33. 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 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 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.

Bewertungsbogen Aus der ersten Runde oder auch aus früheren Wettbewerbsteilnahmen kennen Sie den Bewertungsbogen, der angibt, wie Ihre Einsendung die einzelnen Bewertungskriterien erfüllt hat. Diesmal erhalten Sie keinen Bewertungsbogen auf Papier, sondern können ihn im Anmeldesystem PMS einsehen. Im Zusammenhang mit dieser Neuerung haben wir auch das Punktsystem der zweiten Runde umgestellt. In der ersten Runde ging die Bewertung noch von 5 Punkten aus, von denen bei Mängeln dann abgezogen werden konnte. In der zweiten Runde geht die Bewertung nun von 20 Punkten aus; dafür gibt es deutlich mehr Bewertungskriterien, bei denen Punkte abgezogen oder auch hinzuaddiert werden konnten.

Terminlage der zweiten Runde Für Abiturienten ist der Terminkonflikt zwischen Abiturvorbereitung und zweiten Runde sicher nicht ideal. Doch leider bleibt uns nur die erste Jahreshälfte für die zweite BwInf-Runde: In der zweiten Jahreshälfte läuft nämlich die zweite Runde des Mathewettbewerbs, dem wir keine Konkurrenz machen wollen. 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 der Dokumentation 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.

Bewertungskriterien Bei den im Folgenden beschriebenen Lösungsideen 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, wesentliche Auszüge aus dem Quellcode enthalten), Quellcode (insbesondere: Strukturierung und Kommentierung, gute Übersicht durch Programm-Dokumentation) und Programm (keine Implementierungsfehler).

Danksagung An der Erstellung der Lösungsideen haben mitgewirkt: Meike Grewing (Aufgabe 1), Peter Rossmanith jr. und Nikolai Wyderka (Aufgabe 2) sowie Maximilian Janke (Aufgabe 3). Die Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt, und zwar aus Vorschlägen von Peter Rossmanith sr. (Aufgabe 1), Torben Hagerup (Aufgabe 2) und Rainer Gemulla (Aufgabe 3).

Aufgabe 1: Seilschaften

1.1 Grundlegende Überlegungen

Zunächst einmal wollen wir die Problemstellung verstehen: Die Eingabe besteht aus einer maximalen Gewichtsdifferenz d, einer Menge von *Personen* und *Steinen*, die jeweils ein Gewicht und eine Ausgangsposition (*oben* oder *unten*) haben. Die beiden Körbe des Seilzugs sind zu Beginn leer. Nun gelten die folgenden Bedingungen:

- Personen können selbst in die Körbe hinein bzw. aus ihnen heraus steigen und / oder Steine hineinlegen bzw. herausnehmen.
- Ist der Inhalt des oberen Korbes schwerer als der des unteren, so *fährt* der Seilzug, die beiden Körbe tauschen also ihre Positionen.
- Sofern sich mindestens eine Person in einem der Körbe befindet, darf die Gewichtsdifferenz (genauer: die Differenz zwischen den beiden Gesamtgewichten der Korbinhalte) den Maximalwert d nicht überschreiten. Wenn nur Steine in den Körben sind, spielt d keine Rolle, insbesondere können Steine alleine nach unten fahren.
- Ein Stein kann nur dann in einen Korb gelegt oder herausgenommen werden, wenn sich an seiner Position auch eine Person befindet.

Gesucht ist nun nach einer kürzesten Folge von Fahrten, mit der alle Personen sicher entkommen können. Jede solche Fahrt können wir als Übergang zwischen zwei *Zuständen* betrachten: Ein Zustand ist definiert durch die Positionen aller Objekte. Eine Fahrt vertauscht die Positionen der Objekte, die sich in den Körben befinden, und überführt so einen Zustand in einen anderen.

Gleichwertige Zustände

Wir werden im Folgenden häufig überprüfen müssen, ob zwei Zustände A und B *gleichwertig* sind. A und B sind genau dann gleichwertig, wenn wir A ohne eine Seilzug-Fahrt so ändern können, dass B daraus entsteht. Dabei sind die oben genannten Bedingungen des Rätsels zu berücksichtigen. Zum Beispiel sind zwei Zustände gleichwertig, die sich nur darin unterscheiden, dass eine bestimmte Person im Korb oder außerhalb des Korbes ist - die Person kann einfach aus dem Korb heraus- bzw. in ihn hinein steigen. Das gilt für Steine jedoch nicht. Sofern keine Person anwesend ist, müssen wir also bei den Positionen auch zwischen *im Korb* und *außerhalb* unterscheiden.

Wir können davon ausgehen, dass zu jeder Zeit mindestens eine Person auf dem Turm ist - ansonsten sind schon alle entkommen und es ist nichts mehr zu tun. Das heißt, wir können oben auf die Unterscheidung zwischen *im Korb* und *außerhalb* verzichten und es gibt insgesamt drei verschiedene Positionen: *oben*, *unten im Korb* und *unten außerhalb*. Wir unterscheiden zwei Fälle:

- Falls eine Person unten ist: A und B sind genau dann gleichwertig, wenn in beiden Zuständen dieselben Objekte auf dem Turm sind. Alle anderen Objekte sind dann in beiden Zuständen unten. Dabei ist irrelevant, ob diese Objekte im Korb sind oder nicht; eine anwesende Person kann alle Objekte bewegen.
- Falls keine Person unten ist: A und B sind genau dann gleichwertig, wenn in beiden Zuständen jeweils dieselben Objekte auf dem Turm sind, unten im Korb sind und unten außerhalb des Korbs sind. Insbesondere kann man einen einzelnen Stein zwar nach unten fahren lassen, ohne eine unten anwesende Person kann er den Korb dort aber nicht verlassen.

1.2 Breitensuche

Wie lösen wir mit diesen Überlegungen das Problem? Es liegt nahe, die Menge aller Zustände und die Übergänge dazwischen als *Graph* zu betrachten. Jeder Zustand stellt einen Knoten dar, jeder Übergang eine Kante: Gibt es einen Übergang, der einen Zustand A in einen Zustand B umwandelt, so enthält der Graph eine Kante zwischen den Knoten A und B.

Beginnend mit dem Startzustand, der durch die Eingabe definiert wird, suchen wir nun mittels Breitensuche nach erreichbaren Zuständen: Wir speichern eine Warteschlange Q von Zuständen, die wir gefunden haben und für die wir noch ihre Nachbarzustände suchen müssen. Am Anfang enthält Q nur den Startzustand. In einer Menge F speichern wir alle Zustände, die wir schon komplett bearbeitet haben (d.h. für die wir schon alle Nachbarzustände gesucht haben).

Solange Q nicht leer ist, entnehmen wir den ersten Zustand x und fügen ihn zu F hinzu. Dann wenden wir auf x jeden erlaubten Übergang an. Dies liefert uns eine Menge von *Folgezuständen*. Für jeden dieser Folgezustände überprüfen wir, ob wir ihn schon kennen. Falls ja, verwerfen wir ihn; im Laufe der Suche werden die Wege zu neu entdeckten Zuständen nur länger, wir suchen aber nach einer kürzesten Folge von Seilzug-Fahrten. Falls wir jedoch einen neuen Zustand z finden, prüfen wir, ob er ein *Zielzustand* ist, ob sich in z also keine Person mehr auf dem Turm befindet. Ist dies der Fall, so geben wir z aus, mitsamt der Folge von Seilzug-Fahrten, die uns vom Startzustand zu z geführt hat. Andernfalls reihen wir ihn in die Warteschlange Q ein und fahren mit dem nächsten Knoten fort.

Stellen wir irgendwann fest, dass Q keine Zustände mehr enthält, haben aber noch keinen Zielzustand gefunden, so gibt es einfach keine Folge von Seilzug-Fahrten, mit der alle Personen sicher aus dem Turm entkommen können. Dies geben wir dann aus.

Repräsentation und Abgleich von Zuständen

Der Vergleich zweier Zustände ist in der Breitensuche eine häufige Operation. Es kann sich also lohnen, diese möglichst effizient zu gestalten. Zwei Ideen hierzu: Die Zustände könnten verwaltet werden

- … in einer Hashtabelle, wobei die Hashfunktion für gleichwertige Zustände den gleichen Hashwert berechnet.

- … als Bitvektoren, wobei entscheidend ist, dass gleichwertige Zustände über schnelle Bitoperationen auch als gleichwertig erkannt werden können.

Die Verwaltung der Zustände sollte auch weitere Operationen effizient unterstützen, etwa die Entscheidung über die Anwendbarkeit einer Fahrt (s.u.).

Vorausberechnen aller erlaubten Fahrten

Um die Berechnung etwas zu beschleunigen, berechnen wir vor Beginn der Breitensuche alle zulässigen Fahrten. Bei n Objekten gibt es 2^n Teilmengen, also theoretisch auch etwa 2^n Möglichkeiten die Körbe zu beladen. Durch die Beschränkung, dass Personen nur sicher fahren, wenn sich die Gewichte der beiden Korbinhalte nicht um mehr als d unterscheiden, gibt es im Allgemeinen jedoch viel weniger erlaubte Fahrten. Dadurch, dass wir diese im Voraus berechnen, müssen wir während der Breitensuche nur noch überprüfen, welche von ihnen sich jeweils auf den aktuellen Zustand anwenden lassen.

Eine Fahrt (O,U) ist durch die Angabe von zwei Mengen von Objekten definiert: Der Inhalt O des oberen Korbs und der Inhalt U des unteren Korbs. Dabei muss O schwerer sein als U und die maximale Gewichtsdifferenz eingehalten werden, oder beide Mengen dürfen nur Steine enthalten. Wir können also recht einfach alle zulässigen Fahrten bestimmen:

1. Bestimme alle 2^n Teilmengen der n Objekte.
2. Für jede Teilmenge M und jede dazu disjunkte leichtere Teilmenge N: Falls $M-N \leq d$ gilt oder beide Mengen nur Steine enthalten, füge (M,N) zur Menge aller zulässigen Fahrten hinzu.

Anwenden einer Fahrt

Wir können eine Fahrt (O,U) genau dann auf einen Zustand z anwenden, wenn alle Objekte aus O sich in z auf dem Turm befinden und alle Objekte aus U unten. Falls sich unten mindestens eine Person befindet, reicht dies aus. Falls jedoch unten ausschließlich Steine sind, müssen wir außerdem überprüfen, ob der Inhalt des unteren Korbes genau U ist, da Steine sich bekanntlich nicht selbst bewegen können.

Ist (O,U) anwendbar, so vertauschen wir einfach die Positionen der Objekte in O und U. Anschließend befinden sich die Objekte aus U oben, die aus O unten im Korb.

1.3 Gedanken zur Laufzeit

Vorbereitung

Das Berechnen aller Teilmengen der n Objekte und aller zulässigen Fahrten erfordert Zeit $\mathbf{O}(2^n)$. Wächst n, so wird diese Laufzeit schnell riesig. Es stellt sich also die Frage, ob wir nicht ohne das Berechnen der Teilmengen eine deutlich bessere Laufzeit erreichen könnten. Betrachten wir jedoch die Lösung zur Beispieleingabe 2, so stellen wir fest, dass diese 2^n-1

Seilzugfahrten erfordert. Allein das Ausgeben der Lösung braucht also schon Zeit $\mathbf{O}(2^n)$. Wir wissen somit, dass wir im Allgemeinen keine bessere Laufzeit als $\mathbf{O}(2^n)$ erreichen können.

Breitensuche

Interessanter ist die Laufzeit für die Breitensuche. Diese hängt maßgeblich davon ab, wie viele Zustände es gibt. Im schlimmsten Fall ist erst der letzte gefundene Zustand ein Zielzustand, wie zum Beispiel bei Eingabe 2. Sind s der n Objekte Steine, so gibt es $2^n + 3^s - 2^s$ verschiedene Zustände (jedes Objekt kann entweder oben oder unten sein, und falls alle Personen oben sind, müssen wir bei den Steinen zwischen drei verschiedenen Positionen unterscheiden), also nicht mehr als 3^n.

Die Breitensuche dauert besonders dann „unnötig“ lange, wenn auf die einzelnen Zustände viele Übergänge anzuwenden sind; dann müssen erst sehr viele Zustände betrachtet werden, bevor ein Zielzustand gefunden wird.

Mögliche Verbesserungen

Wir haben oben abgeschätzt, wie hoch die Laufzeit im schlimmsten Fall ist. Vielleicht können wir Heuristiken anwenden, um schneller einen Zielzustand zu finden?

In der Aufgabenstellung ist nach einer *kürzesten* Folge von Seilzug-Fahrten gefragt. Das heißt, wenn wir eine mögliche Folge von Seilzug-Fahrten finden, die zu einem Zielzustand führt, so müssen wir auch sicherstellen, dass es keine kürzere gibt. Die Breitensuche garantiert uns dies, auf einigen Eingaben (z.B. Beispieleingabe 4) ist sie aber sehr langsam. Eine leichte Verbesserung ist möglich, wenn parallel zur Breitensuche vom Ausgangszustand eine Breitensuche vom Zielzustand aus in Richtung Ausgangszustand durchgeführt wird. Man spricht von einer *bidirektionalen Breitensuche*; eine Lösung ist gefunden, wenn sich die beiden Suchen „treffen“, also den gleichen Zustand erreichen.

Eine Alternative ist die $A^\star$-Suche: Bei der Breitensuche haben wir als nächsten zu bearbeitenden Knoten immer einen bekannten Knoten ausgewählt, zu dem der bisherige Weg minimal war. Ist dieser Zustand aber sehr weit weg von einem Zielzustand (etwa, weil sich alle Personen auf dem Turm befinden), so scheint es wahrscheinlicher, dass ein anderer Zustand schneller zum Ziel führt. Wir könnten also eine *Bewertungsfunktion* einführen, die abschätzt, wie viele Seilzug-Fahrten von einem bestimmten Zustand aus noch benötigt werden. Im Suchalgorithmus wählen wir dann einen bekannten Zustand, für den die bisherige Anzahl von Seilzug-Fahrten plus der Wert der Bewertungsfunktion minimal ist.

1.4 Lösungen zu den Beispieleingaben

Die vorgegebenen Beispieleingaben haben die im Folgenden gezeigten Lösungen. Die Ausgaben enthalten in jeder Zeile einen Zustand, beginnend mit dem gegebenen Anfangszustand. Die in den folgenden Zeilen angegebenen Zustände sind jeweils aus dem Zustand der vorhergehenden Zeile durch eine Seilzug-Fahrt direkt erreichbar. In den Zuständen werden die

Personen und Steine ähnlich wie in den Eingaben, aber etwas kompakter beschrieben: Der erste Buchstabe (`P` oder `S`) gibt an, ob es sich um eine Person oder einen Stein handelt; die folgende Zahl gibt das Gewicht an, und abschließend folgt die Position: ^ für oben und _ für unten. In der letzten Zeile müssen also alle mit `P` beginnenden Strings mit _ enden.

