23. Bundeswettbewerb Informatik, 2. Runde

Lösungshinweise und Bewertungskriterien

0 Allgemeines

Es ist immer wieder bewundernswert, wieviel an Ideen und Wissen, an Fleiß und Durchhaltevermögen in den Einsendungen zur zweiten Runde eines Bundeswettbewerbs Informatik steckt. Um aber die Allerbesten für die Endrunde zu bestimmen, müssen wir Ihre Arbeit kritisch begutachten und hohe Anforderungen stellen. Von daher sind Punktabzüge die Regel und Bewertungen über das Soll (5 Punkte) hinaus die Ausnahme, insbesondere bei den schwierigen Aufgaben dieses Jahres. Lassen Sie sich davon nicht entmutigen!

Bewertungsbögen Kein Kreuz in einer Zeile bedeutet, dass der entsprechende Aufgabenteil den Erwartungen entsprechend bearbeitet 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 „ok“ bedeutet eine gute, aber im wesentlichen noch erwartungsgemäße Lösung (also meist ohne Zusatzpunkte; für manche Dinge gab es bestenfalls ein „ok“), und ein Kreuz unter „–“ bedeutet Minuspunkte für Fehlendes oder Unzulängliches.

Termin der 2. Runde Die 2. Runde liegt zu großen Teilen parallel zum Abitur. Das ist sicher nicht ideal. In der zweiten Jahreshälfte läuft aber die 2. Runde des Mathewettbewerbs, dem wir keine Konkurrenz machen wollen. Also bleibt uns nur die erste Jahreshälfte. Und damit liegt der Abgabetermin der 2. Runde immer in der Zeit der Abiturtermine. Aber: Sie haben etwa vier Monate Bearbeitungszeit für die 2. Runde. Rechtzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, Konflikte mit dem Abitur zu vermeiden.

Dokumentation Eine Dokumentation beginnt nicht mit: „Die Prozedur abc übergibt einen Zeiger p auf ein Feld xy, worauf die Funktion f ...“. Beschreiben Sie zunächst die (implementationsunabhängige) Idee, die Sie zur Lösung der jeweiligen Aufgabe entwickelt haben. Schildern Sie diese Lösungsidee erst grob und gehen dann darauf ein, wie Sie sie in ein abstraktes, computertaugliches Modell (in Form von Algorithmen und Datenstrukturen) umgesetzt haben. Nutzen Sie dazu auch (halb-)formale Möglichkeiten zur Beschreibung Ihres Modells. Die Implementierung dieses Modells als Programm beschreiben Sie anschließend in der Programmdokumentation, die die wichtigsten Funktionen, Variablen, Klassen, Objekte etc. mit Bezug auf die Lösungsidee dokumentiert und angibt, wo diese Komponenten im Quellcode zu finden sind. Beachten Sie: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt auch keine saubere Umsetzung in welche Programmiersprache auch immer hin.

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.

Aufgabe 1: Klebende Kugeln

Auf den ersten Blick erscheint diese Aufgabe weniger umfangreich oder schwierig als die anderen. Weniger umfangreich ist sie vielleicht wirklich; umso höher sind die Erwartungen, die an eine Lösung gestellt werden.

1.1 Lösungsidee

Obwohl eine Kugel ein dreidimensionales Objekt ist, lässt sich das Problem, wie auch aus der Abbildung in der Aufgabenstellung erkennbar, auf zwei Dimensionen reduzieren. Im Hinblick auf Teil 2 macht das die ganze Aufgabe wunderbar bildschirm- bzw. papierkompatibel. Auch was die anzuwendenden Verfahren betrifft, geht es also um Kreise, die bei ihrer Bewegung entlang einer Geraden an andere Kreise stoßen können und dann ihre Bewegung beenden. Wir wollen im Folgenden aber im Bild der Aufgabenstellung bleiben und von Kugeln sprechen.

Die wesentlichen Punkte, die bei der Bearbeitung der Aufgabe geklärt werden müssen, sind:

Zufallsprozess Was bedeutet: „kommen . . . aus zufälligen Richtungen angerollt“? Was an der Kugelbewegung soll zufällig sein? In Teil 1 ist explizit gefordert, mehrere verschiedene Antworten auf diese Fragen zu diskutieren.

Kreisberührung Wenn die Bewegung einer neuen Kugel festgelegt ist, muss berechnet werden, ob und wo sie ihre Bewegung beendet – also ob und wann zum ersten Mal sie eine schon ruhende Kugel berührt. Die Berechnung kann auf den zweidimensionalen Fall reduziert werden, deshalb also „Kreisberührung“.

grafische Darstellung Teil 2 fordert, dass Bilder von klebenden Kugeln erstellt werden müssen. Es genügt ein statisches Bild, doch natürlich ist es reizvoll, in einer Bildschirmdarstellung auch die Bewegung der Kugeln zu visualisieren.

1.1.1 Zufallsprozess

Zunächst muss die Bewegung der neuen Kugeln definiert werden. Grundsätzlich gibt es zwei Möglichkeiten, hier den Zufall walten zu lassen:

1. Die Kugeln kommen aus einer beliebigen Richtung und laufen auf die Urkugel zu.

2. Die Kugeln bewegen sich von einem zufälligen Punkt auf der Ebene aus in eine beliebige Richtung.

Die Aufgabenstellung legt sich nicht auf eine Variante fest, auch wenn das auf den ersten Blick so aussehen mag. Beide Varianten sind interessant und sollten besprochen werden.

Bei der Wahl des Startpunktes muss der Zufall gesteuert werden. Da die „unendliche Ebene“ nicht realisierbar ist, kann nur ein Ausschnitt betrachtet werden, aus dem die Startpunkte gewählt werden. Dieser Ausschnitt sollte zumindest in der Grundlösung zentriert um die Urkugel liegen. Ansonsten entstehen recht entartete Gebilde, bei denen die neuen Kugeln vermehrt zu einer Seite angelagert werden. Weiterhin hat die Form des Ausschnitts einen Einfluss auf die Form der entstehenden Bilder. Bei der ersten Variante wird eine Gleichverteilung über die Herkunftsrichtungen erzielt, wenn der Ausschnitt kreisförmig ist. Dann ist es sinnvoll, die Startposition der neuen Kugel über einen zufälligen Winkel festzulegen, während z.B. eine Interpretation von Zufallszahlen als Geradensteigungen nicht zu einer gleichmäßigen Verteilung von Startpunkten auf dem Kreis führt. Legt man die Startposition hingegen zufällig (gleichverteilt) in einem rechteckigen Ausschnitt fest bzw. auf dem Rand eines Rechtecks, so werden die entstehenden Gebilde bei hinreichender Größe einen Umriss aufweisen, der eher rechteckig als kreisförmig ist. Bei der Wahl eines Rechtecks oder eines anderen Polygons sollte dieser Effekt erkannt werden.

