieren in der Regel alle Ansätze, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind. Einige Dinge gibt es allerdings, die – unabhängig vom gewählten Lösungsweg – auf jeden Fall diskutiert werden müssen. Zu jeder Aufgabe gibt es deshalb einen Abschnitt, indem gesagt wird, worauf bei der Bewertung letztlich geachtet wurde, zusätzlich zu den grundlegenden Anforderungen an Dokumentation (insbesondere: klare Beschreibung der Lösungsidee, genügend aussagekräftige Beispiele) und Quellcode (insbesondere: Strukturierung und Kommentierung, gute Übersicht durch Programm-Dokumentation).

Aufgabe 1: Universeller Öffnungscode

In dieser Aufgabe versuchen wir uns als Angreifer eines Verfahrens zur Authentisierung. Wir möchten eine möglichst kurze Reihe von Schlüsseln erzeugen, die uns garantiert Zugang zu einem gesicherten technischen Gerät beschaffen, da mindestens einer von ihnen die korrekte Stellung der aktiven Schalter enthält.

Doch schon bei den ersten Versuchen, einen solchen universellen Öffnungscode zu charakterisieren, fällt uns auf, wie sehr die Schlüssel in unserem Code voneinander abhängen. Es erweist sich als schwierig, den Einfluss zu beschreiben, den das Ändern von Bits in einem Schlüssel unseres Codes hat, da er plötzlich ganz neue „Überdeckungen" liefert, während viele alte wegfallen.

Eine effiziente, direkte Vorschrift zum Erstellen eines optimalen universellen Öffnungscodes erscheint deshalb als sehr unwahrscheinlich. Im folgenden werden wir deswegen Verfahren sehen, die versuchen, in kurzer Zeit möglichst kurze – aber nicht immer optimale – Öffnungscodes zu erzeugen.

1.1 Verifizierung eines Codes

Das Herzstück unserer Verfahren ist es, festzustellen, ob eine gegebene Menge an Schlüsseln der Länge N einen universellen Öffnungscode darstellt. Neben der Möglichkeit zu testen, ob eine von uns erzeugte Lösung tatsächlich einen universellen Öffnungscode darstellt, eröffnen wir uns gleichzeitig eine Reihe von Verfahren zur Lösung des Problems: Nach einem bestimmten Prinzip können wir eine Schlüsselmenge Stück für Stück aufbauen, bis eine gültige Lösung gefunden wurde.

Wie kann dieses Verfahren effizient implementiert werden? Wir müssen für jede mögliche Auswahl von M aktiven Schaltern aus allen N Schaltern jede Belegung aus Nullen und Einsen in der gegebenen Schlüsselmenge wiederfinden. Eine naheliegende Lösung ist es, alle $\binom{N}{M}$ viele Auswahlen aktiver Schalter zu genererieren. Für jede dieser betrachten wir alle 2^M vielen Bitstrings der Länge M und gehen die Liste der Schlüssel durch, ob mindestens einer von ihnen genau die richtige Belegung an den aktiven Schaltern hat. Sollte es für eine Schalterauswahl eine Belegung geben, die von keinem Schlüssel abgedeckt wird, so ist die gegebene Menge kein universeller Öffnungscode, andernfalls sind wir zufrieden. Haben wir l Schlüssel gegeben, so können wir also in Zeit $O(\binom{N}{M}2^M l)$ testen, ob ein universeller Öffnungscode vorliegt.

Während wir uns noch darüber freuen, diese einfache Methode gefunden zu haben, melden sich schon die ersten Zweifel an der Geschwindigkeit dieser Lösung: Wird unsere Schlüsselmenge nicht zu oft durchsucht? Schließlich sollte für eine gegebene Auswahl aktiver Schalter die Menge an Bitstrings, die die betreffenden M Schalter der Schlüsselmenge darstellen, genau alle Bitstrings der Länge M enthalten. Reicht also ein Durchlauf pro Schalterauswahl, womit wir eine Laufzeit von $O(\binom{N}{M}l)$ erhalten würden?

Die Antwort lautet erfreulicherweise Ja. In einem 2^M-elementigen Array speichern wir, ob jeder M-Bitstring überdeckt wird. Für eine gegebene Schalterauswahl gehen wir also die Liste an Schlüsseln durch und bilden für jeden Schlüssel den zugehörigen M-Bitstring (also die Belegung an den aktiven Schaltern). Dieser stellt offensichtlich eine Zahl zwischen 0 und $2^M - 1$ dar, die wir als Index für das Array benutzen können und den entsprechenden Eintrag als „überdeckt" markieren können. Nachdem wie alle Schlüssel betrachtet haben, untersuchen wir, ob alle Einträge in dem Array „überdeckt" sind. Wenn ja, so ist für die gegebene Schalterauswahl die Bedingung noch erfüllt, wenn nein, so liegt kein universeller Öffnungscode vor. Wir können also einen Faktor 2^M einsparen.

1.2 Lösungsideen

Wie oben bereits angesprochen, bietet sich folgendes Verfahren an, um einen universellen Öffnungscode C zu erstellen:

1. $C := \emptyset$
2. Erzeuge k viele neue zufällige N-Bitstrings und füge sie zu C hinzu.
3. Überprüfe, ob C ein universeller Öffnungscode ist. Wenn nein, gehe zu 2.

Es wäre natürlich schön, wenn k zusätzlich automatisch an die Größe des Problems angepasst wird. Dazu kann man einfache Schranken für die Länge des Codes nutzen, oder man wiederholt dieses Vefahren mit immer kleinerem k, wobei bisherige Ergebnisse wiederbenutzt werden.