Beispiel 0: Originales Rätsel

```
10 Fahrten:
P195^ P105^ P90^ S75^
P195^ P105^ P90^ S75_
P195^ P105^ P90_ S75^
P195^ P105_ P90^ S75^
P195^ P105_ P90^ S75_
P195_ P105^ P90^ S75^
P195_ P105^ P90^ S75_
P195_ P105^ P90_ S75^
P195_ P105_ P90^ S75^
P195_ P105_ P90^ S75_
P195_ P105_ P90_ S75^
```

Beispiel 1

```
14 Fahrten:
S1^ P2^ S4^ P8^ S16^ P32^ S64^
S1_ P2^ S4^ P8^ S16^ P32^ S64^
S1^ P2_ S4^ P8^ S16^ P32^ S64^
S1_ P2_ S4_ P8^ S16^ P32^ S64^
S1^ P2^ S4^ P8_ S16^ P32^ S64^
S1_ P2^ S4_ P8_ S16_ P32^ S64^
S1^ P2_ S4_ P8_ S16_ P32^ S64^
S1_ P2_ S4_ P8_ S16_ P32^ S64^
S1^ P2^ S4^ P8^ S16^ P32_ S64^
S1_ P2^ S4^ P8^ S16_ P32_ S64^
S1^ P2_ S4^ P8^ S16_ P32_ S64^
S1_ P2_ S4_ P8^ S16_ P32_ S64^
S1^ P2^ S4^ P8_ S16_ P32_ S64^
S1_ P2^ S4_ P8_ S16_ P32_ S64^
S1^ P2_ S4_ P8_ S16_ P32_ S64^
```

Beispiel 2

```
511 Fahrten:
P1^ P2^ P4^ P8^ P16^ P32^ P64^ P128^ P256^
P1_ P2^ P4^ P8^ P16^ P32^ P64^ P128^ P256^
P1^ P2_ P4^ P8^ P16^ P32^ P64^ P128^ P256^
P1_ P2_ P4^ P8^ P16^ P32^ P64^ P128^ P256^
P1^ P2^ P4_ P8^ P16^ P32^ P64^ P128^ P256^
P1_ P2^ P4_ P8^ P16^ P32^ P64^ P128^ P256^
```

```
P1^ P2_ P4_ P8^ P16^ P32^ P64^ P128^ P256^
P1_ P2_ P4_ P8^ P16^ P32^ P64^ P128^ P256^
...
P1_ P2_ P4_ P8_ P16_ P32_ P64_ P128_ P256_
```

Beispiel 3

```
27 Fahrten:
P116^ P231^ S50_ P55^ P101_ P185^ P211_ P200_ P224^
P116_ P231_ S50^ P55^ P101^ P185^ P211^ P200^ P224_
P116_ P231_ S50_ P55^ P101^ P185^ P211^ P200^ P224_
P116^ P231^ S50^ P55^ P101^ P185^ P211_ P200_ P224_
P116^ P231^ S50_ P55^ P101^ P185^ P211_ P200_ P224_
P116_ P231^ S50^ P55_ P101_ P185^ P211^ P200_ P224_
P116_ P231^ S50_ P55_ P101_ P185^ P211^ P200_ P224_
P116_ P231_ S50^ P55^ P101^ P185^ P211_ P200_ P224^
P116_ P231_ S50_ P55^ P101^ P185^ P211_ P200_ P224^
P116^ P231_ S50^ P55_ P101_ P185^ P211_ P200^ P224_
P116^ P231_ S50_ P55_ P101_ P185^ P211_ P200^ P224_
P116^ P231_ S50^ P55_ P101^ P185_ P211_ P200_ P224^
P116^ P231_ S50_ P55_ P101^ P185_ P211_ P200_ P224^
P116_ P231_ S50^ P55^ P101^ P185_ P211_ P200_ P224^
P116_ P231_ S50_ P55^ P101^ P185_ P211_ P200_ P224^
P116^ P231_ S50^ P55_ P101_ P185_ P211_ P200^ P224_
P116^ P231_ S50_ P55_ P101_ P185_ P211_ P200^ P224_
P116_ P231_ S50^ P55^ P101_ P185_ P211_ P200^ P224_
P116_ P231_ S50_ P55^ P101_ P185_ P211_ P200^ P224_
P116_ P231_ S50_ P55^ P101_ P185^ P211_ P200_ P224_
P116_ P231_ S50^ P55_ P101_ P185^ P211_ P200_ P224_
P116^ P231_ S50^ P55^ P101_ P185_ P211_ P200_ P224_
P116^ P231_ S50_ P55^ P101_ P185_ P211_ P200_ P224_
P116^ P231_ S50^ P55_ P101_ P185_ P211_ P200_ P224_
P116^ P231_ S50_ P55_ P101_ P185_ P211_ P200_ P224_
P116_ P231_ S50^ P55^ P101_ P185_ P211_ P200_ P224_
P116_ P231_ S50_ P55^ P101_ P185_ P211_ P200_ P224_
P116_ P231_ S50^ P55_ P101_ P185_ P211_ P200_ P224_
```

Beispiel 4

```
23 Fahrten:
P195^ P105^ P90^ S75^ P137^ P55^ P101_ P185^ P199^
P195^ P105^ P90^ S75_ P137^ P55^ P101_ P185^ P199^
P195^ P105^ P90_ S75^ P137^ P55^ P101_ P185^ P199^
P195^ P105^ P90_ S75_ P137^ P55^ P101_ P185^ P199^
P195^ P105^ P90_ S75^ P137^ P55^ P101^ P185_ P199^
P195^ P105^ P90_ S75_ P137^ P55^ P101^ P185_ P199^
P195^ P105_ P90^ S75^ P137^ P55_ P101^ P185^ P199_
P195^ P105_ P90^ S75_ P137^ P55_ P101^ P185^ P199_
P195_ P105^ P90^ S75^ P137^ P55_ P101^ P185^ P199_
P195_ P105^ P90^ S75_ P137^ P55_ P101^ P185^ P199_
```

```
P195_ P105^ P90_ S75^ P137^ P55_ P101^ P185^ P199_
P195_ P105^ P90_ S75_ P137^ P55_ P101^ P185^ P199_
P195^ P105_ P90^ S75^ P137_ P55^ P101^ P185_ P199_
P195^ P105_ P90^ S75_ P137_ P55^ P101^ P185_ P199_
P195_ P105^ P90^ S75^ P137_ P55^ P101^ P185_ P199_
P195_ P105^ P90^ S75_ P137_ P55^ P101^ P185_ P199_
P195_ P105_ P90^ S75^ P137_ P55_ P101_ P185^ P199_
P195_ P105_ P90^ S75_ P137_ P55_ P101_ P185^ P199_
P195_ P105^ P90^ S75^ P137_ P55_ P101_ P185_ P199_
P195_ P105^ P90^ S75_ P137_ P55_ P101_ P185_ P199_
P195_ P105^ P90_ S75^ P137_ P55_ P101_ P185_ P199_
P195_ P105^ P90_ S75_ P137_ P55_ P101_ P185_ P199_
P195_ P105_ P90^ S75_ P137_ P55_ P101_ P185_ P199_
P195_ P105_ P90_ S75^ P137_ P55_ P101_ P185_ P199_
```

Beispiel 5

Beispiel 5 hat keine Lösung.

Beispiel 6

```
9 Fahrten:
P109^ S120_ P156^ P55^ P149^ P185^ P85_
P109^ S120^ P156_ P55_ P149^ P185^ P85^
P109^ S120_ P156_ P55_ P149^ P185^ P85^
P109^ S120^ P156^ P55^ P149_ P185_ P85^
P109^ S120_ P156^ P55^ P149_ P185_ P85^
P109_ S120^ P156_ P55_ P149_ P185^ P85^
P109_ S120_ P156_ P55_ P149_ P185^ P85^
P109_ S120^ P156_ P55^ P149_ P185_ P85^
P109_ S120_ P156_ P55^ P149_ P185_ P85^
P109_ S120^ P156_ P55_ P149_ P185_ P85_
```

1.5 Erweiterungen

Sollte ein so schönes Rätsel überhaupt verändert werden? Wer es nicht lassen will, hat schon einige Möglichkeiten. Insbesondere könnten weitere Typen für die beteiligten Subjekte eingeführt werden, für die dann spezielle Regeln gelten. Ein Beispiel: Es werden auch Hunde transportiert, deren Herrchen (oder Frauchen; geschlechtsneutral vielleicht „Persönchen“) unter den Personen ist. Hunde können nur dann in den Korb hinein bzw. aus dem Korb hinaus springen, wenn ihr Persönchen sich an der gleichen Position befindet. Auch könnte man sich überlegen, die Steine durch Glaskugeln zu ersetzen (die Situation ist ja ohnehin recht märchenhaft); mit dem Nachteil, dass Glaskugeln, die alleine nach unten fahren, beim Aufprall zerstört werden, also nur für eine Fahrt nach unten zur Verfügung stehen. Solche Veränderungen können Repräsentation und den Abgleich von Zuständen sowie die Entscheidung über die Anwendbarkeit von Fahrten durchaus verkomplizieren und deswegen als Erweiterung gelten.

1.6 Bewertungskriterien

- Unabhängig vom gewählten Verfahren müssen die genannten Bedingungen des Rätsels alle korrekt eingehalten werden.
- Vollständige Suche und auch Tiefensuche sind zu aufwändig, Greedy-Ansätze liefern keine korrekten Ergebnisse; hierfür gibt es also Abzüge.
- Eine Breitensuche ist der beste bekannte Ansatz, allerdings sind Verbesserungen möglich. Die Laufzeit kann insbesondere dadurch verbessert werden, dass ...
  - ... die Zustandsmenge nicht unnötig groß ist; damit wird der Verzweigungsfaktor des Suchbaums kleiner. Dabei ist wichtig, für Personen und Steine nicht zu viele verschiedene Positionen anzunehmen, aber auch nicht zu wenig. Vier verschiedene Positionen (oben/unten im Korb/nicht im Korb) für Personen und auch Steine zu unterscheiden führt zu einer unnötig großen Zustandsmenge, während nur zwei verschiedene Positionen (oben und unten) zu wenig sind und zu Fehlern führen dürften.
  - ... der Vergleich von Zuständen optimiert wird; einige Möglichkeiten dazu sind oben skizziert.
- Anderweitige Verbesserungen des Verfahrens können mit Pluspunkten belohnt werden.
- Die theoretische Laufzeit sollte diskutiert werden. Eine formal wasserdichte Analyse ist nicht gefordert, aber es sollte nicht nur von konkreten Laufzeiten, sondern auch von Größenordnungen die Rede sein. Im Detail hängt das Ergebnis der Laufzeitanalyse davon ab, welche Positionen voneinander unterschieden werden.
- In der Regel sollten alle vorgegebenen Beispiele bearbeitet und deren Ergebnisse korrekt berechnet und dokumentiert sein. Wichtig ist insbesondere, dass für Beispiel 2 (wobei die Lösung in der Dokumentation nicht komplett angegeben sein sollte) und Beispiel 5 die richtigen Ergebnisse bestimmt und in der Dokumentation angegeben sind. Eigene aussagekräftige(!) Beispiele können positiv bewertet werden.
- Die Ausgabe sollte übersichtlich und nachvollziehbar gestaltet sein. Idealerweise wird die Anzahl der nötigen Fahrten mit ausgegeben – am besten zuerst. Ausführliche Angaben zu Fahrten und Zuständen in der Ausgabe mögen interessant sein, eine längliche Ausgabe kann aber auch schwer nachvollziehbar sein. Die Veränderungen zwischen zwei aufeinanderfolgenden Zügen sollten leicht zu erkennen sein, damit auch die Gültigkeit der durchgeführten Fahrten gut zu überprüfen ist.

Aufgabe 2: Panorama-Kegeln

2.1 Einleitung

Die Aufgabenstellung beschreibt ein Spiel: N Kegel stehen auf einer Kreisscheibe mit Radius R. Die Spieler werfen abwechselnd eine Kugel mit festem Radius 1 und versuchen, langfristig möglichst viele Kegel zu treffen. N und R sind die Spielparameter, für die die Werte $N = 20$ und $R = 2$ initial vorgegeben sind. Ein Spieler ist menschlich, der Gegenspieler (genannt Randy) ist eine zufällig spielende KI, die in jeder Spielrunde anfangen darf.

Die Lösung dieser Aufgabe besteht prinzipiell aus zwei Teilen:

1. Das Spiel soll als Programm implementiert werden, einschließlich des Gegners Randy, wobei ein besonderes Augenmerk auf eine effiziente und einfache gleichmäßige Verteilung der Kegel auf eine Kreisscheibe und auf eine bequeme Bedienbarkeit des Spiels durch den menschlichen Spieler gelegt werden soll.
2. Es soll eine Strategie gefunden werden, die auf lange Sicht bei bestimmten Spielparametern gegen Randy (eine zufällig spielende KI, die immer anfangen darf) gewinnt. Anschließend soll der Erfolg der Strategie bei anderen Spielparametern ermittelt werden.

Der zweite Teil der Aufgabe wird sich als ungleich schwieriger erweisen, da eine solche Strategie bei den gegebenen Werte für die Spielparameter einfach nicht existiert und diese Teilaufgabe damit streng genommen nicht lösbar ist. Daher wird von den Teilnehmern erwartet, dass sie diese Umstände erkennen und vernünftig begründen oder belegen, dass eine solche Strategie nicht existiert und nicht nur die eigene Strategie zu schlecht ist. Außerdem sollen sie (einigermaßen ausführlich) untersuchen, ob und inwiefern andere Spielparameter geeignet sind, um Gewinnstrategien zu finden.