Auch bei zufälliger Bestimmung der Zielrichtung ist Beliebigkeit zu vermeiden. Die Richtung kann durch zufällige Wahl eines Richtungsvektors oder eines Zielpunktes bestimmt werden. Die Wahl sollte aber so eingegrenzt werden, dass die Bewegung nicht völlig von der Urkugel bzw. dem entstehenden Gebilde wegführt. Eine Möglichkeit ist, den kleinsten Kreis um die Urkugel zu bestimmen, der alle Kugeln des Gebildes enthält und den Zielpunkt daraus zu wählen. Anfangs laufen damit die Kugeln mehr oder weniger auf die Urkugel zu, aber die Bewegungsfreiheit wird immer größer. Es ist sogar möglich, dass eine Kugel das Gebilde verfehlt und nicht kleben bleibt – der Verlust einer virtuellen Kugel lässt sich aber leicht verschmerzen.

1.1.2 Kreisberührung

Die Kugeln sollen anhalten, wenn sie den ersten Kontakt zum bisherigen Gebilde haben. Um dies zu berechnen sind zwei Vorüberlegungen notwendig:

1. Zwei Kugeln berühren sich, wenn ihre Mittelpunkte um die Summe der beiden Radien voneinander entfernt sind. In der Aufgabenstellung haben beide Kugeln gleiche Radien r. Wir können zum Kollisionstest also auch annehmen, dass die alten Kugeln den Radius $2r$ haben und die neue Kugel den Radius 0 hat.
2. Betrachtet man die Bewegung der neuen Kugel, so bewegt sich der Mittelpunkt auf einer Geraden. Um Berührungspunkte zu ermitteln, reicht es also aus, den Schnitt zwischen der Geraden und Kreisen des Radius $2r$ an den Positionen der alten Kugeln zu berechnen.

Der Schnitt zwischen Gerade und Kreis berechnet sich wie folgt:

1. Eine Gerade ist gegeben durch die Gleichung $g : \vec{x} = \vec{p} + s \cdot \vec{u}$. $\vec{p}$ ist der Stützpunkt (der Startpunkt der Kugelbewegung), $\vec{u}$ der Richtungsvektor.
2. Ein Kreis ist gegeben durch $k : (\vec{x} - \vec{m})^2 = r^2$, wobei $\vec{m}$ der Kreismittelpunkt ist und r der Radius des Kreises.
3. Zur Berechnung der Schnittpunkte lösen wir die Kreisgleichung auf und setzen die Geradengleichung ein:

$$g \cap k : (\vec{p} + s \cdot \vec{u})^2 - 2(\vec{p} + s \cdot \vec{u}) \times \vec{m} + \vec{m}^2 = r^2$$

$$g \cap k : s^2 \cdot \vec{u}^2 + s \cdot (2\vec{u} \times (\vec{p} - \vec{m})) + ((\vec{p} - \vec{m})^2 - r^2) = 0$$

$$g \cap k : s^2 + s \cdot \frac{2\vec{u} \times (\vec{p} - \vec{m})}{\vec{u}^2} + \frac{(\vec{p} - \vec{m})^2 - r^2}{\vec{u}^2} = 0$$

   Daraus ergibt sich für s:

$$s_{1,2} = \frac{-\vec{u} \times (\vec{p} - \vec{m})}{\vec{u}^2} \pm \sqrt{\frac{(\vec{u} - (\vec{p} - \vec{m}))^2}{\vec{u}^4} - \frac{(\vec{p} - \vec{m})^2 - r^2}{\vec{u}^2}}$$

$$s_{1,2} = \frac{-\vec{u} \times (\vec{p} - \vec{m}) \pm \sqrt{(\vec{u} - (\vec{p} - \vec{m}))^2 - \vec{u}^2 \cdot ((\vec{p} - \vec{m})^2 - r^2)}}{\vec{u}^2}$$

   Ist die Diskriminante negativ, so schneiden sich Gerade und Kreis nicht. Ist sie positiv, gibt es zwei Schnittpunkte (wovon uns nur der interessiert, der näher am Startpunkt $\vec{p}$ liegt). Ist die Diskriminante Null, berührt die Gerade den Kreis nur in einem Punkt.

Diese Berechnung muss als Kollisionstest für jede alte Kugel durchgeführt werden. Derjenige gefundene Schnittpunkt, der am nächsten an $\vec{p}$ liegt, ist der Punkt, an dem die neue Kugel stehen bleibt. Zu beachten sind Rundungsfehler, da hier mit Gleitkommazahlen gerechnet werden muss. Ohne weitere Verbesserungen werden für n Kugeln $\frac{n(n-1)}{2}$ Kollisionstests durchgeführt. Bei einer großen Kugelzahl (z.B. 100.000) kann die Berechnung eines Bildes dann schon recht lange dauern.

Es gibt auch Alternativen zur Berechnung der Endposition einer neuen Kugel. Mathematisch ist es viel einfacher, für zwei Kreise zu prüfen, ob sie sich berühren: ihre Mittelpunkte müssen dazu genau den Abstand der beiden Radien (bei gleichen Radien r den Abstand $2r$) voneinander haben. Diese einfache Rechnung kann eingesetzt werden, wenn die Kugel in (sehr) kleinen Schritten entlang ihrer Bewegungsgeraden verschoben wird und dann immer wieder auf Berührung mit anderen Kugeln geprüft wird. Durch geschickte Sortierung der Mittelpunkte können unnötige Berührungsprüfungen vermieden werden. Außerdem kann die Kugel am Anfang erst einmal so weit verschoben werden, bis zumindest theoretisch eine Berührung möglich ist; z.B. bis an den Rand des kleinsten Rechteckes, das alle Kugeln des Gebildes enthält. Dennoch ist dieser Ansatz weniger effizient und ungenauer als die oben beschriebene Berechnung des Schnitts von Kreis und Gerade. Noch schlechter ist, wenn eine derartige „Bewegungssimulation“ quasi grafisch durchgeführt wird, also auf der Pixelebene der Bildschirmdarstellung. Den entstehenden Abbildungen kann man das ansehen, denn dann sind Überlagerungen bzw. Abstände zwischen den Kugeln des Gebildes zu erwarten.