In der folgenden Tabelle sind die Ergebnisse aufgelistet, die dieser naive Ansatz auf den drei in der Aufgabenstellung genannten Eingaben liefert. Dabei handelt es sich (wie bei allen weiteren Tabellen) um das beste Ergebnis mehrerer Läufe des Algorithmus, die zusammen nicht mehr als einige Minuten Laufzeit zur Verfügung hatten.

N	M	Länge des Codes
9	3	29
18	4	129
25	5	426

Tabelle 1: Ergebnisse des naiven Ansatzes

Es dürfte offensichtlich sein, dass dieser sehr einfache Ansatz einige größere Schwächen offenbart: Es kann passieren, dass Schlüssel hinzugefügt werden, die sich kaum von den bisherigen Schlüsseln unterscheiden, also keine bzw. wenige zusätzlichen Überdeckungen liefern. Dieser naive Ansatz stellt aber eine solide Grundlage da, die in geringer Zeit vertretbare Ergebnisse erzielt.

Löschen redundanter Schlüssel

Um das Problem der überflüssigen Schlüssel zu beheben, kann zunächst ein gültiger universeller Öffnungscode erzeugt werden, um danach sukzessive jeden Schlüssel zu löschen, ohne den der Code immer noch die Voraussetzungen erfüllt.

Dieses Vorgehen ist im Allgemeinen nicht optimal: die Reihenfolge, in der wir redundante Schlüssel löschen, kann das Ergebnis beeinflussen. Das Verfahren lässt also noch viel Platz für Verbesserungen. Doch bereits mit dieser einfachen Vorgehensweise können wir das Ergebnis sichtlich verbessern: In unseren Beispielen verkleinern wir die Codes auf etwas über 60 % ihrer ursprünglichen Größe.

N	M	Länge des Codes
9	3	18
18	4	83
25	5	259

Tabelle 2: Ergebnisse nach Löschen redundanter Schlüssel

Heuristische Auswahl neuer Schlüssel

Der oben besprochene Ansatz zur Vermeidung redundanter Schlüssel hat einen eher „destruktiven“ Charakter. *Nach* Erzeugung einer gültigen Schlüsselmenge werden überflüssige Schlüssel gelöscht. Ein weiterer Ansatz ließe sich als „konstruktiv“ bezeichnen: Bereits beim Aufbau des Codes versuchen wir nur besonders effektive Schlüssel zu generieren.

Dazu haben wir zwei Ansätze verfolgt: Zum Einen über die Auswahl eines besonders geeigneten Schlüssels aus einer Menge zufällig erzeugter, zum Anderen über einen Bit-für-Bit-Aufbau eines neuen Schlüssels.

Im Folgenden nennen wir die Kombination aus einer Auswahl von M aktiven Schaltern mit ihrer zugehörigen Schalterstellung *Zugangscode*. Für beide Varianten des heuristischen Codeaufbaus werden wir eine Liste Z noch nicht überdeckter Zugangscodes verwalten, die am Anfang mit allen möglichen Zugangscodes initialisiert wird.

Bei der ersten Variante werden ähnlich zur naiven Methode zunächst k Schlüssel erstellt. Aus diesen Schlüsseln wählen wir denjenigen aus, der die meisten noch nicht überdeckten Zugangscodes überdeckt, und löschen die entsprechenden Zugangscodes dann aus der Liste Z.

Für die zweite Variante ist die Bemerkung wichtig, dass wir mit jedem Bit, für das wir eine Belegung wählen, die Anzahl an Zugangscodes, die wir mit dem neuen Schlüssel potentiell noch überdecken können, verringern. Es ergibt also Sinn, sich an jeder Stelle die meisten Möglichkeiten offen zu lassen. Dies kann man umsetzen, indem eine Liste L potentiell noch überdeckbarer Zugangscodes verwaltet wird. Wir initialisieren L mit der Liste insgesamt noch

unüberdeckter Zugangscodes Z. Für das erste Bit wählen wir die Belegung, die die meisten Zugangscodes mit aktivem ersten Schalter in L haben. Alle Zugangscodes mit aktivem ersten Schalter und genau der anderen Belegung werden aus L entfernt. Analog dazu wählt man für jedes weitere Bit immer die Belegungen, die die meisten Zugangscodes in L haben. Ist ein Zugangscode aus L vollständig überdeckt, so wird er aus beiden Listen gelöscht. Haben wir dann alle Bits belegt, so wird der neue Schlüssel zum bestehenden Code hinzugefügt.

N	M	Länge des Codes
9	3	12
18	4	54
25	5	182

Tabelle 3: Ergebnisse der heuristischen Auswahl

Bestehende Öffnungscodes lokal verbessern

Bisher haben wir uns auf Verbesserungen beschränkt, die immer nur einen Schlüssel betrachten: entweder bei der Konstruktion eines neuen Schlüssels oder beim Löschen redundanter Schlüssel. Ein neuer, viel versprechender Ansatz ist es zu fragen, ob und in welchen Fällen man zwei Schlüssel zu einem besseren Schlüssel zusammenfügen kann, sodass der Code weiterhin ein universeller Öffnungscode bleibt.

Betrachtet man zum Beispiel für jeden Schlüssel, welche Zugangscodes allein von ihm überdeckt werden, so weiß man, für welche Zugangscodes nur er „verantwortlich“ ist. Die entsprechenden Bits der aktiven Schalter dieser Zugangcodes bezeichnen wir als *tragende Bits*. Nun ist klar: stimmen zwei Schlüssel auf der Schnittmenge ihrer tragenden Bits überein, so können wir sie zusammenführen, indem wir alle tragenden Bits aus den bisherigen Schlüsseln übernehmen und alle anderen Bits beliebig wählen. Damit sparen wir einen Schlüssel ein.