Zunächst seien aber die im Folgenden verwendeten Begriffe definiert:

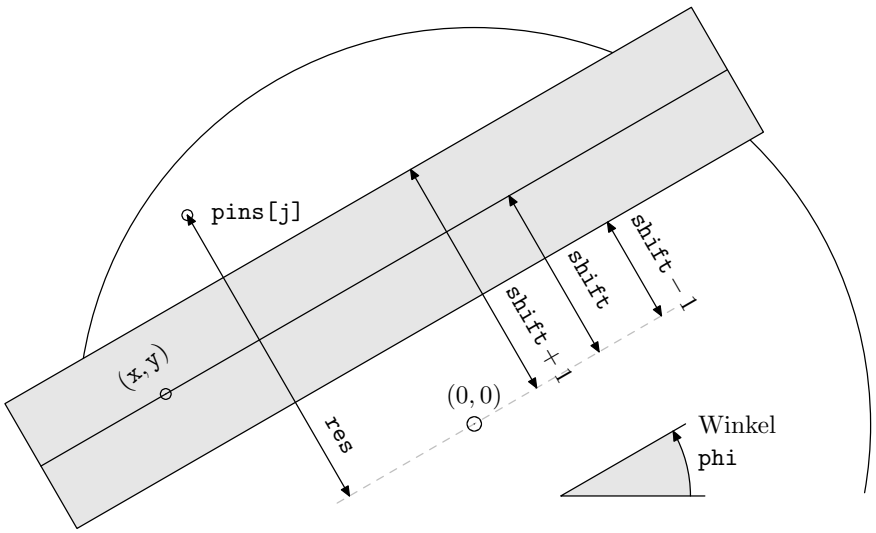


Abbildung 1: Zur Veranschaulichung der Konventionen und des Spielfelds

- Das *Spielfeld* ist ein Kreis mit Radius R, auf dem zu Beginn einer Runde N Kegel gleichmäßig verteilt aufgestellt werden. Die beiden Zahlen N und R bilden die oben genannten Spielparameter.
- Die *Wurfkugel* hat stets einen Radius von $r = 1$ und muss auch nicht variiert werden, da lediglich das Verhältnis R/r relevant ist.
- Der Mittelpunkt des Spielfelds ist der Punkt $(0,0)$. Natürlich sind hier bei den Teilnehmern auch andere Konventionen zu erwarten und zulässig.
- Ein *Wurf* besteht aus der Angabe eines Punktes (x,y) und eines Winkels φ bezüglich der x-Achse, unter dem das Zentrum der Kugel den Punkt passiert (siehe Abb. 1). Es reicht dabei, Winkel im Intervall $[0,180^\circ)$ zuzulassen, da Würfe unter den Winkeln φ und $\varphi + 180^\circ$ zum gleichen Ergebnis führen. Tatsächlich würde es statt der Angabe eines Punktes im Feld auch ausreichen, beispielsweise nur die Ordinate der Wurfbahn anzugeben. Der Übersichtlichkeit der Rechnungen zuliebe gehen wir aber davon aus, dass tatsächlich ein Punkt gegeben ist.

2.2 Verteilung der Kegel

Es gibt viele verschiedene Möglichkeiten, um N Kegel gleichmäßig auf einer Kreisscheibe mit Radius R zu verteilen. Im Folgenden werden drei Methoden vorgestellt.

Monte-Carlo-Methode

Die wohl einfachste Methode, um die Kegel zu verteilen, ist das wiederholte Ziehen eines Punktes (x,y) aus dem Quadrat $[-R,R] \times [-R,R]$. Anschließend wird getestet, ob der Punkt in dem Kreis liegt ($x^2 + y^2 \leq R^2$) und gegebenenfalls verworfen und solange neu gezogen, bis diese Bedingung erfüllt ist. In Python-Code sieht das so aus:

```
1 def randomPoint(r):
2     while True:
3         x,y = (random.uniform(-r, r), random.uniform(-r, r))
4         if x*x + y*y <= r*r:
5             return x,y
```

Diese Methode ist, was die Effizienz angeht, gerade noch in Ordnung. Besser (und eventuell mit Bonuspunkten zu belohnen) ist jedoch folgende:

Radius-Winkel-Methode

Etwas mehr Überlegung erfordert es, wenn man den Radius und Winkel (bezüglich der x-Achse) zufällig bestimmen möchte. Der naive Ansatz, den Radius dabei gleichverteilt aus dem Intervall $[0,R]$ zu ziehen, führt zu einer Verteilung, bei der deutlich zu viele Kegel im Zentrum der Scheibe liegen (Abb. 2, Mitte). Stattdessen muss der Radius mit der Wurzelfunktion gewichtet werden. Zum Verständnis des Gebrauchs der Wurzel: Der Teilkreis mit

Radius $R/2$ hat nur ein Viertel der Fläche des gesamten Kreises mit Radius R. Daher sollte auch die Wahrscheinlichkeit, ein zufälliges $r \leq R/2$ zu ziehen, ein Viertel betragen. Mit der falschen Berechnung wäre sie aber $1/2$, deshalb wären die Punkte in der Mitte zu dicht beieinander. In Python-Code:

```
1 def randomPoint(r):
2     radius = sqrt(random.uniform(0, r*r))
3     phi    = random.uniform(0, 2*pi)
4     return cos(phi)*radius, sin(phi)*radius
```

Quantil-Methode

Die Holzhammer-Methode mit zweifelhaftem Erfolg: Möchte man zunächst die x-Komponente des Punktes bestimmen und erst anschließend die y-Komponente, muss man erstere gemäß der Verteilungsdichte $\rho(x) = \frac{2}{\pi R^2}\sqrt{R^2 - x^2}$ bestimmen. Hat man aber nur Zugriff auf einen gleichverteilenden Zufallsgenerator, benutzt man üblicherweise die Methode der Quantilen. Dazu muss die Umkehrfunktion $I(z) = R^{-1}(z)$ der Stammfunktion $R(x) = \int_0^x \rho(x')\mathrm{d}x'$ der Verteilungsdichte bestimmt werden. Anschließend kann man die gesuchte Zufallsvariable x bestimmen, indem ein gleichverteiltes z aus $\left[-\frac{1}{2}, \frac{1}{2}\right]$ (das Intervall ist abhängig von der Integrationskonstanten) gezogen und x über $x = I(z)$ bestimmt wird.

Das Problem an der Methode: Es gibt keinen analytischen Ausdruck für $I(z)$ bei dieser Wahl von ρ. Stattdessen kann man sie als Taylor-Reihe approximieren und erhält bis zur 6. Ordnung genau

$$I(z) = \frac{R\pi}{2} \cdot (z + \frac{\pi^2}{24}z^3 + \frac{13\pi^4}{1920}z^5).$$

Schließlich wird bei bekanntem x das zugehörige y gleichverteilt zwischen $-\sqrt{R^2 - x^2}$ und $\sqrt{R^2 - x^2}$ bestimmt. In Python-Code sieht das schließlich so aus:

```
1 def I(x,r):
2     return r*(pi/2*x + 1/6.0*(pi/2)**3 * x**3
3             + 13.0/(5*4*3*2) * (pi/2)**5 * x**5)
4
5 def randomPoint(r):
6     z = random.uniform(-0.5,0.5)
7     x = I(z,r)
8     y = random.uniform(-sqrt(r*r-x*x), sqrt(r*r-x*x))
9     return x,y
```

Durch die Näherung ist das Ergebnis aber nicht sehr gut (siehe Abb. 2, rechts). Dies lässt sich zwar durch höhere Ordnung der Näherung verbessern, in jedem Fall entspricht diese Möglichkeit aber nicht der Voraussetzung der Einfachheit und sollte mit Abzug bestraft werden.

2.3 Implementierung der Simulation

Bei der Implementierung der Simulation sind zwei Aspekte besonders interessant: Wie bestimmt man die durch einen Wurf umgeworfenen Kegel und was ist darunter zu verstehen,

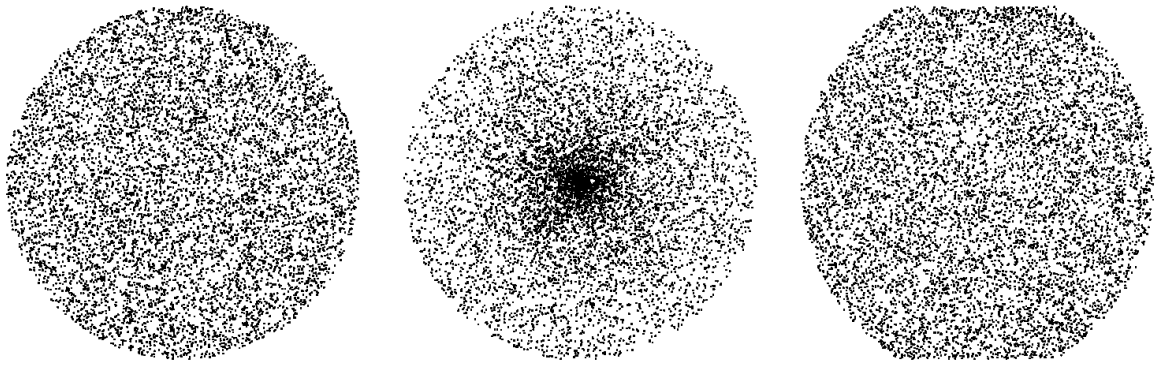


Abbildung 2: Zufallsverteilungen. Links: Korrekte Verteilung, Mitte: Nicht gewichtete Radius-Winkel-Methode, Rechts: Quantil-Methode

dass man möglichst bequem spielen kann, wie es in der Aufgabenstellung gefordert wird? Zunächst zur Bequemlichkeit:

- Da es bei der Simulation darum geht, auf lange Sicht mehr Punkte als der Gegenspieler zu sammeln, soll man mehrere aufeinanderfolgene Runden spielen und den Punktestand mitverfolgen können.
- Der Spielstand der aktuellen Runde soll jederzeit, der über alle Runden akkumulierte Stand zumindest nach jeder Runde angezeigt werden. Die Aufgabenstellung spricht eindeutig davon, dass nicht die Anzahl der Siege, sondern die über viele Runden gesammelten Punkte (als Zahl umgeworfener Kegel) relevant ist.
- Die Eingabe der Wurfparameter, also Punkt und Winkel, kann entweder mit der Maus auf einem gezeichneten Spielfeld oder mittels Eingabe der Daten in ein Textfeld o.Ä. erfolgen. Es darf die Möglichkeit, die Runde nach dem Wurf zu beenden, nicht vergessen werden. Es reicht aus, wenn der Winkel nur als ganzzahlige Gradangabe angegeben werden kann, da die Simulation auf der Wettbewerbsplattform auch nur mit ganzzahligen Winkeln zurechtkommt.
- In jedem Fall muss vor der Bestätigung des Wurfs erkennbar sein, welche Kegel umgeworfen würden.
- Der Wurf des KI-Gegners soll gut nachvollziehbar sein. Es reicht also nicht aus, dass einfach Kegel verschwinden, es müssen zumindest für einige Zeit die Wurfbahn oder die getroffenen Kegel markiert werden.

Zur Bestimmung der durch einen Wurf betroffenen Kegel bieten sich zwei Methoden an:

Kegel-Erfassung durch Vektorrechnung

Das Problem der Bestimmung der umgeworfenen Kegel lässt sich auf das Problem der Bestimmung des Abstands eines Punktes (Kegelposition) zur Wurfbahn zurückführen. Die Wurfbahn lässt sich unter Angabe eines Punktes $\vec{p}$ und eines Winkels φ wie folgt beschreiben:

$$\vec{x}(\lambda) = \vec{p} + \lambda\vec{t}$$

mit $\vec{t} = (\cos\varphi, \sin\varphi)^{\mathrm{T}}$, $|\vec{t}| = 1$. Sei nun die Position $\vec{q}$ eines Kegels gegeben und der Abstand zur Wurfbahn soll bestimmt werden (siehe Abb. 1). Gesucht ist also λ_0, sodass der Abstand zwischen $\vec{x}(\lambda_0)$ und $\vec{q}$ minimal ist. Dies ist offenbar dann der Fall, wenn der Verbindungsvektor $\vec{q} - \vec{x}(\lambda_0)$ senkrecht auf der Wurfbahn steht, also

$$(\vec{q} - \vec{x}(\lambda_0)) \cdot \vec{t} = 0,$$

wobei $\cdot$ das Skalarprodukt bezeichnet. Umgeformt nach λ_0 erhält man

$$\lambda_0 = (\vec{q} - \vec{p}) \cdot \vec{t} \equiv \vec{d} \cdot \vec{t}$$

mit $\vec{d} = \vec{q} - \vec{p}$. Der Abstand l zur Wurfbahn entspricht jetzt der Länge des Verbindungsvekors $\vec{q} - \vec{x}(\lambda_0)$, also

$$\begin{aligned} l^2 &= (\vec{q} - \vec{x}(\lambda_0))^2 = \left(\vec{d} - \lambda_0 \vec{t}\right)^2 \\ &= \vec{d}^2 - 2\lambda_0(\vec{d} \cdot \vec{t}) + \lambda_0^2 \\ &= \vec{d}^2 - (\vec{d} \cdot \vec{t})^2. \end{aligned}$$

Sowohl $\vec{t}$ als auch $\vec{d} = \vec{q} - \vec{p}$ sind bekannt und der Abstand lässt sich berechnen. Ein Kegel wird genau dann getroffen, wenn $l \leq 1$, da der Radius der Wurfkugel 1 beträgt.

Eine Implementierung in Python könnte wie folgt aussehen:

```
1  def dot(p1, p2):
2      return p1[0]*p2[0] + p1[1] * p2[1]
3
4  def findPins(pins, phi, x, y):
5      ans = []
6      t = (cos(phi), sin(phi))
7      for j in range(0, len(pins)):
8          q = (pins[j].x, pins[j].y)
9          d = (p[0] - q[0], p[1] - q[1])
10         distance = sqrt(dot(d, d) - dot(d,t)**2)
11         if distance <= 1: ans.append(j)
12     return ans
```

Die Methode `findPins` gibt dabei die Liste `ans` zurück mit den Indizes aller betroffenen Kegel, die in der Bahn liegen. Sind zum Beispiel drei Kegel betroffen und in `pins` an 3., 8. und 11. Stelle gespeichert, so ist `ans` die Liste `[2,7,10]`.

Kegel-Erfassung durch Drehung

Alternativ kann auch durch Drehung der Koordinaten der Kegel ermittelt werden, ob sie getroffen werden. Dazu werden bei einem Wurf unter dem Winkel φ alle Koordinaten um den Winkel φ im Uhrzeigersinn gedreht (bzw. um den Winkel $-\varphi$ gegen den Uhrzeigersinn). Abb. 1 hilft beim Verständnis. Man stelle sich vor, dass die Kreisscheibe samt aller Kegel und Laufbahn um φ *zurück*gedreht wird. Nun liegt die Bahn waagerecht und die x-Koordinaten

von $\vec{p}$ und $\vec{q}$ spielen keine Rolle mehr, nur die y-Werte, welche sich maximal um 1, den Radius der Kugel, unterscheiden dürfen, damit der Kegel in der Bahn liegt.