1.1.3 Grafische Darstellung

Es ist naheliegend, die Kugeln/Kreise bzw. die entstehenden Gebilde am Bildschirm darzustellen. In der Bildschirmdarstellung können auch das Anwachsen des Kugelgebildes oder sogar die Kugelbewegungen visualisiert werden. Das ist interessant, aber nicht gefordert. Da das entstandene Bild letztlich ein geometrisches Konstrukt darstellt, ist die Ausgabe des Bildes in ein Vektorgrafikformat sehr sinnvoll. Die resultierenden Grafikdateien lassen sich über Betrachtungsprogramme auch am Bildschirm anzeigen, insbesondere aber auch mit hoher Qualität ausdrucken.

1.2 Beispiele

Die auf dieser und den nächsten Seiten folgenden Beispiele sind Ausdrucke von Bildschirmdarstellungen. Die Herkunft der Kugeln wird durch zufällige Wahl eines Winkels bzgl. eines Kreises um die Urkugel bestimmt. Die Gebilde werden dadurch recht kreisförmig, was bei sehr vielen Kugeln besonders gut zu erkennen ist. Weniger kompakt, und stärker verzweigt werden die Gebilde, wenn auch die Bewegungsrichtung zufällig gewählt wird.

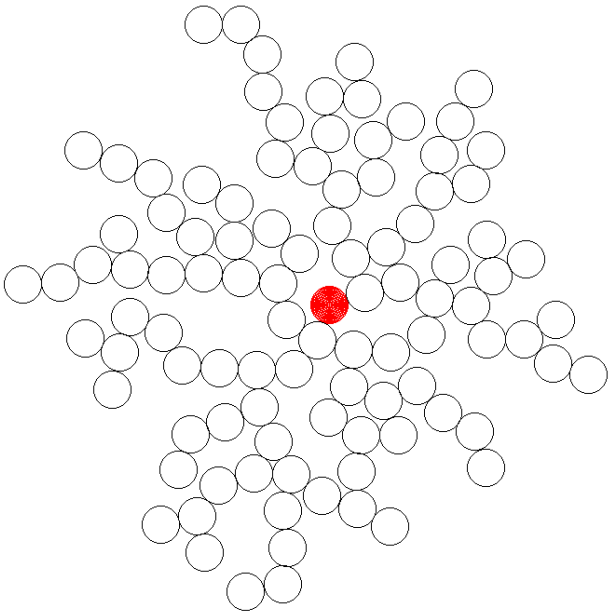


Abbildung 1.1: etwa 100 Kugeln, Bewegung auf die Urkugel zu

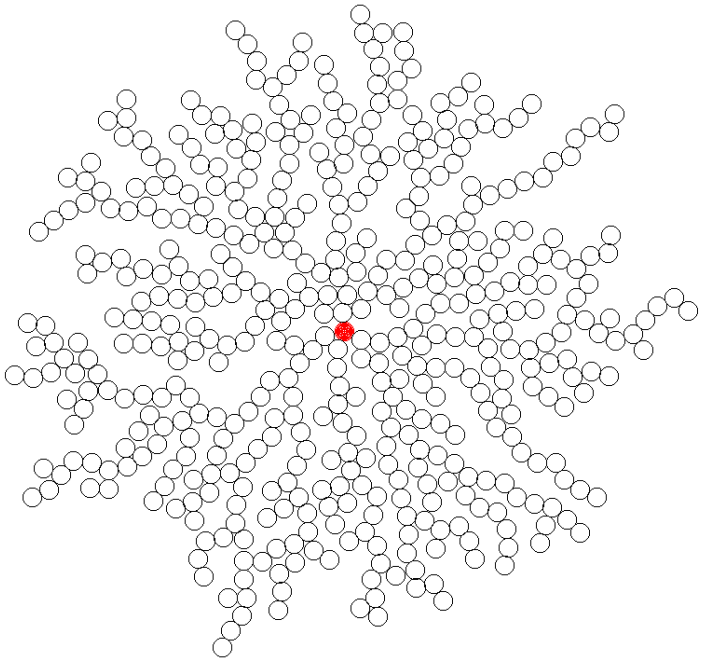


Abbildung 1.2: etwa 500 Kugeln, Bewegung auf die Urkugel zu

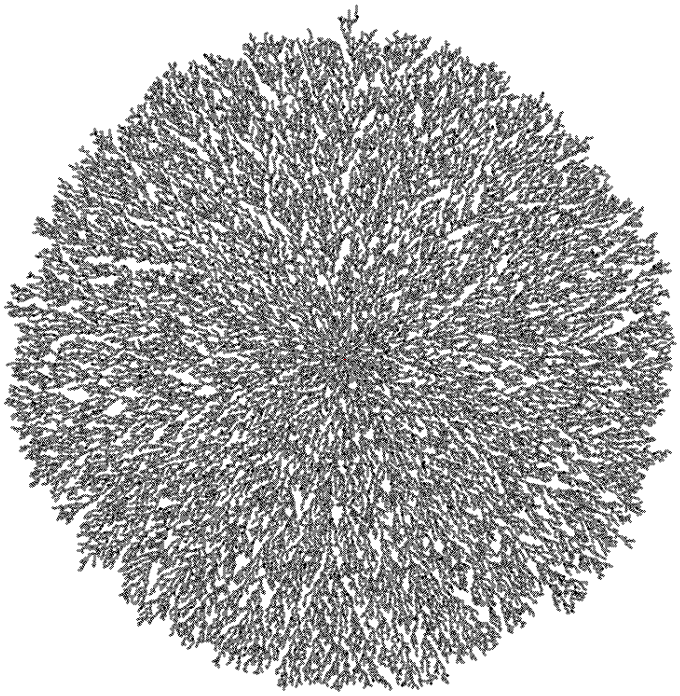


Abbildung 1.3: etwa 50.000 Kugeln, Bewegung auf die Urkugel zu

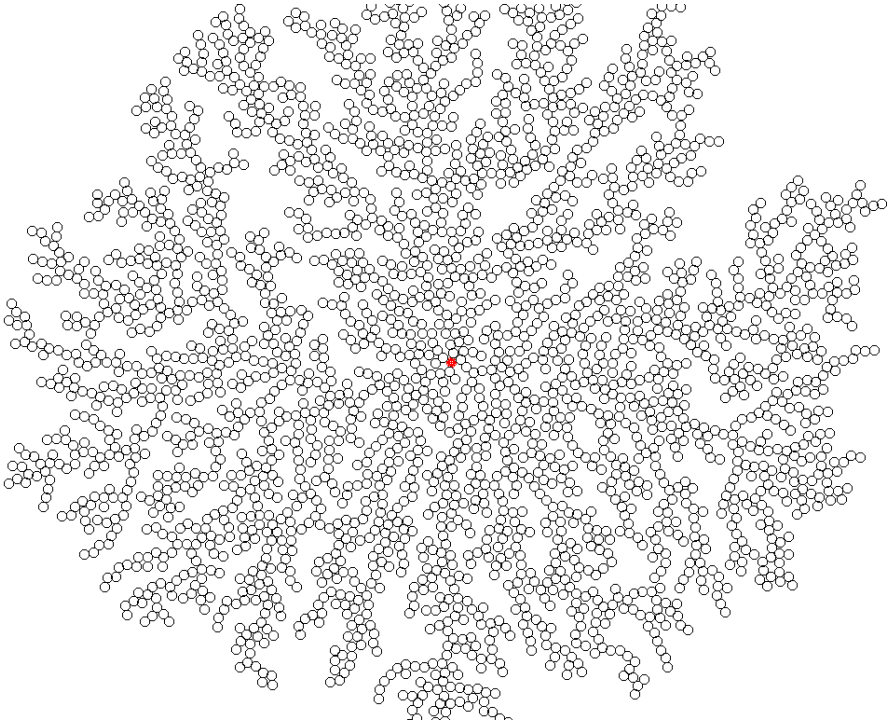


Abbildung 1.4: etwa 10.000 Kugeln, zufällige Richtung

1.3 Erweiterungen

Diese Aufgabe hat die Kreativität vieler Teilnehmer sehr angespornt. Keine Pluspunkte gab es allerdings für besonders schöne Darstellungen; auch dreidimensionale Darstellungen sind nur dann interessant, wenn auch die Kugelbewegung nicht mehr an die zweidimensionale Ebene gebunden ist. Eine nahe liegende Idee ist die Einführung unterschiedlicher, womöglich zufällig gewählter Radien der Kugeln; daraus ergibt sich aber in der Regel keine neue Schwierigkeit, die überwunden werden müsste. Gut gefallen hat uns der Einbau von Hindernissen, mit deren Hilfe das entstehende Gebilde regelrecht geformt werden kann. Hier noch einige weitere Ideen für Erweiterungen:

- Die Form der entstehenden Gebilde wird beeinflusst durch die Form des Bereiches, aus dem die Startpunkte der Kugeln gewählt werden. Eine nette Erweiterung ist es, eine Auswahl verschiedener Formen anzubieten; je freier, desto besser.