Um möglichst viele Schlüssel zusammenführen zu können, starten wir eine zufällige lokale Suche. Dazu nehmen wir uns einen beliebigen Schlüssel und setzen alle seine nicht-tragenden Bits zufällig neu. Dies wiederholen wir eine Zeit lang und führen immer, wenn es möglich ist, Schlüssel zusammen. Die genaue Implementierung bedarf weiterer Überlegung, um ausreichend schnell zu sein. Der Aufwand lohnt sich allerdings, wie die Ergebnistabelle zeigt.

N	M	Länge des Codes
9	3	12
18	4	47
25	5	174

Tabelle 4: Ergebnisse nach lokalen Verbesserungen

Constraint Satisfaction Problem

Zu testen bleibt, ob man das Problem des Findens eines Öffnungscodes mit Standardmethoden (wie einem Integer Program Solver oder Constraint Satisfaction Problem Solver) optimal lösen kann – in akzeptabler Zeit. Zum Beispiel kann das Problem, ob ein Öffnungscode der Länge l existiert, leicht als Constraint Satisfaction Problem modelliert werden. Dazu führen wir $l \cdot N$ boolsche Variablen ein, die die Bits der l Schlüssel repräsentieren, und schreiben die Bedingungen an einen Öffnungscode in der Sprache des Constraint Satifaction Solvers hin.

Für den Fall $N = 9, M = 3$ erhält man auf diese Art sogar ein erfreuliches Ergebnis: Die von den optimierten Methoden erzeugten Öffnungscodes der Länge 12 sind optimal! Für größere Probleminstanzen steigen aber die Anzahl an Bedingungen und Variablen sehr schnell an, was unseren Constraint Satisfaction Solver in die Knie gingen ließ. Solche Standardmethoden helfen beim Öffnungscodeproblem offenbar nicht.

Weitere Methoden

Die hier vorgestellten Lösungsmöglichkeiten erheben keinesfalls Anspruch auf Vollständigkeit. Weitere Heuristiken sind vorstellbar, auch grundsätzlich andere Vorgehensweisen: Anstelle Schlüssel um Schlüssel zu einem Code hinzuzufügen, bis er die Bedingungen erfüllt, könnte man auch mit einer festgesetzten Anzahl Schlüssel starten und diese Bit für Bit simultan aufbauen.

Auch Hybridmethoden, die mehrere Lösungsansätze kombinieren, sind sehr gut vorstellbar. Dabei tritt besonders die Notwendigkeit zu Tage, mit verschiedenen Ansätzen zu experimentieren, da der Effekt und die Qualität der Heuristiken sehr schwer zu beurteilen ist.

1.3 Bewertungskriterien

- Es muss ein gültiger universeller Öffnungscode erzeugt werden. Dazu ist entweder ein korrektes Prüfverfahren nötig, oder es muss begründet werden, warum das realisierte Verfahren nur gültige Codes erzeugt.
- Der Test, ob ein Code gültig ist, darf nicht zu langsam sein, d.h. $\Omega\left(\binom{N}{M}2^M l\right)$. Insgesamt sind gute Ergebnisse wichtiger als schnelle Laufzeit; allerdings sollten die gewählten Verfahren die Berechnung des geforderten Codes (25, 5) nicht verhindern.
- Es sollte ein heuristisches Verfahren angewandt werden; eine naive Lösung (ob per Zufall, vollständiger Suche etc.) löst die Aufgabe nicht vollständig. Leichte Optimierungen sind aber möglich und mildern die Bewertung; etwa wenn der Parameter k der naiven Lösung automatisch an die Größe des Problem angepasst wird.
- Eine gewählte Heuristik muss sinnvoll sein – also die Ergebnisse verbessern – und diskutiert werden: Warum sorgt sie für einen kurzen Code?

- Es gibt Minuspunkte, wenn für die größte Beispielinstanz mehr als 250 Schlüssel benötigt werden. Es gibt Zusatzpunkte, wenn für die größte Beispielinstanz weniger als 180 Schlüssel benötigt werden.
- Der Trade-Off zwischen Qualität/Optimalität und Zeitbedarf sollte angesprochen werden; Komplexitätsüberlegungen sind also Pflicht. Hierzu sollten die wichtigsten Größen (2^M verschiedene Stellungen der aktiven Schalter, $\binom{N}{M}$ mögliche Positionen der aktiven Schalter) grundsätzlich erkannt worden sein.
- Für alle drei in der Aufgabenstellung genannten Eingaben müssen Ergebnisse dokumentiert werden; dies wurde in der Regel auch ohne explizite Aufforderung so gemacht.

Aufgabe 2: Das Turmrestaurant

2.1 Der Ober

Platzierungsstrategien spielen im (theoretischen) Idealfall für den Ober gar keine Rolle: nämlich wenn immer alle freien Plätze am Stück vorhanden sind. Dies trifft auf das leere Restaurant zu, und so kann sich die Platzierung daran orientieren, diesen Zustand möglichst zu erhalten.

Es ist also naheliegend, neu ankommende Gruppen immer so zu platzieren, dass danach möglichst große Lücken bleiben – dann ist die Wahrscheinlichkeit höher, dass spätere große Gruppen noch platziert werden können. Der offensichtliche Algorithmus dazu ist ein Greedy-Algorithmus, der neue Gruppen immer in die kleinste Lücke reinsetzt, die grade groß genug ist. So bleiben die größeren Lücken für Eventualitäten freigehalten.

Die obige Strategie ist aber nicht in allen Fällen ideal. Seien noch eine Lücke der Größe 4 und eine der Größe 6 frei. Der Reihe nach kommen nun Gruppen der Größen 3, 3 und 4 an (ohne dass eine andere Gruppe geht). In diesem Fall können nur dann alle Gruppen platziert werden, wenn gegen das obige Prinzip verstoßen wird; nämlich wenn die erste (3er-) Gruppe nicht in der am besten passenden (4er-) Lücke, sondern in der 6er-Lücke platziert wird. Für dieses Beispiel ist es also besser, bei der Platzierung die Lücken möglichst gleich groß zu halten.