Die im Uhrzeigersinn gedrehten Koordinaten (x', y') bestimmt man aus den ursprünglichen (x, y) so:

$$\begin{aligned} x' &= x\cos\varphi - y\sin\varphi \\ y' &= x\sin\varphi + y\cos\varphi \end{aligned}$$

In Python-Code ließe sich das beispielsweise so implementieren:

```
1 def findPins(pins, phi, x, y):
2     ans = []
3     shift = x*math.sin(phi) + y*math.cos(phi)
4     for j in range(0, len(pins)):
5         res = pins[j].x*math.sin(phi) + pins[j].y*math.cos(phi)
6         if shift-1 <= res <= shift+1: ans.append(j)
7     return ans
```

Alternative Methoden

Natürlich sind auch andere Methoden erlaubt und denkbar. Beispielsweise kann die Wurfbahn und deren obere und untere Begrenzung der Reichweite als lineare Funktionen dargestellt werden. Ein Kegel ist dann getroffen, wenn er zwischen den Begrenzungen liegt.

2.4 KI-Strategien

Schlussendlich soll Anna durch eine KI ersetzt werden, die langfristig gegen Randy gewinnt. Randy wirft zwar zufällig, darf aber stets anfangen. Es stellt sich also zunächst die Frage, wie viele Kegel Randy bei gegebenem Spielfeldradius R im ersten Zug durchschnittlich umwirft. Dazu muss der Mittelwert der durch einen Wurf überdeckten Flächen über alle möglichen Würfe gebildet werden. Dabei reicht es aufgrund der Symmetrie aus, waagerechte Würfe zu betrachten.

Trägt man diesen Flächenmittelwert gegen den Radius R auf, ergibt sich das Diagramm in Abb. 3.

Der in der Aufgabe gegebene Radius liegt also sehr knapp unter der 50 %-Marke, tatsächlich beträgt der Wert etwa 49.66 %. Allein das lässt es fraglich erscheinen, ob eine Gewinnstrategie existiert. Bei Radien kleiner als $R \approx 1.984$ liegt die von Randy erhaschte Fläche bei über 50 % und eine Gewinnstrategie kann nicht existieren.

Ob Anna bei $R = 2$ überhaupt gewinnen kann, hängt tatsächlich von der Anzahl der Kegel N ab.

- Ist N klein genug, sodass Anna sehr häufig in ihrem ersten Zug alle der durchschnittlich 50.34 % übrigen Kegel erwischen kann, so kann sie langfristig gewinnen.

- Ist N zu groß, kann Anna in nur wenigen Fällen genügend übrige Kegel abräumen, bevor Randy wieder am Zug ist.

Bei welchem N genau diese Grenze liegt, hängt von Annas Strategie ab. Im folgenden werden einige Strategien vorgestellt. Keine dieser Strategien besteht jedoch bei $N = 20$, der in der Aufgabenstellung genannten Kegelzahl. Es ist daher mit an Sicherheit grenzender Wahrscheinlichkeit keine Strategie in der Lage zu gewinnen.

Randy*100

Diese Methode ermittelt eine bestimmte Anzahl von Randy-artigen Zufallswürfen und wählt von diesen denjenigen aus, der am meisten Kegel umwirft.

Die folgende Funktion `findBest` liest dazu die Kegel als Liste `pins` ein und berechnet `amount` viele Zufallsbahnen, wählt davon die beste aus und gibt die Bahnparameter sowie ein Array der Indizes aller Kegel zurück, welche durch die Bahn umgehauen werden. Zur Berechnung einer zufälligen Bahn ziehen wir einen zufälligen Winkel $\varphi \in [0, \pi]$ und einen Punkt (x, y) auf der Kreisscheibe mittels einer der oben vorgestellten `randomPoint`-Methoden; die jeweils betroffenen Kegel werden mithilfe einer der oben erläuterten `findPins`-Methoden ermittelt.

Die Indizes der getroffenen Kegel der aktuell berechneten Bahn werden in `curBahn` gespeichert. Sobald ihre Länge größer ist als die der besten bisher Gefundenen (`bestBahn`), wird natürlich `bestBahn` gleich der neuen besseren Bahn `curBahn` gesetzt. Die Funktion zur Berechnung einer Listenlänge ist `len`.

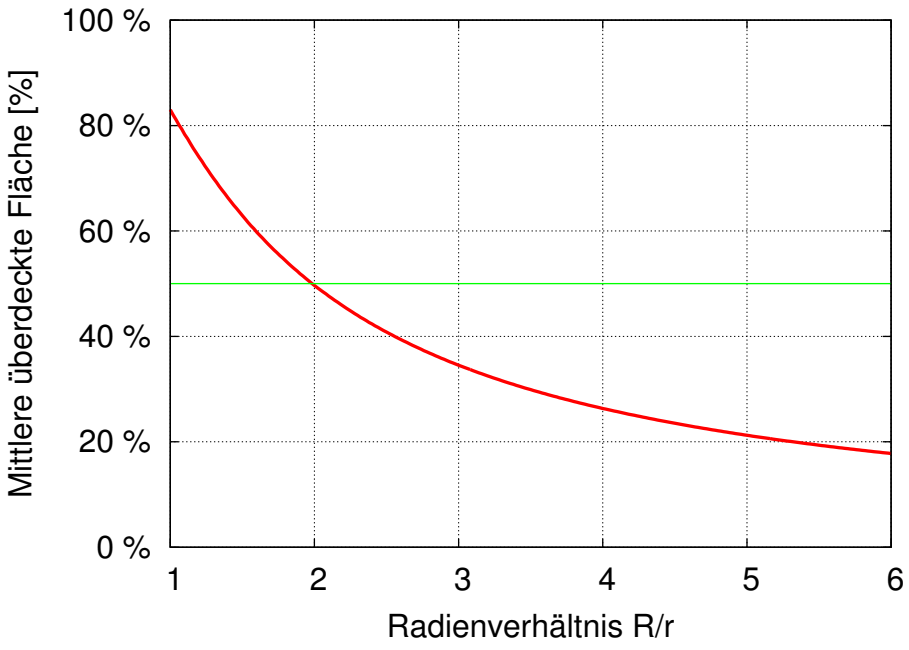


Abbildung 3: Die Fläche, die von Randys zufälligem Zug überdeckt wird (rot). In grün ist die 50 %-Marke eingetragen.

```
1 def findBest(pins, amount):
2     bestBahn = curBahn = []
3     for i in range(0, amount):
4         phi = random.uniform(0, math.pi)
5         x, y = randomPoint(r)
6         curBahn = findPins(pins, phi, x, y)
7         if len(curBahn) > len(bestBahn):
8             bestBahn, bestX, bestY, bestPhi = curBahn, x, y, phi
9     return bestBahn, bestX, bestY, bestPhi
```

Die Laufzeit dieser Methode ist bei k Zufallswürfen und N Kegeln durch $\mathbf{O}(k \cdot N)$ beschränkt.

MaxPins

Mit dieser Methode können wir eine bzw. *die* Wurfbahn finden, die die maximale Anzahl an Kegeln umwirft. Dazu nutzt sie folgendes aus: Hat man eine beliebige Laufbahn mit bestimmten Kegeln im Visier, kann man sie seitlich verschieben, bis einer der Kegel am Bahnrand ist. Nun wird die Laufbahn an diesem Kegel gedreht bis noch ein anderer Kegel ebenfalls an den Rand gelangt. Die Menge und Anzahl von Kegeln im Visier ist gleich geblieben. Deshalb gibt es eine (perfekte) Bahn immer in der Variante, bei der zwei Kegel am Rand sind. Eine Ausnahme ist der Fall, wenn nur ein Kegel vorhanden ist oder nur zwei, deren Abstand zueinander kleiner als 2 ist. Abb. 4 zeigt Beispiele für diese Fälle.

Nun zur Methode: Alle möglichen Paare von Kegeln (jeweils 2) werden gewählt. Da es $\binom{N}{2} = \frac{N^2-N}{2}$ Möglichkeiten gibt, wird dies auch so oft durchgeführt. Seien die Positionen der Kegel (x_1, y_1) und (x_2, y_2), und sei $d = \sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$ der Abstand zwischen ihnen. Wir suchen *die* vier Bahnen, in denen diese beiden Kegel am Rand liegen; im Falle von $d < 2$ nur zwei. Bei der Implementierung muss beachtet werden, dass Kegel auf dem Rand der Bahn aufgrund der Rechenungenauigkeiten nicht zuverlässig getroffen werden, daher sollten diese Kegel gesondert behandelt werden.

Dazu werden Punkt $\vec{p}$ und Winkel φ berechnet, durch die die Bahn bestimmt ist.

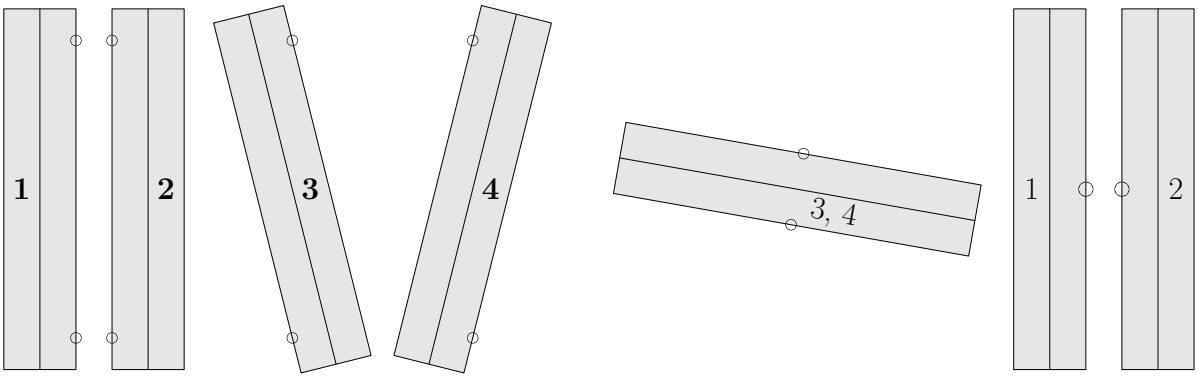


Abbildung 4: Die möglichen Fälle der Bahnen mit zwei Randkegeln.

Sei hierzu $\vec{d} = (x_1 + x_2, y_1 + y_2)^{\mathrm{T}}/2$ der Punkt zwischen den Kegeln, und α der Winkel der Geraden, die die Kegel verbindet. Bei dessen Berechnung ist Vorsicht geboten. Falls beide Koordinaten verschieden sind, also $x_1 \neq x_2 \wedge y_1 \neq y_2$, ist $\alpha = \tan^{-1}((y_1 - y_2)/(x_1 - x_2))$. Ansonsten müssen wir die Division durch 0 vermeiden. Sollten beide Punkte identisch sein, ist der Winkel beliebig, wir setzen $\alpha = 0$. Falls nur $x_1 = x_2$, ist $\alpha = \pi/2$, die Bahn steht senkrecht zur x-Achse. Die folgende Tabelle gibt die Berechnung von $\vec{p}$ und φ für die vier Fälle an:

Fall	Bedingung	Punkt $\vec{p}$	Winkel φ
1	$d \geq 0$	$\vec{r} + (-\sin\alpha, \cos\alpha)^{\mathrm{T}}$	α
2	$d \geq 0$	$\vec{r} + (\sin\alpha, -\cos\alpha)^{\mathrm{T}}$	α
3	$d \geq 2$	$\vec{r}$	$\alpha + \sin^{-1}(2/d)$
4	$d \geq 2$	$\vec{r}$	$\alpha - \sin^{-1}(2/d)$
5		*der* einzige Kegel	*beliebig* bzw. 0

Man könnte eine Berechnung der Wurfbahn überspringen, wenn die Positionen beider Kegel gleich sind, doch ist dies wegen der äußerst geringen Wahrscheinlichkeit, die eigentlich gegen 0 geht, nicht nötig. Die Bahnen der Fälle 1 und 2 können stets berechnet werden.

Ein interessanter Fall ist $d = 2$, der stets $d \geq 2$ erfüllt. Die Winkel der Laufbahnen des 3. und 4. Falls unterscheiden sich dann um 180° bzw. π, was letztendlich heißt, dass die Bahnen identisch sind.

Es sei nun kurz erläutert, wie die Berechnung der Bahnparameter zu begründen ist. In den Fällen 1 und 2 verläuft die Bahn offensichtlich parallel zu der Geraden g, die die Kegel verbindet, daher ist $\varphi = \alpha$. In den Fällen 3 und 4 schneidet die Bahn offensichtlich g in der Mitte, weshalb sich der Schnittpunkt $\vec{r}$ als Wurfpunkt der Bahn eignet.

In den Fällen $1,2$ muss der Wurfpunkt um 1, den Radius der Wurfkugel, in die Richtung verschoben werden, welche senkrecht zu g ist. Der zugehörige zweidimensionale Vektor mit Länge 1 und Winkel α ist $\pm(-\sin\alpha, \cos\alpha)^{\mathrm{T}}$.

In den Fällen $3,4$ muss der Winkel noch angepasst werden.

Hier ist die schräge Linie (Teil von g) mit Länge d die Hypotenuse und die senkrechte Linie die Gegenkathete (siehe Abb. 5). Es gilt die Regel $\sin\alpha' = \text{Gegenkathete}/\text{Hypotenuse}$. In unserer Vorstellung können wir beide Längen durch d teilen und kommen so auf das rechte Teilbild. Aus $2/d = \sin\alpha'$ folgt $\alpha' = \sin^{-1}(2/d)$, was wir zu α addieren bzw. von α subtrahieren.

Sollte es auf der Kreisscheibe nur einen Kegel geben, ist klar, dass es immer eine Bahn gibt, die ihn trifft. Daher müssen wir gar nicht rechnen, sondern geben in der Funktion als Liste der Indizes im Falle von 0 oder 1 Kegel entsprechend `[]` oder `[0]` zurück. Für die Simulation wäre im Falle nur eines Kegels Sonderfall 5 als Laufbahn passend.

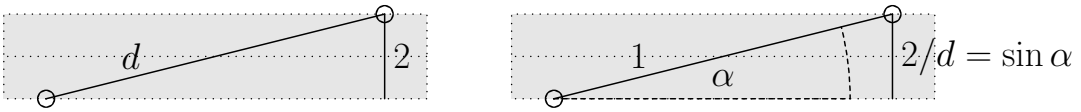


Abbildung 5: Zur Berechnung des Winkels α.