- Interessant zu sehen ist auch, was passiert, wenn die Kugeln nicht entlang einer Geraden rollen, sondern entlang von Kurven. Diese Überlegung eröffnet zahlreiche Varianten, im Zweifel wird aber auch die nötige Mathematik deutlich schwieriger. Dies ist zwar ein Informatik-Wettbewerb; dennoch können diejenigen belohnt werden, die diese Schwierigkeiten sauber in den Griff bekommen.

- Anziehungskräfte können realisiert werden, indem Kugeln nicht nur bei Berührung, sondern auch bei Unterschreitung eines bestimmten Abstandes an einer anderen Kugel kleben bleiben. Wenn dieser „Klebeabstand“ mit der Distanz von der Urkugel geringer wird, ist dies wie ein Magnetismus, der von der Urkugel ausgeht und mit abnehmender Kraft durch die anderen Kugeln hindurch wirkt.

1.4 Bewertungskriterien

Beim implementierten Zufallsprozess sollten beide Hauptmöglichkeiten erkannt und besprochen worden sein. Wichtig ist, dass die zufällige Wahl von Startpunkt und ggf. Richtung vernünftig eingeschränkt wird. Auch hierzu sollten verschiedene Varianten besprochen werden. Die Wahl der implementierten Variante muss substanziell begründet sein.

Zur Berechnung der Kreisberührung ist die Berechnung des Schnitts zwischen Bewegungsgerade und Kugeln des Gebildes (mit verdoppeltem Radius) oder eine vergleichbare geometrische Berechnung die beste Möglichkeit. Die Bewegungssimulation ist nur dann noch akzeptabel, wenn dabei geometrisch gearbeitet wird und Maßnahmen zur Minimierung der Anzahl nötiger Berechnungen getroffen wurden; einfaches Vorwärtsschieben der Kugeln mit ständiger Berührungsprüfung mit allen anderen Kugeln oder gar ein Rechnen mit Pixeln ist nicht sinnvoll. Insgesamt sollte die gewählte Methode so effizient sein, dass, wie in der Aufgabenstellung gesagt, „mit möglichst vielen Kugeln“ gearbeitet werden kann. Optimierungen, die die Anzahl der nötigen Berührungsberechnungen deutlich verringern konnten, werden belohnt. Diese Anzahl sollte auch theoretisch besprochen und formell angegeben worden sein.

Bei der grafischen Darstellung sind sowohl Bildschirmausgabe als auch die Ausgabe des endgültigen Bildes in ein vektororientiertes Grafikformat erlaubt. Bitmap-Formate sind nur als Bildschirmfotos sinnvoll. Die Urkugel sollte zumindest bei nicht allzu großen Kugelzahlen von den anderen Kugeln unterschieden werden können. Die Form des Gebildes muss gut erkennbar sein. Entsprechend der Aufgabenstellung sind mindestens drei Bilder zu erstellen, eines davon mit 100 Kugeln. Von den beiden anderen Bildern sollte mindestens eines eine deutlich fünfstellige Zahl von Kugeln zeigen.