Für die Platzierung spielt also nicht nur die aktuelle Situation eine Rolle, sondern auch mögliche zukünftige Situationen, und insbesondere das Freiwerden von Plätzen – denn dies erst reißt unschöne und möglicherweise schwierig zu besetzende Lücken auf und zerstört die ideale Situation.

Ein wenig vorausschauend kann der Ober zum Beispiel dann handeln, wenn eine neu ankommende Gruppe in der gewählten Lücke ausgerichtet wird. Selbstverständlich wird eine neue Gruppe direkt neben eine schon vorhandene platziert. Außerdem kann man zum Beispiel danach gehen, wie groß die an die Lücke angrenzenden Gruppen sind, oder wie viele Gruppen darauf bis zur nächsten Lücke folgen, damit die Wahrscheinlichkeit zur Entstehung größerer Lücken erhöht wird, wenn eine Gruppe das Restaurant verlässt. Interessant ist, auch die (bekannte oder vermutete) Anwesenheitszeit der vorhandenen Gruppen zu berücksichtigen. Setzt man eine neue Gruppe neben eine andere, die sich erst möglichst kurze Zeit im Restaurant aufhält, wird es wahrscheinlicher, dass beide Gruppen zur (annähernd) gleichen Zeit wieder gehen und dann eine große Lücke entsteht.

Die mögliche zukünftige Entwicklung im Restaurant kann nicht nur anhand derartiger „einfacher" Plausibilitätsüberlegungen, sondern auch systematisch berücksichtigt werden. Platzierungsanforderungen und Platzierungen sind wie Züge zweier Gegner in einem Spiel. Die Entscheidung über eine Platzierung kann also auch auf der Grundlage einer Spielbaumanalyse erfolgen. Dabei ist der eigene Zug eine mögliche Platzierung, während ein gegnerischer Zug aus dem Weggang einer (möglicherweise leeren) Menge von Gruppen und dem Eintreffen einer neuen Gruppe besteht. Der Spielbaum verzweigt stark, aber einige Schritte lassen sich

durchaus vorberechnen. Spielsituationen können dann z. B. danach bewertet werden, wie nahe sie dem Idealzustand sind, also wie viele Plätze maximal am Stück vorliegen – womit wir dann doch wieder beim Erhalt möglichst großer Lücken wären. Aber auch andere, komplexere Bewertungen sind möglich.

2.2 Die Schüler

Eine einfache Möglichkeit bei einem weniger schlauen Ober ist die folgende: Schicke nacheinander 1 Schüler, $N/2-1$ Schüler und wieder 1 Schüler. Wenn der Ober die Schüler nun immer in der gleichen Richtung aneinander gesetzt hat, befindet sich die große Gruppe in der Mitte (am Beispiel $N = 16$): `1222222230000000`. Ziehe nun die große Gruppe ab – damit gibt es zwei Lücken der Größe $N/2-1$ – und komme mit $N/2$ Schülern wieder: der Ober ärgert sich. Dazu werden $N/2+2$ Schüler benötigt. Ist aber der Ober etwas schlauer und richtet sich vorausschauend danach, die Wahrscheinlichkeit für möglichst viele freie Plätze am Stück hoch zu halten, dürfte die folgende Belegung entstehen: `3122222220000000`. Damit bringt das Abziehen der großen Gruppe nichts mehr. Alternativ kann die große Gruppe in Einergruppen aufgeteilt werden, die erst nacheinander kommen und dann, bis auf zwei sich gegenüber sitzende Schüler, gehen. Dann hat der Ober gegen $N/2+2$ (bzw. $\lceil N/2 \rceil + 2$, bei ungeraden N müssen drei Schüler sitzen bleiben) keine Chance.

Eine andere Idee ist, die Schülergruppen erst ganz klein zu machen und dann die Größe langsam zu erhöhen. Dann werden die Gruppen immer so entfernt, dass in die entstehenden Lücken eine Gruppe der nächsten Größe grade nicht mehr passt. So wird eine recht große Fragmentierung erreicht, da der Ober immer größere Lücken nutzen muss, um die Gruppen unterzubringen. Bei diesem Verfahren ist nicht von vorne herein klar, wie viele Schüler benötigt werden, um den Ober zu ärgern. Eine Lösung dieses Problems ist, die „minimale Ärger-Anzahl" systematisch zu bestimmen, etwa mit einer binären Suche und immer wieder neuen Durchläufen der Simulation (Teil 3). Die Lösungen einiger Teilnehmer zeigen, dass die Zahl nicht linear von N abhängt, auch bei einem auf einfache Weise agierenden Ober.

Ein konkretes Beispiel: Ein „einfacher" Ober eines Restaurants mit 16 Plätzen kann nach obigem Prinzip von 9 Schülern geärgert werden (_ bezeichne einen freien Platz, die Zahlen sind die Nummern der Gruppen, rechts wird die Anzahl der freien Schüler angegeben):

```
_  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _     9S
1  2  3  4  5  6  7  8  9  _  _  _  _  _  _  _     0S
1  _  3  _  5  _  7  _  9  _  _  _  _  _  _  _     4S
1  _  3  _  5  _  7  _  9 10 10 11 11  _  _  _     0S
1  _  _  _  5  _  _  _  9  _  _ 11 11  _  _  _     4S!
```

Das muss aber nicht funktionieren: Wenn der Ober Gruppe 11 anders platziert (bei leicht anderer Strategie), werden doch wieder mehr Schüler benötigt, denn dann können nicht 4 Schüler so abgezogen werden, dass maximal 3er-Lücken bleiben:

```
1  _  3  _  5  _  7  _  9 10 10  _  _  _ 11 11    OS
```

Auch die Schüler können natürlich aufwändiger agieren und sich um Vorausschau auf mögliche Reaktionen des Obers bemühen. Es ist aber jedenfalls in Ordnung, dass sie die Vorgehensweise des Obers kennen bzw. eine bestimmte vernünftige Vorgehensweise annehmen.