Diese KI sieht in Python-Code so aus:

```
def findBest(pins):
    if len(pins)==1: return [0], pins[0].x, pins[0].y, 0
    if len(pins)==0: return [], 0, 0, 0
    curBahn = bestBahn = []
    # Durchgang aller Kegelpaare a,b mit 0<=a<b<len(pins)
    for a in range(0, len(pins)-1):
        for b in range(a+1, len(pins)):
            # Berechnung von d, Vektor r, hier r=(px,py)^T
            d = (pins[a].x - pins[b].x) ** 2
              + (pins[a].y - pins[b].y) ** 2
            d = math.sqrt(d)
            px = (pins[a].x + pins[b].x)/2
            py = (pins[a].y + pins[b].y)/2
            # Berechnung von alpha, Vorsicht wegen Ausnahmefällen
            alpha = 0
            if pins[a].x == pins[b].x:
                if not pins[a].y == pins[b].y: alpha = math.pi/2
            else:
                alpha = (pins[a].y-pins[b].y)/(pins[a].x-pins[b].y)
                alpha = math.atan(alpha)
            # verschiedene phi entsprechend bis Fall 2 oder 4
            phi = [alpha, alpha]
            if d>=2:
                phi = [alpha, alpha,
                       alpha+math.asin(2.0/d),
                       alpha-math.asin(2.0/d)]
            # Vektor p, hier p=(x,y)^T für alle 4 Fälle
            x = [px-math.sin(alpha), px+math.sin(alpha), px, px]
            y = [py+math.cos(alpha), py-math.cos(alpha), py, py]
            for i in range(0, 4):
                if i==2 and d<2: break
                curBahn = findPins(pins, phi[i], x[i], y[i])
                # Randpunkte wegen Rechenungenauigkeit prüfen
                if not a in curBahn: curBahn.append(a) # ob sie
                if not b in curBahn: curBahn.append(b) # dabei sind
                if len(curBahn) > len(bestBahn):
                    bestBahn = curBahn
                    bestX, bestY, bestPhi = x[i], y[i], phi[i]
    # wenn a, b am Ende hinzugefügt wurden, ist bestBahn nicht
    bestBahn.sort() # mehr sortiert, soll es aber sein
    return bestBahn, bestX, bestY, bestPhi
```

Da für jedes Paar von Punkten für alle N Kegel geprüft werden muss, ob sie von einer der vier Bahnen getroffen werden, und da es $\frac{N^2-N}{2}$ Paare gibt, ist die Laufzeit durch $\mathbf{O}(N \cdot \frac{N^2-N}{2}) \subset \mathbf{O}(N^3)$ beschränkt. Diese Methode ist somit deutlich langsamer als die erste.

Smarty

Man kann die Strategie um eine offensive Komponente erweitern: Wir wählen eine Bahn, welche K_1 Kegel umwirft und ermitteln anschließend die Anzahl K_2 der umgeworfenen Kegel der besten Bahn, die der andere Spieler finden kann, und versuchen die Differenz $|K_1 - K_2|$ zu maximieren. Dazu muss die MaxPins-Strategie nur so angepasst werden, dass beim Ermitteln der Punktzahl einer Bahn die Punktzahl des folgenden besten Zugs auf einem Spielbrett ohne die betreffenden Kegel abgezogen wird.

Alternativ kann man auch die Randy*100-Methode verwenden und wieder k zufällige Würfe ermitteln und anschließend den besten darauffolgenden Zug ermitteln, was bei einer großen Anzahl von Pins eine vernünftige Vereinfachung wäre.

Von besonderem Interesse ist hier, mit welchem Zug des Gegners gerechnet wird. Entwickelt man eine Strategie gegen einen intelligenten Spieler, sollte man von dem Zug ausgehen, der die meisten Kegel umwirft oder man setzt diese Idee rekursiv fort, wenn man davon ausgeht, dass der Gegner die gleiche Strategie verwendet. Um das steuern zu können, führen wir einen weiteren Parameter `recurse` ein, der angibt, ob die maximale Gegnerpunktzahl berücksichtigt werden soll oder nicht. Der Python-Quellcode der MaxPins-Strategie muss dann so angepasst werden:

```
1  def findBest(pins, recurse=True):
2      bestPoints = -N
3      [...]
4      curBahn = findPins(pins, deg[i], x[i], y[i])
5      if recurse:
6          #Umgeworfene Pins wegnehmen
7          newBahn = list(set(pins) - set(curBahn))
8          #K1-K2 berechnen
9          newPoints = len(curBahn)
10                     - len(findBest(newBahn, False)[0])
11     else:
12         newPoints = len(curBahn)
13     if newPoints > bestPoints:
14         bestBahn, bestPoints = curBahn, newPoints
15         bestX, bestY, bestPhi,  = x[i], y[i], deg[i]
16     [...]
```

Diese Variante nennen wir Smarty(Best). Spielt man allerdings gegen Randy, sollte man besser die durchschnittliche Punktzahl des Gegners ermitteln, etwa wieder durch 100 Zufallswürfe. Diese Variante wird im Folgenden als Smarty(Average) bezeichnet.

Benutzt man die MaxPins-Methode, ist die Laufzeit hier durch $\mathbf{O}(N^5)$ beschränkt, verwendet man stattdessen Randy*100, beträgt die kombinierte Laufzeitschranke $\mathbf{O}(k^2N^2)$. Auch hybride Methoden sind denkbar, bei der der beste Zug des Gegners mit jeweils der anderen Methode bestimmt wird. In dem Fall beträgt die Laufzeitschanke $\mathbf{O}(kN^4)$. Welche Methode letztendlich am sinnvollsten ist, hängt von der Wahl der Startparameter ab. Bei $R = 2$ hat Randy relativ gute Chancen auf Kegel, deshalb ist in diesem Fall die offensive Strategie sicher einen Versuch wert. Die Frage ist, ob sie sich auch für manch andere Werte lohnt. So ist z.B. bei großem R

und kleinem N die Wahrscheinlichkeit minimal, dass Randy eine zufällig gewählte Laufbahn mit vielen Kegeln trifft.

2.5 Beenden der Runde

Die Aufgabenstellung erlaubt es Anna, die Runde nach ihrem eigenen Zug zu beenden und neu starten zu lassen; auf diese Möglichkeit sind wir bisher noch nicht eingegangen. Dies hat den Grund, dass die Möglichkeit bei den gegeben Spielparametern nicht sinnvoll ist, da die durchschnittliche Rundenlänge zu kurz ist.

Allgemeiner ist es für Anna in erster Näherung dann von Vorteil, die Runde zu beenden, wenn die Differenz zwischen der erwarteten Punktzahl Randys im nächsten Zug und Annas Punktzahl im übernächsten Zug größer ist als die gleiche Differenz nach Neustarten der Runde. Die erste Differenz kann durch eine Simulation von genügend vielen zufälligen Zügen Randys ermittelt werden, die zweite Differenz kann im Prinzip theoretisch bestimmt werden. Als Grundlage dient dafür das Diagramm in Abb. 3. Genauer, aber entsprechend aufwändiger wäre natürlich das Berücksichtigen des überübernächsten, usw. Zuges.

Neben diesem Kriterium, eine Runde zu beenden, gibt es auch viele andere einfachere und kompliziertere. Die Teilnehmerinnen und Teilnehmer sollten die von ihnen gewählte ausreichend begründen.

2.6 Erfolg der Strategien

Im folgenden Abschnitt haben wir verschiedene der oben vorgestellten Strategien bei verschiedenen Startparametern gegen Randy antreten lassen und die Ergebnisse dokumentiert. Getestet wurden

- Randy*100 und MaxPins so wie oben beschrieben,
- Smarty(Best) und Smarty(Average) in Implementierungen, die jeweils 100 Zufallszüge erzeugen und anschließend den besten Wurf (Smarty(Best)) bzw. die durchschnittliche Punktzahl (Smarty(Average)) aus weiteren 100 Zufallszügen des Gegners ermitteln,
- Smarty(Average) in einer Implementierung, in der für alle möglichen Wurfkombinationen jeweils die durchschnittliche Punktzahl aus 100 Zufallszügen des Gegners ermittelt wird. Diese Strategie ist deutlich laufzeitintensiver als die anderen und wurde daher nur in einem eingeschränkten Parameterraum getestet.

Für verschiedene N und R zeigen die folgenden Tabellen den Punktestand nach 200 000 Runden in der Form Punkte Anna/Punkte Randy für die verschiedenen Strategien gerundet auf drei Nachkommastellen. Die Zahl in Klammern gibt dabei die statistische Unsicherheit an, der Eintrag $1.904(13)$ ist zu lesen als 1.904 ± 0.013.

Zahlen unter 1 bedeuten, dass Randy auf lange Sicht gewinnt und sind rot markiert. Zahlen über 1 bedeuten hingegen, dass Anna eine Gewinnstrategie gefunden hat und sind grün eingefärbt. Ist aufgrund der statistischen Unsicherheit keine (2σ)-signifikante Aussage möglich, wurde der Eintrag gelb gesetzt.

Der Einfluss der beiden Parameter wurde oben bereits qualitativ beleuchtet. Interessant ist besonders die erste Zeile für $N = 1$, da diese Werte (ein wenig umgerechnet) genau den theoretischen Vorhersagen über die von Randy durchschnittlich abgedeckte Fläche entsprechen müssen und auch tun, vorausgesetzt Annas Strategie und ihr Koordinationsvermögen sind hinreichend gut um einzelne Kegel sicher umzuschmeißen.

Insgesamt zeigt sich kaum ein Unterschied zwischen den drei zufallsbasierten Strategien. Es hat den Anschein, dass sich besonders ausgeklügelte Strategien hier nicht lohnen, schneidet doch die MaxPins-Strategie besser ab. Dies ist wohl darin zu begründen, dass in den Smarty-Implementierungen aus Laufzeitgründen nicht alle möglichen Würfe durchprobiert, sondern nur 100 zufällige Würfe erwürfelt werden. Diese Vereinfachung hat deutlichen Einfluss auf die Güte der KI.

Auch beim Vergleich zwischen MaxPins und der klügeren Smarty-Implementierung, die alle Kombinationen durchprobiert, ist kein Unterschied zu sehen. Festzuhalten ist aber, dass bei $R = 2$ und $N = 10$ eine Unterscheidung zwischen den zufallsbasierten und den probierenden Strategien möglich ist; letztere gewinnen in dem Fall.

Randy*100

$N \backslash R$	1	1.5	1.75	1.984	1.99	2	2.01	2.02	2.25	3
1	0.208(14)	0.593(7)	0.800(6)	0.994(7)	1.016(7)	1.017(7)	1.020(7)	1.031(7)	1.228(8)	1.904(13)
2	0.206(10)	0.592(5)	0.800(5)	1.000(5)	1.003(5)	1.017(5)	1.020(5)	1.032(5)	1.233(6)	1.893(9)
3	0.206(8)	0.591(4)	0.796(4)	1.001(4)	1.003(4)	1.011(4)	1.018(4)	1.029(4)	1.227(5)	1.841(8)
4	0.208(7)	0.590(4)	0.800(3)	0.998(4)	1.001(4)	1.006(4)	1.019(4)	1.025(4)	1.215(4)	1.779(6)
5	0.207(6)	0.590(3)	0.796(3)	0.992(3)	0.997(3)	1.003(3)	1.013(3)	1.021(3)	1.199(4)	1.737(6)
10	0.206(5)	0.589(3)	0.795(2)	0.972(2)	0.975(2)	0.980(2)	0.992(2)	0.998(2)	1.151(3)	1.579(4)
20	0.206(3)	0.592(2)	0.784(2)	0.941(2)	0.944(2)	0.949(2)	0.956(2)	0.960(2)	1.095(2)	1.453(2)
50	0.206(2)	0.588(1)	0.767(1)	0.909(1)	0.912(1)	0.917(1)	0.922(1)	0.931(1)	1.041(1)	1.329(2)

MaxPins

$N \backslash R$	1	1.5	1.75	1.984	1.99	2	2.01	2.02	2.25	3
1	0.207(14)	0.593(7)	0.806(6)	0.993(7)	1.006(7)	1.017(7)	1.018(7)	1.035(7)	1.223(8)	1.893(13)
2	0.207(10)	0.591(5)	0.798(5)	1.000(5)	1.006(5)	1.018(5)	1.011(5)	1.027(5)	1.234(6)	1.898(10)
3	0.207(8)	0.590(4)	0.802(4)	0.999(4)	1.005(4)	1.014(4)	1.029(4)	1.029(4)	1.225(5)	1.893(8)
4	0.207(7)	0.590(4)	0.799(3)	1.005(4)	1.007(4)	1.011(4)	1.019(4)	1.030(4)	1.232(4)	1.866(7)
5	0.206(6)	0.590(3)	0.800(3)	1.000(3)	1.007(3)	1.014(3)	1.023(3)	1.033(3)	1.228(4)	1.854(6)
10	0.205(5)	0.592(3)	0.800(2)	0.995(2)	0.999(2)	1.004(3)	1.016(3)	1.023(3)	1.201(3)	1.730(4)
20	0.207(3)	0.590(2)	0.795(2)	0.976(2)	0.980(2)	0.987(2)	0.995(2)	1.002(2)	1.157(2)	1.594(3)
50	0.207(2)	0.590(1)	0.785(1)	0.943(1)	0.950(1)	0.954(1)	0.961(1)	0.968(1)	1.100(1)	1.450(2)

Smarty(Best), 100 Würfe

$N \backslash R$	1	1.5	1.75	1.984	1.99	2	2.01	2.02	2.25	3
1	0.207(14)	0.591(7)	0.796(6)	1.006(7)	1.006(7)	1.005(7)	1.024(7)	1.027(7)	1.227(8)	1.895(13)
2	0.206(10)	0.592(5)	0.800(5)	1.002(5)	1.004(5)	1.017(5)	1.024(5)	1.030(5)	1.230(6)	1.889(9)
3	0.207(8)	0.592(4)	0.799(4)	1.005(4)	1.004(4)	1.012(4)	1.023(4)	1.029(4)	1.227(5)	1.833(7)
4	0.206(7)	0.593(4)	0.799(3)	0.996(4)	1.001(4)	1.011(4)	1.022(4)	1.024(4)	1.214(4)	1.777(6)
5	0.206(6)	0.591(3)	0.798(3)	0.989(3)	0.996(3)	1.006(3)	1.010(3)	1.025(3)	1.204(4)	1.741(6)
10	0.206(5)	0.590(3)	0.793(2)	0.970(2)	0.977(2)	0.983(2)	0.990(2)	1.000(3)	1.148(3)	1.581(4)
20	0.207(3)	0.590(2)	0.782(2)	0.941(2)	0.945(2)	0.951(2)	0.957(2)	0.963(2)	1.097(2)	1.451(2)
50	0.207(2)	0.587(1)	0.768(1)	0.911(1)	0.911(1)	0.918(1)	0.923(1)	0.926(1)	1.041(1)	1.318(2)