Aufgabe 2: Spinnen im Netz

2.1 Lösungsidee

2.1.1 Allgemeines

Ein *Graph* ist ein Gebilde aus Knoten und Kanten. Eine Kante verbindet genau zwei Knoten und kann eine Richtung besitzen. Einen Graphen mit ausschließlich gerichteten Kanten bezeichnet man als *gerichteten Graphen*. Ein *Teilgraph* oder *Subgraph* eines Graphen besteht aus jeweils einer Teilmenge der Knoten und Kanten. Dabei dürfen natürlich nur Kanten übernommen werden, deren beider Endknoten in der Teilmenge der Knoten enthalten sind. Eine Kante ist *adjazent* zu einem Knoten, wenn dieser einer ihrer Endknoten ist. Zwei Knoten sind *benachbart*, wenn sie durch eine Kanten verbunden sind. Eine *Clique* ist ein Teilgraph, bei dem alle möglichen Paare von Knoten benachbart sind.

Die Verweis-Struktur der Website von GOTE SAN[1]stellt genau einen gerichteten Graphen W dar; die Seiten entsprechen den Knoten und die Links den gerichteten Kanten. Wenn man die Unterstruktur-Beschreibung durch einen weiteren Graphen U repräsentiert, lässt sich damit die Aufgabenstellung für „Cyberspinne-S“ direkt auf das Problem der so genannten *Subgraphenisomorphie* abbilden: Ist der Graph U isomorph (dt. etwa: gleichförmig) zu einem Teilgraphen des Graphen W? Isomorphie ist folgendermaßen definiert: Gibt es eine Abbildung, die jedem Knoten aus U einen (verschiedenen) Knoten in W zuordnet, so dass für jede Kante aus U auch zwischen den beiden Abbildern der Endknoten in W eine entsprechende Kante vorhanden ist? Dieses Problem gehört zur Klasse der so genannten *NP-vollständigen Probleme*. Für keines dieser Probleme ist ein effizienter Algorithmus bekannt, und höchstwahrscheinlich existiert auch keiner. Bei allen bekannten Algorithmen steigt die Laufzeit exponentiell in der Größe des Graphen, d. h. für jeden weiteren Knoten vervielfacht sich die Dauer des Programmlaufs um einen bestimmten Faktor. Daher kann schon für relativ kleine Anzahlen von Knoten (unter 50) die Laufzeit so stark ansteigen, dass ein Programmlauf nicht mehr praktikabel ist. Auch die heutigen schnellen Rechner helfen da nicht weiter.

Zurück zur eigentlichen Aufgabe: Es sei hier angenommen, dass das Hinzufügen von beliebigen weiteren Seiten und Links das Vorhandensein der Unterstruktur nicht beeinträchtigt. Auf

[1]GOTE SAN ist ein Anagramm von STEGANO(graphie), denn um diese Kryptographie-Methode geht es in dieser Aufgabe. Auch bei THORA TRYPER lassen sich die Buchstaben umstellen – in HARRY POTTER.

eine bestimmte Ordnung der Links untereinander wird auch kein Wert gelegt, und die Abwesenheit bestimmter Links kann von der Unterstruktur ebenso nicht gefordert werden. Damit ergeben sich genau die Bedingungen, die für Subgraphenisomorphie verlangt sind.

Unabhängig davon muss man sich aber zunächst entscheiden, ob die Richtung der Kanten durch die Unterstruktur-Beschreibung festgelegt werden kann (vgl. den folgenden Abschnitt zu Teil 1). Das Beispiel des Tetraeders suggeriert, dass die Richtung egal ist, denn ein Tetraeder ist vollständig symmetrisch, und seine Kanten sind richtungslos. Auch würde je nach Wahl der Beschreibung der Tetraeder aus dem gegebenen Beispiel nicht gefunden. Für andere zu identifizierende Unterstrukturen könnte eine Richtung der Kanten aber sinnvoll sein, z. B. für baumartige Subgraphen. Eine Lösung sollte beide Möglichkeiten berücksichtigen. Für jeden Link in der Unterstruktur-Beschreibung kann in unserem Lösungsvorschlag angegeben werden, ob die Richtung für diesen festgelegt oder egal ist. Damit wird durch eine solche Beschreibung genau genommen eine Menge dazu passender Subgraphen spezifiziert. Das ändert jedoch nichts Wesentliches an der algorithmischen Schwierigkeit des Problems.

Theoretisch könnte es in den Graphen auch Mehrfachkanten geben, was dann mehrfacher Verlinkung der Seiten entspräche. Dies wird jedoch hier nicht berücksichtigt, mehrere Links von einer Seite zu einer anderen werden wie ein einzelner betrachtet. Weiterhin könnte das Beispiel des Tetraeders den Eindruck erwecken, es solle nach Cliquen einer gewissen Größe im Graphen gesucht werden, denn ein Tetraeder ist einfach nur eine 4-Clique. Das wäre jedoch eine zu stark vereinfachende Annahme, zur Beschreibung der Unterstrukturen würde dann ja eine einzige Zahl ausreichen.

2.1.2 Teil 1: Format zur Beschreibung der Unterstrukturen

Zur Beschreibung des zu identifizierenden Subgraphen wird eine Adjazenzlisten-Darstellung verwendet. Diese wird in folgendem Format aus einer Textdatei eingelesen: In jeder Zeile steht die Adjazenzliste eines Knotens, also eine Liste der von ihm ausgehenden Kanten. Dazu enthält die Zeile den Bezeichner des Knotens und danach pro adjazenter Kante den Bezeichner des jeweiligen Endknotens. Die Gesamtmenge der Knoten ergibt sich implizit. Isolierte Knoten können mittels einer leeren Adjazenzliste kodiert werden. Folgt einem Ziel-Bezeichner direkt ein Sternchen `*`, so bedeutet das, dass die Richtung der Kante beliebig sein darf, ansonsten wird sie als gerichtet betrachtet. Die Bezeichner dürfen daher beliebige Zeichen außer dem Leerzeichen, dem Zeilenendezeichen und dem Sternchen `*` enthalten.

Die Unterstruktur des Tetraeders würde also z. B. folgendermaßen kodiert:

```
W X* Y*
X Z*
Y X* Z*
Z W*
```

Für alle Kanten ist die Richtung egal, deshalb stehen dort überall Sternchen `*`.

2.1.3 Teil 2: Programm „Cyberspinne-S“

Die Website-Struktur wird beginnend mit einer Datei im lokalen Dateisystem eingelesen. HTTP wird nicht unterstützt, jedoch durchaus geschachtelte Verzeichnisse und relative Verweise. Als eindeutige Kennzeichnung dient der absolute Pfad der Datei. Dies ist wichtig, falls Dateien gleichen Namens in verschiedenen Verzeichnissen vorkommen.