2.3 Das Duell

In Teil 3 geht es darum, den Wechsel von Anforderungen und Platzierungen mit den in den Teilen 1 und 2 entwickelten Strategien zu simulieren. Es genügt, sich auf das Wechselspiel zwischen Ober und Schülern zu konzentrieren. Komplizierter wird es, wenn die Platzanforderungen der Schüler sich mit Platzanforderungen weiterer Gäste mischen.

Interessant ist es, verschiedene Strategien durchzuspielen, zumindest auf der Seite des Obers.

2.4 Bewertungskriterien

Teil 1:

- Die Schwächen eines ganz einfachen Vorgehens sollen erkannt worden sein.
- Die Strategie des Obers sollte nicht ganz einfach sein (also etwa nur die am besten passende Lücke wählen), sondern zumindest an Hand plausibler Überlegungen auch vorausschauend agieren. Noch besser ist, wenn der Ober systematisch vorausschauend handelt oder auf andere Weise sich intelligent verhält (etwa durch Berücksichtigung statistischer Verhältnisse oder Entwicklungen).
- Insgesamt sollen verschiedene Strategien zumindest angesprochen werden.
- Es ist nicht erforderlich, dass der Ober besonders realitätsnahe Kriterien verwendet. Die Aufgabe ist auch so interessant genug.
- Auch für den Ober alleine sollte es Beispiele bzw. Probeläufe geben. Um die Vorgehensweise des Obers beurteilen zu können, sollten die Beispiele nicht nur manuell, sondern systematisch erzeugt sein. Hierzu ist dann anzugeben, wie und unter welchen Bedingungen die Platzanforderungen generiert werden und wie die Frage der Aufenthaltszeit geregelt ist (wobei auch eine manuelle Regelung – wenn das Kommen und Gehen der Gruppen vom Benutzer gesteuert werden muss – akzeptabel ist).

Teil 2:

- In vielen Einsendungen wird für die Schüler nur die oben erwähnte „Killerstrategie“ realisiert und dazu von einem einfachen Ober ausgegangen. Andere mögliche Strategien sollten wenigstens besprochen werden.

- Die möglichen realen Bedingungen des Turmrestaurants müssen nicht berücksichtigt sein. Es findet sich vermutlich für jede Strategie eine Erklärung, wie die Schüler sie auch unter angenommenen realen Bedingungen umsetzen können – es gibt immerhin Mobiltelefone.

Teil 3:

- Die gesuchte „minimale Ärger-Anzahl“ hängt (u.a.) stark von den Strategien von Ober und Schüler ab. Es sollte erkannt worden sein, dass die eigenen Aussagen dazu nicht unbedingt allgemeingültig sind. Gegen $\lceil N/2 \rceil + 2$ Schüler kann der Ober wohl wirklich nichts anrichten. Es kann aber auch mit weniger gehen – je nach Strategie.
- Es sollten mehrere Probeläufe mit unterschiedlichen Restaurantgrößen dokumentiert sein, die korrekt und nachvollziehbar sind.

Allgemein können bei dieser Aufgabe auch mathematische Analysen sinnvoll sein. Hilfreiche und korrekte mathematische Betrachtungen sind deshalb Zusatzpunkte wert.

Aufgabe 3: Anagramme

3.1 Textquellen

Englische Texte gibt es im Internet in verschiedensten Größenordnungen. Als Beispiele seien an dieser Stelle zu nennen:

E-Books: Suchbegriffe wie „free ebooks“ liefern sowohl zahlreiche Linksammlungen, die häufig in Universitätsarchive führen, als auch Online-Portale, u.a. www.gutenberg.org. Aus zahlreichen PDF-Dateien lassen sich z.B. mit Adobe Reader ebenfalls brauchbare Textdokumente exportieren.

Wikipedia und vergleichbare Wissenssammlungen: Eher langsam und wenig rücksichtsvoll ist das systematische Herunterladen von Artikeln mittels (eigenem) Crawler, denn Wikimedia-basierte Websites verfügen für diesen Zweck über eine Query-API, die reinen Text zurückgeben kann. Die Server static.wikipedia.org und download.wikipedia.org bieten auch Komplettdownloads als HTML (14 GB) oder XML/Wikitext (6 GB).

3.2 Textaufbereitung

Große und kleine Datenmengen

Je größer die Datenmenge, desto wichtiger ist die Kunst des Weglassens. Natürlich kann man die heruntergeladene Wikipedia entpacken (HTML: 250 GB, XML: 25 GB), sinnvoller wäre der Einsatz einer Bibliothek für das entsprechende Archivformat und selektives Entpacken (z.B. Artikel ab 10 KB Größe). Der Einsatz eines Crawlers bzw. „API-Bots“ oder das manuelle Beschaffen ist sicherlich auf den ersten Blick unbequemer und verringert die verfügbare Textmenge entsprechend, kann aber aus Ressourcengründen die sinnvollere Variante sein. Idealerweise sind die Textquellen für das Programm austauschbar und/oder mit geringem Aufwand für den Nutzer beschaffbar.

Textextraktion

Häufig sind Texte mit Metadaten verknüpft (z.B. HTML-/Wikitext-Tags). Zum Filtern bietet sich ein einschließendes Verfahren (Inklusion) an, indem man idealerweise durch reguläre Ausdrücke oder eine Parserbibliothek die gewünschte Information erkennt (z.B. alles zwischen `<p>` und `</p>`). Ebenso kann ein ausschließendes Verfahren (Exklusion) genutzt werden, welches ungewünschte Information eliminiert (z.B. alles zwischen `<script ...>` und `</script>` oder < und >), aber weniger Feinabstimmung ermöglicht.