Smarty(Average), 100 Würfe

$N \backslash R$	1	1.5	1.75	1.984	1.99	2	2.01	2.02	2.25	3
1	0.207(14)	0.592(7)	0.794(6)	0.995(7)	1.008(7)	1.012(7)	1.019(7)	1.033(7)	1.231(8)	1.897(13)
2	0.207(10)	0.593(5)	0.798(5)	1.003(5)	1.005(5)	1.017(5)	1.019(5)	1.031(5)	1.230(6)	1.893(9)
3	0.206(8)	0.588(4)	0.802(4)	0.998(4)	1.002(4)	1.014(4)	1.021(4)	1.024(4)	1.224(5)	1.852(8)
4	0.207(7)	0.592(4)	0.796(3)	1.000(4)	0.996(4)	1.007(4)	1.019(4)	1.024(4)	1.215(4)	1.800(6)
5	0.206(6)	0.590(3)	0.801(3)	0.996(3)	0.997(3)	1.008(3)	1.016(3)	1.026(3)	1.208(4)	1.743(6)
10	0.206(5)	0.589(3)	0.794(2)	0.973(2)	0.978(2)	0.982(2)	0.990(2)	0.997(2)	1.155(3)	1.591(4)
20	0.206(3)	0.591(2)	0.783(2)	0.945(2)	0.945(2)	0.951(2)	0.959(2)	0.966(2)	1.098(2)	1.458(2)
50	0.206(2)	0.589(1)	0.768(1)	0.909(1)	0.915(1)	0.919(1)	0.925(1)	0.929(1)	1.043(1)	1.333(2)

Smarty(Average), Alle Kombinationen

$N \backslash R$	1.984	1.99	2	2.01	2.02
1	0.997(7)	1.007(7)	1.003(7)	1.016(7)	1.035(7)
2	0.998(5)	1.004(5)	1.013(5)	1.020(5)	1.034(5)
3	1.002(4)	1.004(4)	1.016(4)	1.024(4)	1.033(4)
4	0.999(4)	1.002(4)	1.009(4)	1.018(4)	1.033(4)
5	0.998(3)	1.004(3)	1.011(3)	1.023(3)	1.028(3)
10	0.997(2)	1.000(2)	1.004(3)	1.016(3)	1.025(3)
20	0.976(2)	0.979(2)	0.985(2)	0.994(2)	1.004(2)
50	0.941(1)	0.944(1)	0.953(1)	0.956(1)	0.963(1)

2.7 Erweiterungen

Mehr Dimensionen Wie bei allen Aufgaben, die die Simulation von Spielen beinhalten, lassen sich zahlreiche Erweiterungen durch das Ändern der Spielregeln finden. So kann man beispielsweise das zweidimensionale Spielbrett relativ einfach auf D Dimensionen erweitern. Kegel sind dann Punkte in dem hyperkugelförmigen Spielfeldraum, die mittels angepasster Zufallsfunktionen platziert werden: Sie müssen nun Punkte $(x_1, \ldots, x_D)^{\mathrm{T}}$

aus $\mathbb{R}^D$ ziehen, die der Bedingung $\sum_{i=1}^{D} x_i^2 \leq 1$ gehorchen. Um Würfe zu beschreiben, benötigt man nunmehr einen Punkt und $D-1$ Winkel. Die auf Vektorrechnung basierte Abstandsberechnung lässt sich einfach auf den D-dimensionalen Fall erweitern.

Andere Spielfeldformen Auch interessant sind andere Spielfeldformen. Dies ist recht einfach zu implementieren, indem die Funktion zum Ziehen von Zufallspunkten angepasst wird. Die Strategien müssen natürlich angepasst werden.

Besserer Gegner und Turnier-System Ebenso reizvoll ist es, von einem stärkeren Gegner als Randy auszugehen und spezielle Gegenstrategien zu entwickeln. Das ist vor allem interessant, wenn die KI auch im Turniersystem genutzt wird.

Andere Kegelverteilungen ... gelten als einfache Erweiterung, für die es keine Pluspunkte gibt.

2.8 Bewertungskriterien

Spiel und Simulation

- Die zufällige Verteilung der Kegel muss korrekt, ausführlich dokumentiert und nicht unnötig kompliziert sein. Als zu kompliziert gelten Methoden, die umständlicher als die Monte-Carlo-Methode sind.
- Es wird eine grafische Ausgabe der erzeugten Zufallsverteilung erwartet.
- Das Spiel muss geeignet implementiert sein (vgl. Abschnitt 2.3):
  - Man muss mehrere Runden hintereinander spielen können.
  - Der Spielstand (akkumulierte Punkte) soll regelmäßig angezeigt werden.
  - Die Eingabe der Parameter muss komfortabel genug sein, und man soll vor Bestätigung des Zugs sehen können, welche Kegel betroffen wären.
  - Der Wurf des Gegenspielers soll gut erkennbar sein.
- Im automatischen Modus (KI gegen KI) muss erkennbar sein, welche KI (auf lange Sicht) gewinnt.

KI (Annas Strategie)

- Die Strategie der KI soll ausführlich erläutert werden.
- Die Strategie, nach der die eigene KI entscheidet, ob die Runde nach dem Zug beendet wird, muss erläutert und begründet werden.
- Es muss erkannt werden, dass die Strategie bei den vorgegebenen Werten für die Spielparameter ($N = 20, R = 2$) Randy nicht schlagen kann. Eine Begründung ist ausnahmsweise nicht zwingend erforderlich.

- Die Strategie sollte mindestens so gut sein wie Randy*100, also in der Lage sein, Randy bei $N = 5$ und $R = 2$ zu schlagen; wir sprechen dann von einer *ausreichenden* KI). Eine *gute* KI zeichnet sich dadurch aus, dass sie bei $N = 10$ und $R = 2$ gegen Randy gewinnen kann.
- Die Strategie soll für mindestens fünf verschiedene Spielparameterpaare (N,R) getestet und das Ergebnis dokumentiert werden. Die eigene Strategie sollte dabei mindestens einmal verlieren und einmal gewinnen.
- Ausführliche theoretische Überlegungen, beispielsweise dazu, wie viele Kegel Randy im ersten Wurf umwirft oder wann das Beenden der Runde durch Anna sinnvoll ist, können – je nach Qualität – mit Pluspunkten belohnt werden.

Aufgabe 3: Mississippi

3.1 Teilstrings suchen

Gesucht sind also diejenigen Teile einer Zeichenkette, die darin mindestens mit einer bestimmten Häufigkeit auftreten, mindestens eine gegebene Länge haben und unter den gleich häufigen Teilen maximal im Sinne der Aufgabenstellung sind (also nicht selbst Teil einer anderen gleich häufigen Teilzeichenkette sind).

Grundsätzlich kann man die Suche nach diesen Teilstrings direkt umsetzen: Die Zeichenkette ω wird systematisch durchsucht. Dazu kann man z.B. zwei Zeiger a und b auf Positionen in der Zeichenkette verwalten. Es geht vorne los, a zeigt also auf den Anfang und b auf die Position $a+$Mindestlänge. Den Teilstring von Position a bis Position b (kurz: $\omega(a,b)$ prüft man dann auf Häufigkeit, ggf. auch auf Maximalität und verwaltet gefundene „gute“ Teilstrings so, dass sie bei erneuter Beobachtung nicht wieder neu geprüft werden müssen. b wird zunächst so weit erhöht, bis $\omega(a,b)$ selbst nicht häufig genug ist. Dann geht es wieder von vorne los, aber mit a auf der nächsten Position, usw.

Bei korrekter Beachtung der Bedingungen für Häufigkeit und Maximalität können auf diese Weise durchaus korrekte Ergebnisse gefunden werden. Aber der Aufwand für ein derartiges Verfahren ist hoch und insbesondere umgekehrt abhängig von der Mindesthäufigkeit k: Je kleiner k, desto mehr „gute“ Teilstrings müssen verwaltet und bei der Suche immer wieder neu abgeglichen werden. Die Verarbeitung einer Zeichenkette der Länge 10.000 dauert für kleines k mit einem solchen Vorgehen relativ lang.

Entscheidend für eine erfolgreiche Bearbeitung dieser Aufgabe könnte es also sein, die Teilstrings einer Zeichenkette so in einer Datenstruktur zu organisieren, dass sich daraus gut ablesen lassen kann, welche mit der geforderten Häufigkeit auftreten, die gewünschte Mindestlänge haben und maximal im Sinne der Aufgabenstellung sind.

Es gibt dazu einige Möglichkeiten. Die für den Zweck dieser Aufgabe und auch einige andere Problemstellungen bei der Verarbeitung von Zeichenketten geeignetste Datenstruktur ist der sogenannte Suffixbaum. In dieser Datenstruktur stecken folgende für eine gute Lösung dieser Aufgabe zentrale Ideen:

- Mehrfach in (Teil-)Strings vorkommende Teile werden zusammengefasst, was durch eine Baumstruktur möglich ist. Diese Idee ist z. B. auch in der Datenstruktur *Trie* umgesetzt.
- Nicht verzweigende Pfade in einer zur Abspeicherung von (Teil-)Strings verwendeten Baumstruktur werden zu einem Knoten zusammengefasst. Diese Idee steckt auch im *Patricia-Trie*.
- Für die Suche nach inneren Teilstrings eignet sich die Speicherung von Suffixen (Endstücken) besonders gut (während Tries und Patricia-Tries sich auf Präfixe, also Anfangsstücke konzentrieren). Ein innerer Teilstring einer Zeichenkette ist immer Präfix eines Suffixes der Zeichenkette.

- In einer auf Suffixe orientierten Baumstruktur entspricht ein innerer Knoten einem Teilstring. In diesem inneren Knoten lassen sich die für diese Aufgabe entscheidenden Werte ablegen: Die Häufigkeit des Teilstrings entspricht der Anzahl der Pfade durch diesen Knoten und damit der Anzahl der Blätter „unter“ diesem Knoten (also des Teilbaums, der diesen Knoten als Wurzel hat). Die Länge des Teilstrings lässt sich auf dem Pfad von der Wurzel zum Knoten berechnen, der mit den Teilstücken dieses Teilstrings beschriftet ist.

3.2 Der Suffixbaum

Für ein Wort ω der Länge n ist der zugehörige Suffixbaum ein gewurzelter Baum, mit folgenden Eigenschaften:

1. Die Blätter sind von 0 bis $n-1$ durchnummeriert.
2. Die Kanten sind mit nichtleeren Wörtern beschriftet.
3. Jeder innere Knoten (außer der Wurzel) hat mindestens zwei Nachfolgeknoten.
4. Für einen Knoten gibt es keine zwei ausgehenden Kanten, deren Beschriftung den gleichen Anfangsbuchstaben hat.
5. Hängt man die Beschriftungen der Kanten entlang eines Pfades von der Wurzel zu dem Blatt mit der Nummer i aneinander, erhält man das Suffix, welches mit dem $(i+1)$-ten Buchstaben von ω beginnt, gefolgt von einem Schlusssymbol, z. B. \$, das nicht in ω vorkommt.

Man bemerke, dass der Suffixbaum zu ω bereits durch diese Eigenschaften eindeutig definiert ist.

Abbildung 6 zeigt den Suffixbaum zu dem Beispielwort CAGGAGGATTA (die grauen Zahlen werden erst später verwendet). Betrachtet man den Pfad von der Wurzel zum Blatt Nummer 5, so ist er mit den Teilwörtern G, GA und TTA\$ beschriftet und entspricht dem Suffix GGATTA. Man sieht auch, warum man das Symbol \$ in der Eigenschaft 5 einführt. So ist die Kante zum Blatt Nummer 10 nicht leer.

3.3 Anwendung

Ehe wir uns überlegen, wie wir einen solchen Suffixbaum zu einem gegebenen Wort ω berechnen, wollen wir sehen, wie sich die Datenstruktur einsetzen lässt, um die Aufgabe zu lösen. Grundlegend ist dazu folgende Beobachtung:

Sei ein Knoten v gegeben, dessen Teilbaum h Blätter besitzt, und sei u das Wort, was entsteht, indem man die Beschriftungen entlang des Pfades von der Wurzel nach v aneinander hängt. Dann kommt u genau h-mal als Teilwort in ω vor (wenn wir eventuell das Zeichen \$ weglassen).

Tatsächlich definiert jedes der h Blätter einen Suffix, der an einem anderen Index (der Nummer des jeweiligen Blattes) mit u beginnt. Das Wort u kommt also mindestens h-mal vor.

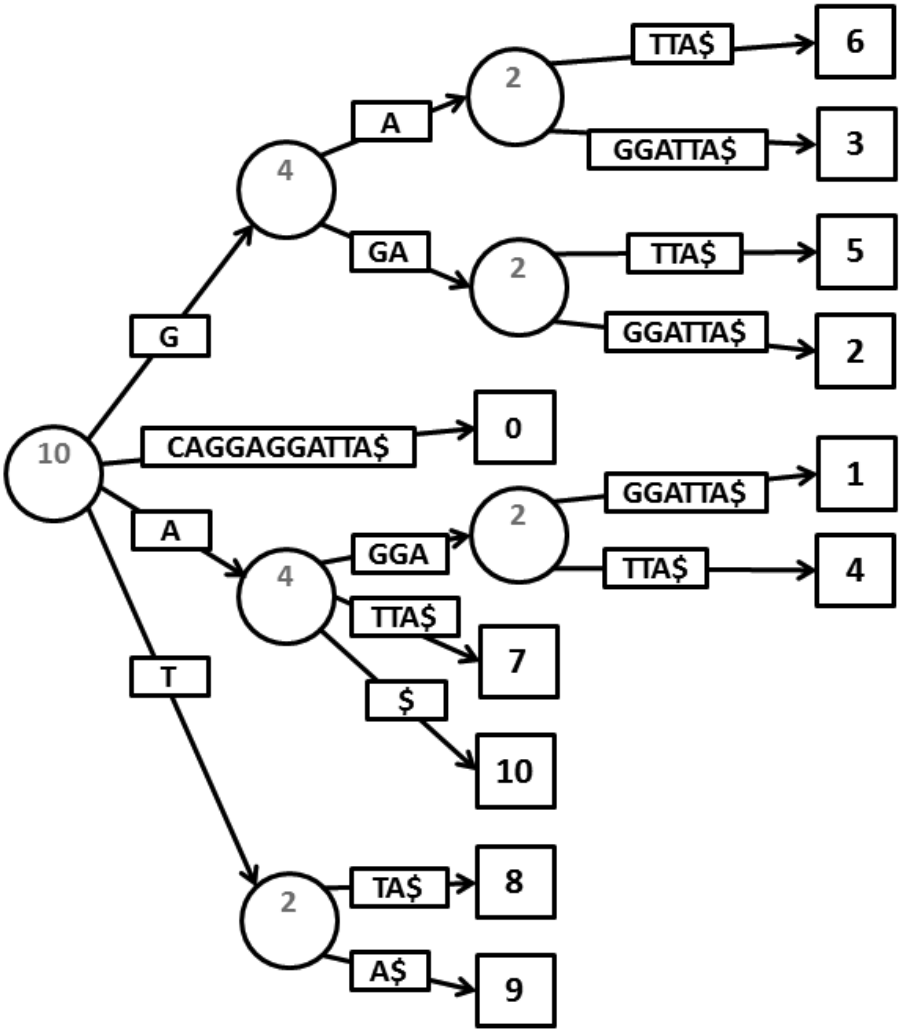


Abbildung 6: Suffixbaum für das Wort CAGGAGGATTA

Für jedes Mal, dass das Wort u an einer Stelle i vorkommt, beginnt auf der Suffix, der an Stelle i beginnt, mit diesem Wort. Also muss der Pfad von der Wurzel nach i durch v verlaufen. Der Teilbaum von v hat also mindestens h Blätter.