Zunächst muss die Website in eine Graph-Datenstruktur eingelesen werden, beginnend mit der Startseite. Für jede Seite wird ein Knoten angelegt und mit dem regulären Ausdruck `<a.*href="(.*?)".*>` nach Links gesucht. Für jeden Link wird für die Zielseite ein Knoten erzeugt, rekursiv die verlinkte Seite bearbeitet (falls Knoten noch nicht vorhanden) und eine entsprechende Kante eingetragen. In der Datenstruktur gibt es dabei drei verschiedene Sorten von Kanten: *Gerichtete Vorwärtskanten* gehen in die gleiche Richtung wie die (gewünschten) Links, *gerichtete Rückwärtskanten* sind immer in gegengesetzter Richtung gespeichert und erlauben ein Rückwärtsgehen im Graphen. *Ungerichtete Kanten* werden immer symmetrisch und nur aus der Unterstruktur-Beschreibung angelegt, sie können in dieser Form auf einer Website nicht vorkommen. Die Kanten werden repräsentiert, in dem jeder Knoten drei Mengen von Zielknoten speichert. Dafür werden Hashes verwendet, um in erwarteter konstanter Zeit die Adjazenz zweier Knoten feststellen zu können.

Zur Erkennung der Unterstrukturen wird ein Backtracking-Algorithmus ausgeführt. Dabei wird nacheinander in einer beliebigen, aber festgelegten Reihenfolge jedem Knoten der Unterstruktur U ein noch freier Knoten aus dem Graphen W der Website zugeordnet und überprüft, ob die Zuordnung insgesamt noch konsistent ist: Existiert für jede Kante in U, deren beide Endknoten schon zugeordnet sind, auch die bzw. eine entsprechende Kante in W? Es reicht dabei jeweils, alle adjazenten Kanten des gerade zugeordneten Knoten K aus U (gerichtet oder ungerichtet) zu überprüfen, denn nur bei ihnen gibt es eine Veränderung. Bei der Überprüfung wird darauf Rücksicht genommen, ob die Richtung der gewünschten Kante festgelegt ist. Ist das nicht der Fall, so genügen auch die entsprechenden Kanten in entgegengesetzter Richtung zur Erfüllung der Bedingung. Sind alle Knoten erfolgreich zugeordnet, so wird die Zuordnung und eine Erfolgsmeldung ausgegeben und das Programm terminiert.

Zur weiteren Beschleunigung führt folgendes Auswahlkriterium: Einem Knoten K aus U wird ein Knoten L aus W nur zugeordnet, wenn der Eingangs- und der Ausgangsgrad (gezählt werden gerichtete Kanten) von L größer gleich sind als die von K. Außerdem muss die Anzahl *aller* Kanten adjazent zu L größer gleich der Zahl der ungerichteten Kanten adjazent zu K sein. Mit dieser Bedingung wird die Anzahl der Zuordnungen also eingeschränkt, ohne korrekte Vorkommen auszuschließen.

Es gibt noch viele weitere Möglichkeiten, den Suchbaum zu beschneiden. Die Menge aus- bzw. eingehender Kanten in die Menge der schon zugeordneten Knoten in U muss z. B. in der entsprechenden Menge an Kanten in W enthalten sein. Die Knoten müssen nicht in festgelegter Reihenfolge zugeordnet werden, sondern z. B. zuerst Knoten, die schon adjazent zu anderen zugeordneten sind. Oder man kann wenigstens die Reihenfolge geschickt wählen, indem man die Knoten zunächst sortiert, z. B. nach Grad absteigend. Die minimalen Abstände

von Knoten im Teilgraphen sind eine weitere Eigenschaft, die man geschickt zum Ausschließen von Teillösungen verwenden kann. Die Berechnung ist zwar u. U. sehr aufwendig, kann sich aber trotzdem rentieren.

Sei n_W die Anzahl der Knoten im Graph der Website, n_U die Anzahl der Knoten in der gesuchten Unterstruktur. Dann hätte dieser Algorithmus im schlechtesten Fall Laufzeit $O\left(\frac{n_W!}{(n_W-n_U)!}\right)$, um alle Subgraphisomorphismen zu finden. Es wird zwar nach dem Finden des ersten Vorkommens abgebrochen und es werden auch hoffentlich nicht wirklich alle Zuordnungsmöglichkeiten ausprobiert, mit Pech wird aber eben überhaupt nichts gefunden. Das bedeutet, dass die Graphen nicht allzu groß sein können, denn sonst benötigt das Programm sehr viel Rechenzeit. In der Praxis sind aber immerhin noch um die hundert Knoten möglich.

Ein einfacher Fall ist, dass die Unterstruktur größer ist als der Website-Graph, also mehr Knoten hat. In diesem Fall wird die Suche erst gar nicht begonnen.

2.1.4 Teil 3: Sicherer Kanal

Dieser Teil der Aufgabe ist sehr offen und hat wenig mit den anderen zu tun. Es geht einfach darum, Daten sicher von GOTE SAN zu seinen Schülern zu übertragen. Das kann eigentlich nicht durch das hier beschriebene Verfahren geschehen, denn sonst hätte man ein Henne-Ei-Problem. Es reicht auch nicht, sich eine Verschlüsselung der Daten (also Unterstruktur und zugehörige Bedeutung) zu überlegen; entscheidend ist ein sicherer Übertragungs*weg*.

Lösungsmöglichkeiten wären z. B. Übertragung mittels Public-Key-Kryptographie oder auch einfach eine persönliche Übergabe der Daten. Ein Brief mit der Beschreibung kann ebenso ausreichende Sicherheit bieten, und wäre nur für die eigentlichen Botschaften zu langsam. Eine Begründung für die vorgeschlagene Lösung ist gefordert.

2.1.5 Teil 4: Programm „Cyberspinne-M“

Um eine Unterstruktur in eine Website zu integrieren, muss der Benutzer die gewünschte Anzahl Knoten $n_W > n_U$ und die Anzahl Kanten m_W mit $m_U \leq m_W \leq {n_W}^2$ dem Programm übergeben. Dann geht das Programm so vor: Der Graph der gewünschten Unterstruktur wird aufgebaut und einfach um $n_W - n_U$ Knoten und um $m_W - m_U$ Kanten zwischen zufälligen Endknoten erweitert.