Für Wikitext wird mehr Geschick benötigt, z.B. muss [[*Linkziel* | *Linktext*]] zu *Linktext* werden. Zu beachten sind die Grenzen regulärer Ausdrücke, insbesondere bei der Verarbeitung rekursiver Metadaten (z.B. verschachtelte Tags).

Auch nach Sinn der Textbestandteile sollte eine grobe Auswahl erfolgen. Beispielsweise dienen `<tt>`- und `<pre>`-Tags oft der Formatierung von Quell- und Pseudocode. Das vermehrte Auftreten von Sonderzeichen deutet auf fremdsprachige Inhalte hin, Listen und Tabellen beinhalten selten sinnvoll aufgebaute Sätze.

Zum Extrahieren der Wörter genügt ein kurzer regulärer Ausdruck, z.B.

```
[^a-z]([a-z])+(?=[^a-z])
```

oder das Aufspalten der gesamten Zeichenfolge an Trennzeichen mit Nachbearbeitung (u.a. Entfernen von Zahlen). Das entstehende Wortverzeichnis sollte entweder von sehr seltenen Begriffen bereinigt werden (insbesondere wenn man sich an Wikipedia zu schaffen gemacht hat), oder man verkauft klingonisches Essen und walisische Dorfnamen als wertvolle Bereicherung.

Verwendbare Informationen

Aus den aufbereiteten Texten lassen sich u.a. folgende Informationen gewinnen, welche im Folgenden zum Einsatz kommen:

- Menge der Wörter
- Wort- und Buchstabenhäufigkeit
- Aufeinanderfolgende Wörter bzw. sinnvolle Wortpaare und ihre Häufigkeiten

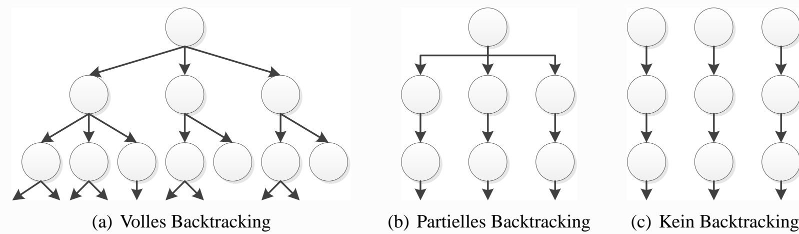
3.3 Nützliche Datenstrukturen und Algorithmen

Nutzereingabe

Die Nutzereingabe stellt mathematisch eine Multimenge dar: Wichtig ist, wie oft ein Buchstabe vorkommt, nicht wo. Zwar kann man sie als Zeichenfolge behandeln, effizienter wird jedoch das Verarbeiten als Hashtable (Dictionary), die jedem Buchstaben seine Anzahl zuordnet, als Histogramm (26-stelliges Array o.ä.), indem die Häufigkeiten von a bis z abgetragen werden oder als Zeichenfolge mit sortierten Buchstaben.

Wortliste

Die Menge der extrahierten Wörter aneinandergereiht bildet diese einfache Datenstruktur. Ein Suchalgorithmus muss hier prüfen, ob ein Wort Teilmenge der Eingabemenge ist. Bei einem Treffer muss mit den verbleibenden Buchstaben in der Eingabe-Menge genauso verfahren werden. Ist kein Treffer in Sicht, kann die Entscheidung für das letzte Wort zurückgenommen werden (volles Backtracking), an anderer Stelle neu begonnen werden (kein Backtracking) oder in einer bestimmten Rekursionsebene neu eingesetzt werden (partielles Backtracking).



(a) Volles Backtracking (b) Partielles Backtracking (c) Kein Backtracking

Abbildung 3.1: Mögliche Vorgehensweisen bei der Anagrammsuche

Die Reihenfolge der Wörter kann entweder durch eine Sortierung (z.B. nach Häufigkeit) oder zufällig festgelegt werden. Je nach Beschaffenheit der Wortliste liefern die Verfahren Ergebnisse unterschiedlichster Qualität hinsichtlich Laufzeit und Auswahl und können kombiniert werden.

Eine ganz grundsätzliche Alternative ist es, nicht Worte, sondern gleich Wortfolgen abzuspeichern. Das verursacht natürlich einen deutlich höheren Speicheraufwand, bei geeigneter Organisation (s.u.) und einer sinnvollen Begrenzung der Länge kann das verschmerzbar sein. Bei der Suche nach Anagrammen kann man dann gleich die Multimenge der Eingabe mit der abgespeicherten Multimenge der gesamten Wortfolge vergleichen – und dann aufs Backtracking verzichten.

Wortpaare

Um „Sinn" in die Anagramme zu bringen, sollte beim Übergang von einem Wort zum nächsten eine entsprechende Information die Auswahl lenken, z.B. die Häufigkeit, mit der ein Wort im Text nach einem anderen auftritt. Eine effiziente Realisierung könnte beispielsweise eine jedem Wort zugeordnete Liste sein, die jeweils Verweise (z.B. die Positionen einiger möglicher Folgewörter) sortiert nach Häufigkeit oder mittels Heuristik sammelt. Dafür ist eine ausführlichere Analyse der Textquelle Voraussetzung, denn nicht nur Reihenfolge, sondern auch Interpunktion entscheiden, ob zwei Wörter benachbart sind oder nicht.

Trie

Ein *Trie* (von Englisch „Retrieval") ist ein Baum, der gemeinsame Anfangsbuchstaben zu einem Knoten zusammenfasst und von dort rekursiv zu allen möglichen Fortführungen verzweigt. Wortenden müssen markiert werden, um eine Unterscheidung zweier Wörter, von denen eines den Anfang eines anderen Darstellt (z.B. „be" und „bear") zu gewährleisten.