In unserem Suffixbaum (Abbildung 6 ist jeder innere Knoten mit der Anzahl der Blätter des entsprechenden Teilbaums beschriftet (graue Zahlen). Wir können nun Teilwörter finden, welche mindestens zweimal in ω auftreten, indem wir Pfaden zu inneren Knoten (ungleich der Wurzel) folgen (der Teilbaum, der in einem solchen Knoten beginnt, hat mindestens 2 Blätter aufgrund der Suffixbaumbedingung 3) und dabei die Kantenbeschriftungen aneinander hängen. In unserem Fall finden wir die Teilwörter: G, GA, GGA, A, AGGA, T Der aufmerksame Leser der Aufgabenbeschreibung wird bemerkt haben, dass drei Teilwörter, welche mindestens zweimal auftreten, fehlen (AG, AGG, GG). Wir merken, dass aber in diesem Beispiel alle maximalen Wörter unter den gefundenen Wörtern sind. Tatsächlich können wir beobachten, dass Wörter, die gefunden werden, folgende stärkere Bedingung erfüllen:

Sei u ein (nichtleeres) Teilwort von ω, sodass es kein Teilwort u' von ω gibt, welches mindestens genau so oft auftritt und u als echten Präfix hat (das heißt u' beginnt mit u gefolgt von mindestens einem weiteren Zeichen). Dann gibt es einen inneren Knoten v ungleich der Wurzel, sodass u durch Aneinanderhängen der Beschriftungen entlang des Pfades von der Wurzel nach v entsteht.

Sei v der Knoten, der am weitesten von der Wurzel entfernt liegt, sodass der zugehörige Teilbaum alle Blätter enthält, welche Suffixen von ω entsprechen, die mit u beginnen. Sei u' das Wort, das aus den Beschriftungen entlang des Pfades von der Wurzel nach v entsteht.

Wenn wir zeigen, dass u' gleich u ist, sind wir fertig. Sei s ein beliebiges Suffix von ω, welches mit u beginnt. Dann beginnt s auch mit u', da der Pfad zu dem zu s gehörigen Blatt durch v läuft. Es gibt also folgende drei Möglichkeiten:

1. u ist gleich u'
2. u ist ein echter Präfix von u'
3. u' ist ein echter Präfix von u

Nach den Forderungen, die wir an u stellen, kann Fall 2 nicht eintreten. Um zu zeigen, dass u' gleich u ist, genügt es also zu zeigen, dass auch Fall 3 nicht möglich ist. Im Fall 3 wäre v ein innerer Knoten. Dann muss v zwei Kinder v_a und v_b besitzen, sodass jeder der entsprechenden Teilbäume ein Suffix s_a und s_b enthält, welches mit u anfängt. a und b bezeichnen die verschiedenen ersten Buchstaben der Kanten von v nach v_a bzw. v_b. Per Definition beginnt dann s_a mit $u'a$ und s_b mit $u'b$, da heißt der längste gemeinsame Präfix dieser beiden Suffixe ist u'. Also ist u' kein echter Präfix von u.

Tatsächlich gilt auch die Umkehrung zu den beiden Beobachtungen:

Sei ein innerer Knoten v ungleich der Wurzel gegeben, dessen Teilbaum h Blätter besitzt und sei u das Wort, was entsteht, indem man die Beschriftungen entlang des Pfades von der Wurzel nach v aneinander hängt. Dann gibt es kein Teilwort u' von ω, welches mindestens h-mal in ω auftritt und u als echtes Präfix hat.

Angenommen das Gegenteil. Wir nehmen an, dass u' mit u beginnt, gefolgt von dem Zeichen a und eventuell noch weiteren Zeichen. Offensichtlich kann u' nicht häufiger als u in ω auftreten. Ferner tritt es genau dann h-mal auf, wenn jedes Suffix, welches mit u beginnt, auch mit u'

beginnt. Das bedeutet aber, dass jede ausgehende Kante von v mit dem Buchstaben a anfängt (sonst gäbe es ein Blatt, welches mit u, nicht aber mit u' beginnt). Da v ein innerer Knoten ungleich der Wurzel ist, widerspricht das der Bedingung 3) und 4) des Suffixbaumes.

Diese Beobachtungen ergeben das nachfolgende, für unseren Algorithmus essentielle, Resultat:

Gegeben seien l und k. Sei S die Menge aller Wörter, die durch Aneinanderhängen der Beschriftungen entlang des Pfades von der Wurzel zu einem Knoten ungleich der Wurzel entstehen. Man entferne aus S alle Wörter, die

1. *weniger als k-mal in ω vorkommen.*
2. *echtes Suffix eines Wortes aus S sind, welches gleich häufig Teilwort von ω ist.*
3. *weniger als l Zeichen besitzen.*

Dann enthält S genau die maximalen Wörter, die mindestens k-mal in ω enthalten sind und mindestens l Zeichen besitzen.

Nach den obigen Beobachtungen ist es klar, dass diese Wörter auch nach Anwenden der Schritte 1), 2) und 3) in S enthalten sind.

Sei wiederum ein Wort u aus S gegeben, welches 1) und 3) erfüllt. Es genügt zu sehen, dass dies maximal ist. Die vorherige Beobachtung sichert uns zu, dass jedes Wort, welches u enthält und ebenso oft in ω vorkommt, u auch bereits als Suffix enthält. Außerdem haben wir gesehen, dass ein maximal langes Wort mit dieser Eigenschaft in S enthalten ist. Ergo muss u bereits maximal sein, wenn es 2) erfüllt.

3.4 Der Algorithmus

Algorithm 1 Berechnung maximaler häufiger Teilzeichenketten

Eingabe: Eine Zeichenkette ω, Zahlen k, l

Ausgabe: Alle maximalen Teilwörter der Länge mindestens l, die mindestens k-mal in ω auftauchen.

Bemerkung: OBdA sei $k \geq 2, l \geq 1$ – Ansonsten müssen wir zusätzlich noch eventuell die gesamte Zeichenkette (falls $k = 1$) bzw. die leere Zeichenkette (falls $l = 0$) ausgeben.

1. Berechne einen Suffixbaum T zu ω
2. Berechne zu jedem Knoten v in T die Anzahl der Blätter k_v des entsprechenden Teilbaums und die Länge l_v des in ihm endenden Wortes.

while T enthält Knoten außer der Wurzel und den Blättern **do**

- 3.1. Sei v ein innerer Knoten mit l_v maximal; unter allen solchen sei k_v minimal
- 3.2. Wenn $k_v \geq k$ und $l_v \geq l$, gib das zu v gehörige Wort aus.
- 3.3. Sei b die Nummer eines Blattes des zu v gehörigen Teilbaumes, $w \leftarrow v$
- **while** $l_w \geq l_v$ und $k_w \geq k_v$ **do**
  - Markiere w, b- -, l_v- -, w sei der Vorgänger vom Blatt Nummer b
- **end while**
- Lösche alle markierten Knoten und hänge ihre Blätter an ihre Vorgänger.

end while

Unser Verfahren ist in Algorithmus 1 angegeben.

Der Algorithmus arbeitet korrekt. Sei S die Menge aller Wörter, die Pfaden von der Wurzel des Suffixbaumes T zu einem inneren Knoten entsprechen. Sei u das zu v gehörige Wort in einer Iteration von 3. Wir bemerken zuerst, dass in 3.3 alle Knoten gelöscht werden, deren zugehörige Wörter Suffixe von u sind, welche nicht häufiger als u in ω auftreten. Weiter sehen wir, dass u kein echtes Suffix von einem Wort aus S sein kann, welches gleichhäufig Teilwort von ω ist, sonst wäre v nach obiger Argumentation in einer vorherigen Iteration markiert und gelöscht worden. Entsprechend werden in 3. genau die Knoten v betrachtet, deren zugehörige Wörter Bedingung 2) aus der letzten Beobachtung im vorherigen Abschnitt erfüllen. In 3.2 werden dann genau die Wörter ausgegeben, die auch 1) und 3) erfüllen. Die Korrektheit ergibt sich dann direkt mit der letzten Beobachtung.

Wir wollen die Laufzeit analysieren. Sei n die Länge von ω, a die Länge, die wir für die Ausgabe benötigen und $s(n)$ die Zeit, die wir für Schritt 1 brauchen.

Dann hat der obige Algorithmus Laufzeit $\mathbf{O}(s(n)+n+a)$. Wir bemerken, dass jeder Binärbaum mit n Blättern und keinem inneren Knoten vom Grad 1 maximal $2n-1$ Knoten besitzt. Entsprechend hat auch unser Suffixbaum T $\mathbf{O}(n)$ Blätter.

Wir analysieren die verschiedenen Teile des Algorithmus getrennt:

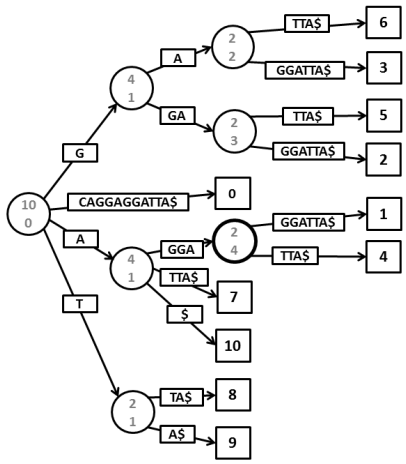
1. Läuft per Definition in Zeit $s(n)$
2. Lässt sich also mit einer Tiefensuche in Linearzeit, d.h. $\mathbf{O}(n)$, umsetzen.

3.1. Funktioniert ebenfalls in Zeit $\mathbf{O}(n)$, indem wir die Knoten vorsortieren. Wir verwenden dabei, dass k_v und l_v nur ganze Werte von 1 bis n annehmen können.

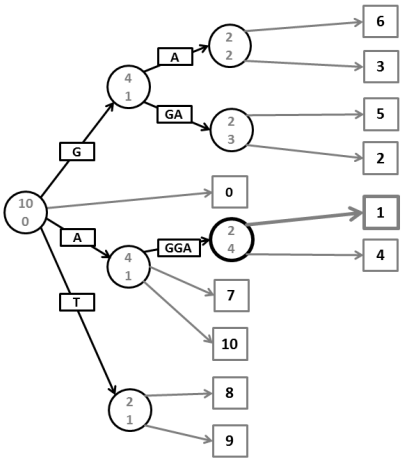
3.2. Hat offensichtlich Laufzeit $\mathbf{O}(n+a)$

3.3. Hat pro Iteration der Schleife konstant viel Laufzeit, plus den Aufwand, den das Markieren und Löschen von Knoten verursacht. Da jeder Knoten nur einmal markiert und gelöscht wird, liegt dieser jedoch in $\mathbf{O}(n)$.

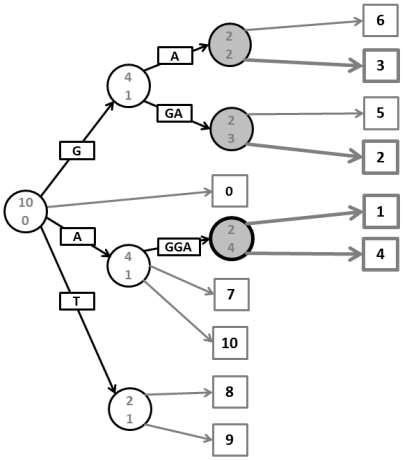
Wir illustrieren den Algorithmus am Beispiel des Wortes CAGGAGGATTA mit $k = 2$ und $l = 1$.



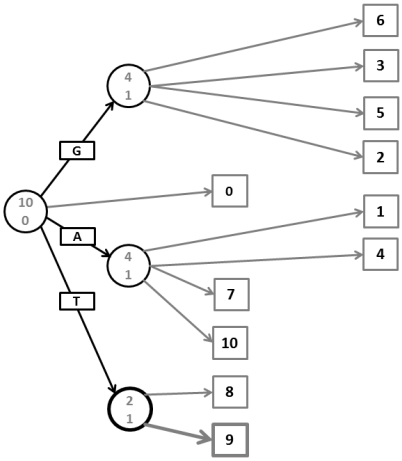
Oben ist der entsprechende Suffixbaum. Die obere graue Zahl entspricht dem Wert von k_v, die untere dem von l_v. Der in Schritt 3.1. gewählte Knoten v ist dick markiert.



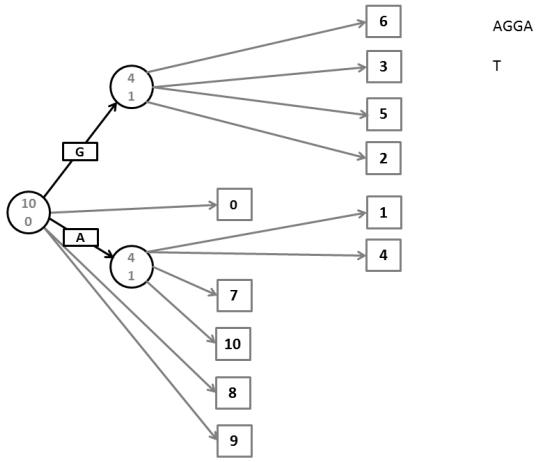
Der Übersicht halber wurden nicht benötigte Beschriftungen entfernt und Blätter grau gemacht. Der Algorithmus gibt AGGA aus und wählt $b = 1$ in 3.3.



Hier sind die Werte hervorgehoben, die b in 3.3. annimmt und die jeweils markierten Knoten sind grau unterlegt.



Die markierten Knoten wurden gelöscht, die Kinder umgehängt. Eine neue Iteration der Schleife (3.) beginnt mit einem neuen Knoten v.



Der Rest der Schleife verläuft relativ unspektakulär, da der betrachtete Knoten $l_v = 1$ hatte. Ähnlich sieht es mit den weiteren Iterationen aus. Insgesamt gibt der Algorithmus AGGA, T, G und A aus.