In jedem Fall kann man aus dem fertigen Graphen leicht eine Website erzeugen, indem man für jeden Knoten eine HTML-Seite mit zufälligen, aber eindeutigen Namen erzeugt (hier `n.html` mit $n \in \{0, \ldots, 10n_W - 1\}$) und für jede Kante einen entsprechenden Link einfügt. Nun gibt es nur noch ein Problem. Es sind eventuell nicht alle (für das Erkennen der Unterstruktur wichtigen) Knoten von einem Startknoten aus erreichbar. Daher wird statt einer normalen Seite eine Index-Seite namens `index.html` eingefügt, die einfach nur Links auf alle Seiten enthält (für diese Aktion müssen natürlich noch ein Knoten und entsprechend viele Kanten übrig sein). Damit sind alle Seiten erreichbar.

2.1.6 Teil 5: Interessante Unterstrukturen

Auch diese Teilaufgabe ist ziemlich offen. Zwei Aspekte können berücksichtigt werden.

Erstens die „interessanten Entsprechungen“: Um die Absprache der Graphen zu vereinfachen, können hier wie schon beim Tetraeder Dinge aus dem wirklichen Leben einen Graphen repräsentieren. Beispiele sind z. B. ein Ausschnitt aus einem Straßen- oder Verkehrsnetz, ein real existierender Baum mit seinen Verzweigungen, chemische Moleküle oder einfach weitere platonische Körper. Auch Symbole wie ein Pentagramm oder die Olympischen Ringe können für einen Graphen stehen. Ebenso sind soziale Netzwerke wie Bekanntschafen (siehe Aufgabe „Tratsch Tratsch“ aus der 1. Runde) oder ein Stammbaum eines Cybermeisters würdig.

Zweitens der eigentliche Zweck der Übung: Es sollen geheime Botschaften ausgetauscht werden. Daher sollte ein Angreifer die geheimen Unterstrukturen nicht erraten können. Dabei kommt es natürlich auch auf das „zufällige“ Drumherum an. Einerseits muss die Unterstruktur so komplex sein, dass man auch große Graphen bauen kann, die zufällig aussehen und trotzdem die Unterstruktur *nicht* enthalten. Ein Beispiel sollte also eine Unterstruktur mit mindestens 10 Knoten enthalten. Andererseits darf die Unterstruktur auch nicht so groß und regelmäßig sein, dass ein zufälliges Auftauchen extrem unwahrscheinlich ist und sie damit „auffällt“(z. B. sehr große Cliquen). Dies könnte der Angreifer sonst auch wieder erkennen.

2.2 Mögliche Erweiterungen

Das *automatische Zeichnen von Graphen* ist eine eigene Disziplin der Informatik. Eine graphische Darstellung der Linkstruktur und der ggf. gefundenen Unterstruktur erleichtert die Erstellung und Überprüfung des Programms. Ist die Erzeugung des Layouts algorithmisch anspruchsvoll, z. B. durch Platzierung der Knoten durch ein kräftebasiertes Verfahren, so kann das als Erweiterung gelten.

Der exponentielle Suchalgorithmus lässt sich u. U. *beschleunigen*, in dem man Selbstähnlichkeit in der Unterstruktur ausnutzt (so genannte Automorphismen von U). Bei jeder Clique (insbesondere beim Tetraeder) ist jede Permutation ein Automorphismus, d. h. man kann die Knoten beliebig austauschen, ohne die Struktur zu zerstören. Dies könnte man dahingehend ausnutzen, die Menge der möglichen Zuordnungen einzuschränken. Beim Tetraeder müsste man z. B. die Zuordnung $A \to W, C \to X$ nicht noch ausprobieren, wenn schon $A \to W, B \to X$ schief gegangen ist, denn B und C sind isomorph. Im allgemeinen ist jedoch schon allein das Finden der Automorphismen schwierig und komplex. Eine wirklich funktionierende Implementierung dieser Idee bedeutet deshalb eine bemerkenswerte Erweiterung.

Das hier vorgeschlagene Verfahren zu Teilaufgabe 4 ergibt keine für das World Wide Web *typische Seitenstruktur*. Hier bietet sich daher viel Raum für Erweiterungen. So könnte man z. B. ausgehend von einer hierarchischen Menüstruktur der Website einen Baum aufbauen,

den man mit einigen Querverweisen anreichert. Die Verwendung von Unterverzeichnissen ist auch schön.

2.3 Bewertungskriterien

Teil 1 Die Richtung der Kanten in der Unterstruktur sollte festgelegt werden *können*. Es muss aber auch Kanten in der Unterstruktur geben können, bei denen die *Richtung* im Link-Graphen *egal* ist, sonst ist das Tetraeder-Beispiel nicht sinnvoll. Die Art der Eingabe des beschriebenen Formats ist egal. Ein Textformat ist zwar die einfachste Lösung, aber auch Eingabemasken oder die grafische Definition von Unterstrukturen sind denkbar.

Teil 2 Es wird nicht verlangt, dass die Programme die Seiten wirklich über HTTP einlesen, lokale Dateien genügen. Allerdings müssen geschachtelte Verzeichnisse und relative Pfade unterstützt sowie Dateien gleichen Namens in verschiedenen Verzeichnissen unterschieden werden. Ohne diese Fähigkeiten würde das Programm auf realistischen Webseiten nicht funktionieren.
Die gefundene Knotenzuordnung muss ausgegeben werden, sonst lässt sich das Ergebnis schwer überprüfen. Das gehört zu einem guten Beispiel dazu.
Insbesondere sollte der Matching-Algorithmus korrekt sein und eine vorhandene Unterstruktur auch entdecken. Wegen der Schwierigkeit des Problems sind Verfahren mit erschöpfender Suche oder Backtracking nicht zu vermeiden, die zu langen Laufzeiten bei größeren Beispielen führen. Diese Problematik sollte erkannt werden. Außerdem sollten wenigstens Versuche zur Verbesserung unternommen werden, besonders gute Verbesserungen werden belohnt.

Teil 3 Der gewählte Übertragungsweg muss einigermaßen sicher sein. Das sollte insbesondere aus der geforderten Begründung hervorgehen, die entsprechend verständlich formuliert sein muss.