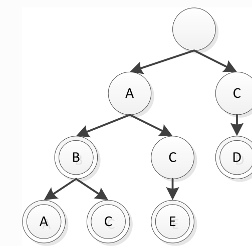


Abbildung 3.2: Trie mit den Wörtern AB, ABA, ABC, ACE und CD

Er stellt in Hinblick auf Suchzeit und Speicherplatz eine effizientere Struktur als die Liste dar. Das Suchen nach einem passenden Wort wird nun zu einem Backtracking mit dem Ziel, ein Blatt zu erreichen; es muss nicht bei jedem Fehlschlag an einem Wortanfang neu ansetzen. Zusätzliche Optimierungen beinhalten z.B. das Sortieren von Tochterknoten nach Häufigkeit, um wahlweise die „exotischsten" oder „trivialsten" Treffer vorzuziehen. Erlaubt man in den Knoten eine Zeichenfolge anstelle eines einzelnen Zeichens, so kann man durch Zusammenfassen nicht-verzweigender Abschnitte einen noch stärker optimierten sog. *Patricia-Trie* herbeiführen:

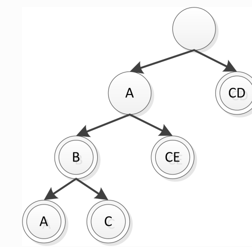


Abbildung 3.3: Patricia Trie mit den Wörtern AB, ABA, ABC, ACE und CD

Wortpaare/Ketten können hier beispielsweise durch eine Zeigerliste von einem Blatt auf andere Blätter realisiert werden.

Komplexitätsbetrachtungen fallen sehr individuell aus, sollten jedoch die Wahl der Datenstruktur begründen können. Generell wachsen Patricia-Tries langsamer als Tries, welche wiederum langsamer als Wortlisten wachsen. Dem kleineren Speicherverbrauch steht jedoch ein Laufzeitoverhead beim Erstellen gegenüber.

Modifizierte Hashtable

Die sortierte Buchstabenfolge von Anagrammen lautet stets gleich. Diese Invariante lässt sich für schnelle Zugriffe auf Hashtables/Dictionaries nutzen, indem das sortierte Wort als Schlüssel genommen und die daraus gebildeten sinnvollen Wörter in einer Liste als Wert gespeichert werden. Nun kann man mit konstantem Aufwand feststellen, ob man ein Anagramm direkt mit der (ebenfalls sortierten) Nutzereingabe finden kann. Anschließend gilt es, Zerlegungen der Eingabe zu finden, bei der jede Teilmenge mindestens einen Treffer landet. Geschieht dies durch Ausprobieren, wächst die Worst-Case-Laufzeit annähernd exponentiell mit der Wortlänge. Eine Kombination aus Tries (finden eines „Anfangs“) und Hashtables (Verarbeiten der verbleibenden Buchstabenmenge) kann sich als sehr schnell erweisen.

3.4 Nachbearbeitung

Kleine Trefferanzahl

Dass eine Nutzereingabe keine oder zu wenige Anagramme hervorbringt, ist besonders bei einer kleinen Datenbasis und experimentierfreudigen Nutzern nicht selten. Die bloße Meldung über den Misserfolg kann jedoch von einigen unsinnigen Vorschlägen begleitet werden, die einen fast-Treffer um eine vielleicht amüsante Wortschöpfung (z.B. ein Wortfragment) ergänzen.

Große Trefferanzahl

Wenn man bereits bei der Suche geschickt vorgegangen ist (Sortierungen, Heuristik) und tatsächlich einige Treffer landet, reicht ggf. schon das Abbrechen des Algorithmus nach einer überschaubaren Anzahl. Andernfalls kann eine Bewertung angebracht sein, die z.B. den „Sinn“ der Wortgruppe anhand der Häufigkeit der enthaltenen Wortpaarungen einschätzt. Auch Faktoren wie die Anzahl der Wörter oder die Anzahl seltener Buchstaben kann einfließen, um die Ergebnisse interessanter zu machen - hier sind keine Grenzen gesetzt.

3.5 Zusätzliche Erweiterungen

Heuristik

Wenn es darum geht, Buchstaben oder Folgewörter nach (kontextabhängiger) Häufigkeit auszuwählen, muss nicht die komplette Häufigkeitsverteilung bekannt sein. Als Beispiel sei die Heuristik *Move-To-Front* aufgeführt: In einer ein- oder zweifach verketteten Liste werden wichtige Elemente gesammelt, indem jedes neue Element vorne angefügt wird und jedes bekannte Element aus der Liste entnommen und nach vorn gestellt wird. Begrenzt man die

Listenlänge und schneidet nach 100 Gliedern ab, so kann man z.B. 100 sehr häufige Paarbildungen von Wörtern erfassen ohne jemals mehr als 100 Paare gespeichert zu haben. Der quadratische Speicheraufwand beim Sammeln möglicher Wortverbindungen kann damit linear beschränkt werden.

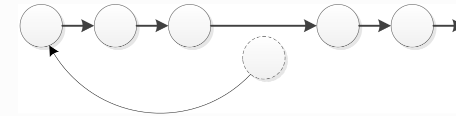


Abbildung 3.4: Prinzip der Move-To-Front-Heuristik

Benutzerdefinierte Parameter

Gerade bei der Nachbearbeitung der Ergebnisse kann der Nutzer wählen, nach welchen Kriterien die Anagramme bewertet und sortiert werden, ggf. die Anzahl vorgeben und bei kleinen Trefferzahlen somit kreative Wortschöpfungen erzwingen. Auch bei der Auswahl der Wörter aus dem Text kann der Nutzer bereits eingreifen um z.B. eine Minimalhäufigkeit (oder interessanterweise auch eine Maximalhäufigkeit) der verwendeten Wörter einzustellen. Wenn man die sinngebende Komponente abschalten kann, so erhält man von Zeit zu Zeit sehr lustige Ergebnisse.