3.5 Konstruktion eines Suffixbaumes

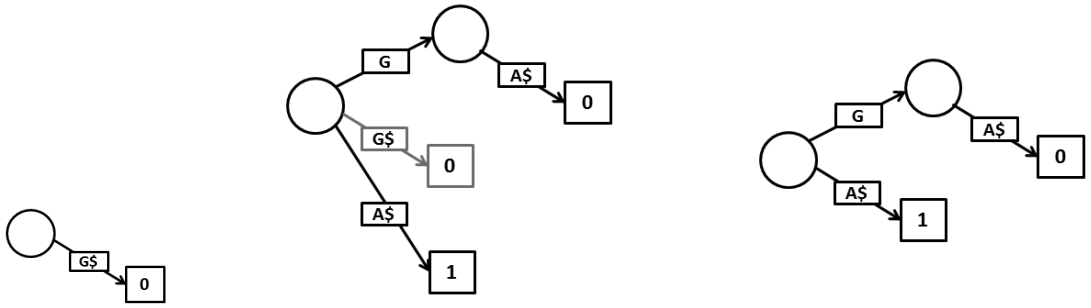
Zu guter Letzt wollen wir uns damit befassen, wie wir Schritt 1 unseres Algorithmus umsetzen. Wir bemerken, dass sich Suffixbäume im Allgemeinen in Linearzeit konstruieren lassen, wodurch unser Algorithmus mit Laufzeit $\mathbf{O}(n+a)$ implementiert werden kann, sprich linear in benötigter Ein- und Ausgabelänge. Diese Implementierung ist jedoch nicht so einfach, weswegen wir auf weiterführende Literatur zu diesen Themen verweisen und hier einen einfacheren Ansatz zu Konstruktion von Suffixbäumen verfolgen, welcher quadratische Laufzeit hat.

Dazu bauen wir den Suffixbaum iterativ für alle Präfixe des Eingabewortes ω auf, wobei wir zuerst die Bedingung 3) des Suffixbaumes ignorieren und stattdessen die Invariante erhalten, dass alle Kanten mit genau einem Zeichen beschriftet sind, ausgenommen die Kanten zu Blättern, deren Beschriftung ein Zeichen gefolgt von \$ sei.

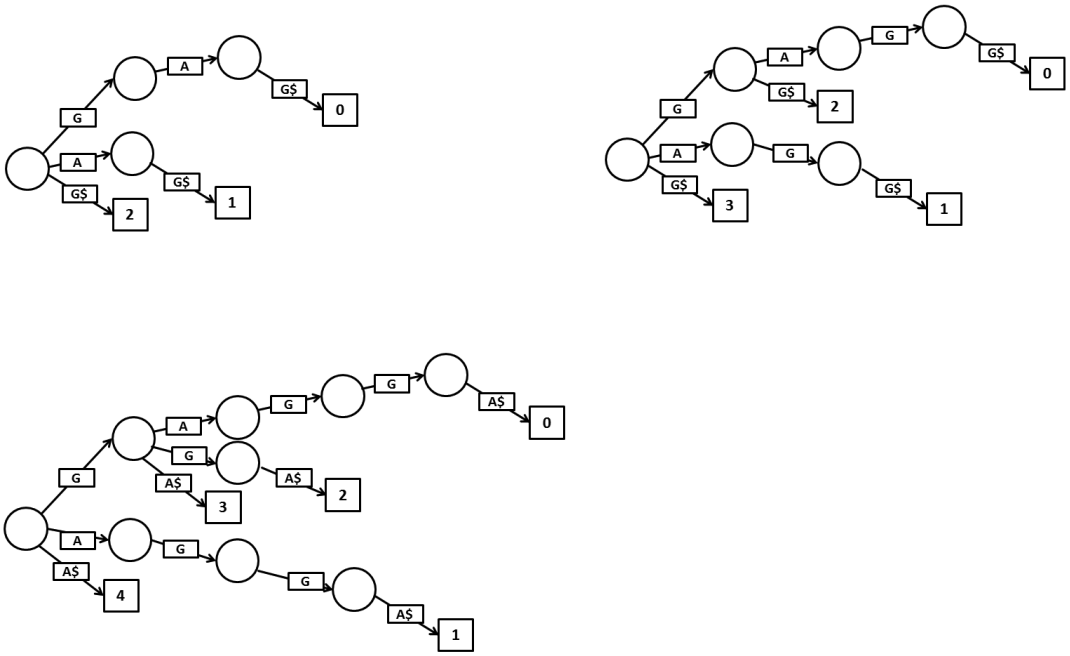
Zu anfangs starten wir einzig mit der Wurzel. Fügen wir in einem Schritt ein weiteres Zeichen c hinzu, definiert uns das einen Suffix der Länge 1, den wir an die Wurzel hängen. Für alle weiteren Blätter b sei v der Vorgängerknoten und a das Zeichen, mit dem die Kante von v nach b (gefolgt von \$) beschriftet ist. Sei w ein Kind von v, das über eine Kante, die mit a beschriftet ist, erreichbar ist. Falls kein solches Kind existiert, fügen wir es hinzu. Wir hängen nun b an w an und ändern die Beschriftung der Kante von w nach b in c\$ um.

Nachdem wir so einen vorläufigen Suffixbaum für ω erstellt haben, müssen wir Bedingung 3 noch herstellen. Das ist jedoch einfach, wir verschmelzen einfach Kanten, welche einen gemeinsamen Knoten von Ausgangsgrad 1 haben.

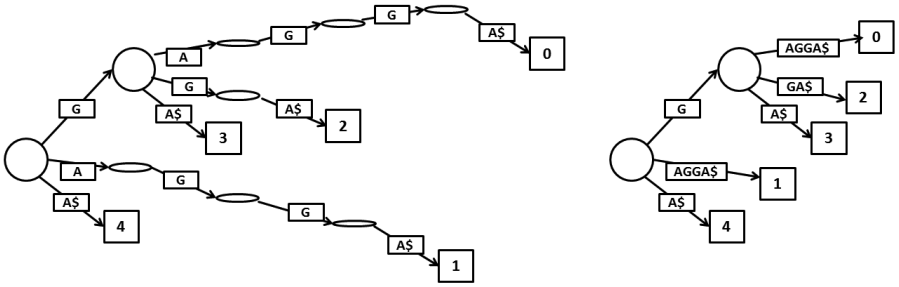
Wir illustrieren dieses Verfahren am Beispiel des Wortes GAGGA:



Das Hinzufügen von G definiert das Blatt 0. Wenn wir A hinzufügen, müssen wir das Blatt 0 verschieben.



Die weiteren Schritte des Algorithmus.



Herstellen von Bedingung 3.

3.6 Erweiterungen

Es ist bereits sehr gut, einen schnellen Algorithmus zur Konstruktion von Suffixbäumen zu implementieren.

Eine Erweiterung wäre, mehrere Sequenzen gleichzeitig nach Repetitionen zu durchsuchen. Dazu sind Zahlen $h, l, k_1, \ldots, k_h$ und Eingabewörter $\omega_1, \ldots, \omega_h$ gegeben. Gesucht sind nun maximale Wörter u, sodass u mindestens Länge l hat und jeweils k_i-fach in ω_i vorkommt. Maximal heißt in dem Fall, das u nicht Teil einer längeren Zeichenkette ist, die in allen ω_i gleichhäufig auftritt.

3.7 Bewertungskriterien

- Ein Bewertungskriterium ist bereits in der Aufgabenstellung gegeben: Es sollen mindestens Sequenzen der Länge 10.000 verarbeitet werden können – und zwar in angemessener Zeit. Einfache Ansätze wie der ganz am Anfang (Abschnitt 3.1) geschilderte kommen bereits bei solchen Wortlängen aus der Puste und sind nicht ausreichend. Für eine gute Lösung musste man sich weitergehende Gedanken machen.
- Allgemein ist die theoretische Laufzeit des implementierten Algorithmus ein sehr guter Bewertungsmaßstab, da schnelle Algorithmen für den Suffixbaum recht kompliziert sind (und andere schnelle Ansätze sind uns nicht bekannt). Insgesamt ist eine quadratische Laufzeit akzeptabel. Die entsteht zum Beispiel bei einer „einfachen“ Konstruktion des Suffixbaumes. Eine Schwäche ist, wenn die Laufzeit gegen k wächst, also bei kleinerem k hoch ist. Das ist etwa bei einfachen Ansätzen der Fall, wenn gefundene Teilstrings abgespeichert und immer wieder durchsucht werden müssen; je kleiner k, desto mehr Teilstrings werden gefunden und desto länger dauern solche Suchen. Wer das Verfahren zur Konstruktion von Suffixbäumen in Linearzeit korrekt selbst implementiert hat, so dass insgesamt lineare Laufzeit entsteht, kann mit Pluspunkten belohnt werden.
- Nicht nur die Laufzeit, sondern auch der Bedarf an Speicherplatz ist kritisch. Ein quadratischer Speicherplatzverbrauch ist maximal und nicht akzeptabel; es geht deutlich besser.
- Selbstverständlich sollte das gewählte Verfahren in sich korrekt sein und die vorgegebenen Werte k (Häufigkeit) und l (Länge) sowie die Forderung nach Maximalität berücksichtigen.
- Die theoretische Laufzeit sollte diskutiert werden. Eine formal wasserdichte Analyse ist nicht gefordert, aber es sollte nicht nur von konkreten Laufzeiten, sondern auch von Größenordnungen die Rede sein.
- Auch zur Korrektheit sollten Überlegungen vorhanden sein.
- Eine fundierte Herleitung oder Begründung des Algorithmus sollte vorhanden sein. Es ist akzeptabel, wenn die Ansätze zur Lösung dieser Aufgabe aus der Fachliteratur übernommen wurden. Es ist für Schülerinnen und Schüler durchaus eine Leistung, die Literatur soweit zu verstehen, dass man die Lösungen übernehmen und implementieren kann. Aber: eine „Literatur-Lösung“ wird gegenüber eigenständig erarbeiteten Ideen

nicht zu hoch bewertet. Deshalb soll auch und gerade in solchen Fällen die Beschreibung der Lösungsideen nachvollziehbar vermitteln, dass der Einsender verstanden hat, warum die in der Literatur angegebenen Verfahren funktionieren.

- Wer die anfangs genannten zentralen Ideen oder zumindest einige davon offensichtlich eigenständig erkannt hat, kann mit Pluspunkten belohnt werden.
- Wer von Suffixbäumen oder ähnlich speziellen, aber bekannten Datenstrukturen redet, wird nicht von alleine auf diese Begriffe gekommen sein; hier werden Verweise auf Literatur oder Internetquellen erwartet.
- Mindestens die kleinere der beiden vorgegebenen Beispieleingaben sollte, mit unterschiedlichen Werten für k und l, bearbeitet und die zugehörigen Ergebnisse dokumentiert sein. Wenn das Programm mit längeren Zeichenketten zurechtkommt, sollte auch die größere Beispieleingabe bearbeitet (und dokumentiert) sein. Falls nicht, sollte die Dokumentation klar dazu Stellung nehmen. Eigene Beispiele können die Pflichtbeispiele ergänzen, insbesondere dann, wenn das große Pflichtbeispiel nicht bearbeitet werden kann.
- Die Ausgabe sollte übersichtlich gestaltet sein. Idealerweise wird die Anzahl der gefundenen Teilstrings mit ausgegeben – am besten zuerst. Außerdem ist eine Sortierung der Strings nach Häufigkeit sinnvoll, auch wenn eine ansonsten lineare Laufzeit dann nicht mehr garantiert werden kann.

Perlen der Informatik – aus den Einsendungen

Teilweise mit Kommentaren von der Bewertung

Allgemeines

Worte des Wettbewerbs: Breath-First-Search *Vor dem Suchen erst mal tief Atem holen!*
standardtifiziert; kompeilt; Terminator-Symbol

Die Unterfunktion [...] wendet Voodoo an, [...] – glauben Sie mir, es funktioniert!

Die Umsetzung ist ziemlich genau wie in der Lösungsidee beschrieben umgesetzt worden.

Um den Lösungsweg zurückzuverfolgen, müssen alle Knoten chronisch in einer Liste gespeichert werden.

... durch eine Methode verkörpert, die beim Aufruf alles tat, was in dieser Phase getan werden musste.

Das hat keinen praktischen Nutzen, beruhigt aber besonders bei längeren Tests den Nutzer mehr, als wenn man keinen Fortschritt erkennen kann.

Die Anzeige bei paralleler Verarbeitung hing an sehr vielen Stellen, und auch das Betriebssystem reagierte nur sehr langsam. Das zeigt jedoch auch, wie gut mein Programm die Systemressourcen nutzen kann.

Aufgabe 1: Seilschaften

Eine Struktur steht für ein Gewicht, welches lebendig sein kann ...

Bei sehr vielen Personen kann man ja ein paar vom Turm werfen, dann schafft das Programm die Lösung für die anderen rechtzeitig.

... unser Korb sollte also in der Lage sein, 64 korpulente Menschen gleichzeitig zu transportieren.

... sperren wir eine 10kg schwere Prinzessin und einen 5kg schweren Stein in einen Turm ein. ... *und eine Packung Windeln!*

`MussGerettetWerden` ist ein Boole'scher Wert, der angibt, ob eine Person gerettet werden muss. So kann man die Typen „Hässliche Prinzessin“ und „Prinzessin Leia“ unterscheiden.

Da Steine nicht sterben können, ist es möglich, Steine im Korb einfach abzulassen, ohne die Fahrtgeschwindigkeit beachten zu müssen.

Ich entschied mich dagegen, dass Personen eine zu rasante Fahrt mit Kraftaufwand bremsen können, denn dies ist meiner Meinung nach nur mit großen Verbrennungen möglich und somit nicht geeignet, angewendet zu werden.

Aufgabe 2: Panorama-Kegeln

Pinanazahl

Der menschliche Spieler sowie Randy erben alle von der gemeinsamen abstrakten Klasse „KI“. *Ob das etwas bringt? Wir Menschen mit unserem irrationalen Getue versauen die ganzen tollen von einer abstrakten KI ererbten Eigenschaften doch ohnehin wieder.*

Aufgabe 3: Mississippi

Suffix-Bau

Um diese Suffixe explizit in den Baum einzutragen, muss es ein Symbol geben, das nicht schon vorher im String auftaucht. Da ich Geld mag, habe ich „$“ gewählt.

Der Stack erfüllt hier noch nicht die Funktion eines Stacks, sondern einer Liste.

… ließ sich das Programm letztlich doch soweit verbessern, dass es eine 10.000 Zeichen lange Kette in einigermaßen angemessener Zeit bearbeiten konnte, ohne sich selbst oder den Computer zu „zerstören“.