Quellenabgleich

Große Texte sind für die Extraktion von Satzbau-Informationen und Worthäufigkeiten unverzichtbar. Dennoch können die Wörter mit einer seriösen Quelle abgeglichen werden um sehr fiktive Erscheinungen zu umgehen, z.B. mit Wörterbüchern von www.dicts.info. Dabei sollten Flexionen zumindest rudimentär behandelt werden, denn die Wortendung „s“ für Plural-Substantive, Verben in der dritten Person, Partizipial- und Präteritums-Endungen usw. sind in den Wörterbüchern selten extra aufgeführt.

Sprachverarbeitung

Das Python-Framework NLTK (Natural Language Toolkit) bietet einerseits mächtige Werkzeuge zum Parsen und Rekonstruieren von Sprachsyntax, ist aber umfangreich und aufwändig zu bedienen. Jedoch kann der integrierte „Collocation Finder“ beispielsweise Gruppen aus zwei oder drei Wörtern anhand bestimmter Kriterien auswählen und als Basis für eigene Algorithmen dienen.

Auch andere Formen der Sprachverarbeitung sind denkbar: Wissen über Wortarten kann bei der Bewertung der Sinnhaftigkeit von Anagrammen nützlich sein. Auch der Einsatz von Grammatiken ist möglich, aber natürlich aufwändig.

3.6 Bewertungskriterien

- Die verwendeten Textquellen sollen angegeben, ihre Eigenschaften insbesondere die Länge und Anzahl der enthaltenen unterschiedlichen Wörter) sollen ersichtlich werden.
- Laut Aufgabenstellung sind „hinreichend umfangreiche Texte“ zu verwenden. Das ist keine klare Vorschrift, aber es soll doch mit Texten gearbeitet werden, die in der Größenordnung (kürzerer) Bücher liegen.
- Die Aufgabenstellung erwähnt außerdem „weitere für das Bilden sinnvolle Anagramme wichtige Informationen“. Es ist deshalb unzureichend, wenn allein die enthaltenen Wörter aus den Textquellen extrahiert werden.
- Die für die Darstellung der Eingabe und der Wortmengen (sowie der weiteren Informationen aus den Texten) verwendeten Datenstrukturen sollen nicht unnötig ineffizient sein. Grundsätzlich sollte erkannt sein, dass die Reihenfolge der Buchstaben in der Eingabe ignoriert werden und die Anagrammsuche davon profitieren kann.
- Auch die Suchverfahren sollen nicht zu ineffizient sein. An einer Eingabe von 15 Zeichen sollte kein Programm scheitern, auch nicht bei umfangreicher Wortmenge.
- Es ist nicht ausreichend, einfach nur beliebige Wortkombinationen zu ermitteln, die zwar ein Anagramm der Eingabe darstellen, aber sprachlich unsinnig sind. Es muss eine Methode realisiert sein, die (gemäß einer plausiblen Definition) sinnvolle Anagramme findet. Ob das Suchverfahren selbst sinnvolle Anagramme bevorzugt oder ob erst nach der Suche die gefundenen Anagramme bewertet werden, ist nicht entscheidend. Letzlich soll aber die Menge der endgültig ausgegebenen Anagramme überschaubar sein (vgl. Aufgabenstellung).
- Eingabe und Ausgaben sollen aus mehreren Wörtern bestehen können.
- Auch bei dieser Aufgabe, in der mit großen Datenmengen umgegangen werden muss, sind ein paar grundlegende Gedanken zur Effizienz der eingesetzten Methoden Pflicht.
- Wurden überraschend gute Anagramme angegeben, dann wurde genauer getestet. Konnte das eingesandte Programm die Beispielausgaben nicht reproduzieren, wurden reichlich Minuspunkte vergeben.

Perlen der Informatik – aus den Einsendungen

Allgemeines

Da die Programme in Java sind, sollten sie prinzipiell überall laufen – ich warte noch darauf, dass Sun die JRE auf meine Socken portiert.

… und dann hätten wir eine Komplexität im Bereich von O(1!), was jedoch recht abwegig ist.

Aufgabe 1: Universelle Öffnungscodes

Jedes Mal, wenn mein Programm einen neuen Bitstring herausgefunden hat, werden die eliminierten Schalterstellungen in der Tabelle abgehackt.

Aufgabe 2: Das Turmrestaurant

Der Ober ärgert sich bei etwa jeder 20. eintreffenden Gruppe. Hoffentlich hat er ein starkes Herz.

Betritt beispielsweise die rote Gruppe das Restaurant, so werden sie hinter die Grauen platziert. *Interessant, insbesondere wenn die entsprechende Abbildung schwarz/weiß ist.*

Nun muss man nur noch die Gruppen speichern, damit man sie später auch wieder löschen kann.

```
void sitDown(Group group) throwsWaiterAngryException
```

Aufgabe 3: Anagramme

Die erste Idee war es, die Anagramme nur aus Wörtern zu erstellen, die auch zusammenhängend in Texten auftauchen. Doch Tests haben gezeigt, dass dies nicht praktisch ist, denn nur selten wurden überhaupt Wörter gefunden, die zusammen in Texten auftauchen.

Jedoch hat es zuweilen einen gewissen Reiz, dem Programm dabei zuzusehen, wie es bei seiner Reise entlang der quer durch die gesamte Wikipedia gespannten Assoziationsketten die unterschiedlichsten Themenbereiche kreuzt.

Beispielhafte (oder -lose) Anagramme:

anagram making	managing karma	computer science	creep cuts income
combinatorics	ironic cat mobs	george bush	he bugs gore
informatics	firm actions	mathematics	tea mismatch