Teil 4 Die oben beschriebene Lösung ist nicht wirklich elegant. Man könnte sich auch einen Startknoten bestimmen und dazu einen aufspannenden Baum konstruieren. Auf jeden Fall muss aber der Aspekt der Erreichbarkeit berücksichtigt worden sein. Es nützt nichts, wenn die Unterstruktur vollständig vorhanden ist, aber nicht zu allen Knoten traversiert werden kann. Irgendeinen Knoten der Unterstruktur erreichen zu können, hilft u. U. auch nicht, da die Unterstruktur nicht unbedingt stark zusammenhängend sein muss.
Da die Lösungen zu den Teilen 2 und 4 viele Gemeinsamkeiten haben, sollte die Implementation das auch berücksichtigen und so viele Elemente mit der Implementation zu Teil 2 gemeinsam verwenden wie möglich.

Teil 5 Die Beispiele sollten Strukturen entsprechen, die außerhalb der Welt des Internets vorkommen; Möglichkeiten sind oben reichlich angegeben. Es ist zu berücksichtigen, dass verwendete Unterstrukturen nicht allzu auffällig sein sollten.

Aufgabe 3: Strand

Mit dieser Aufgabe wurde ein ganz spezielles Problem gestellt, für das es keine Standardlösung gibt. Insbesondere war nicht von vornherein klar, ob es optimale Lösungen gibt bzw. wie die Güte einer Lösung gemessen werden kann. Zu dieser Aufgabe werden deshalb einige grundsätzliche Überlegungen angestellt und verschiedene Lösungsansätze beschrieben. Das geht natürlich weit über das hinaus, was wir von einer guten Einsendung erwartet haben, ist aber wahrscheinlich für jeden interessant, der die Aufgabe bearbeitet hat – und auch für diejenigen, die vor den Untiefen der Aufgabenstellung gescheut haben, aber doch ganz gerne wissen wollen, was aus den Rentnern von Norderfrierum geworden ist.

3.1 Diskussion der Anforderungen und wünschenswerten Eigenschaften (Teil 1)

In dieser Teilaufgabe geht es um erste Vorüberlegungen. Wir sollen ein Programm schreiben, dass ankommenden Besuchern am Strand in Norderfrierum hilft, ihre Zelte so aufzustellen, dass folgende Zeltsetzbedingung erfüllt ist:

- Hat der i-te Besucher sein Zelt aufgestellt, so müssen alle i Zelte in einem eigenen i-tel des Strandes liegen. Klappt dies nicht, müssen Zelte versetzt werden.

Die Besucher machen uns das Problem etwas einfacher, indem sie zwar zu unterschiedlichen Zeiten anreisen, aber wenigstens alle gemeinsam abreisen.

Wir können uns nun Gedanken über Anforderungen und gewünschte Eigenschaften machen. Anforderungen an das Programm sind solche Eigenschaften, die das Programm unbedingt erfüllen muss, um überhaupt eingesetzt werden zu können. Dazu zählen die Fähigkeit,

- mit einem leeren Strand beginnen zu können,
- die aktuellen Zelte zu verwalten und auch Zeltsetzungen und -versetzungen entgegennehmen zu können, die die Besucher sich selbst ausgesucht haben,
- immer einen Vorschlag für das nächste Zelt ausgeben zu können, der eventuell auch Zeltversetzungen beinhaltet, und
- nur korrekte Vorschläge auszugeben, bei denen die aktuellen n Zelte in den n verschiedenen n-teln des Strandes liegen.

Wenn man möchte, kann man das Programm auch dazu verpflichten, passende Zeltversetzungen auszugeben für den Fall, dass ein Besucher sein Zelt willkürlich aufgestellt und die Aufteilung durcheinander gebracht hat.

Mit der Festlegung und vor allem der Gewichtung wünschenswerter Eigenschaften legt man grob fest, wie man in Teilaufgabe zwei seinen Algorithmus anlegt. Es gibt hier unterschiedliche Möglichkeiten:

Geschwindigkeit des Programms. Die Wahrscheinlichkeit, dass Besucher den Rat des Programms befolgen, sinken extrem, wenn das Programm zur Berechnung zu viel Zeit benötigt. Auch, wenn man das Programm in der Wartezeit auf den nächsten Besucher vorausberechnen lässt, sollte man darauf achten, dass sich die Berechnungszeit in Grenzen hält.

Hohe „Qualität" der Vorschläge. Hiermit kann man zum Beispiel meinen, dass das Programm möglichst spät die erste Zeltversetzung verursacht, oder insgesamt möglichst wenige (wobei „insgesamt" nicht ganz klar ist), oder auch, dass auch bei unfolgsamen Besuchern die Wahrscheinlichkeit von Kollisionen mit Zelten nicht so hoch ist.

Verarbeitung vieler Zelte möglich. Durch die Einschränkung, dass der Strand von Norderfrierum eine Breite von einem Kilometer hat, kann man der Ansicht sein, dass das Programm nur eine kleine Anzahl von Zelten verwalten muss, vielleicht sogar nur zwanzig, wie in Teilaufgabe drei verlangt. Andererseits möchte man vielleicht sicherstellen, dass das Programm auch in einem sehr heißen Sommer in Norderfrierum einsetzbar ist, wenn sich Besucher vielleicht mit sehr wenig Platz zufrieden geben.

Ausgehend von den gewünschten Eigenschaften kann man sich beim Entwurf des in Teil zwei geforderten Algorithmus in zwei unterschiedliche Grundrichtungen bewegen. Legt man großen Wert darauf, „optimale" Lösungen in Bezug auf erste Zeltversetzung und Anzahl der Zeltversetzungen zu berechnen, so muss man dafür geringere Geschwindigkeit bzw. eine geringere maximale Anzahl von Zelten in Kauf nehmen. Diesen Ansatz werden wir in Abschnitt 3.3 besprechen. Möchte man hingegen ein sehr schnelles Programm, das für sehr viel größere Zeltanzahlen geeignet ist, und nimmt dafür schlechtere Lösungen in Kauf, kann man einen Greedy-Ansatz wie in Abschnitt 3.4 versuchen.

3.2 Darstellung der wichtigsten Größen (Teil 2)

3.2.1 Darstellung von Strandabschnitten

Bei unserem Problem stoßen wir immer wieder auf die unterschiedlichen Einteilungen des Strandes in Strandabschnitte. Wir müssen diese also geeignet speichern können. Für eine vollständige Suche ist es wünschenswert, eine möglichst genaue und flexible Darstellung zu benutzen. Dazu gibt es unterschiedliche Möglichkeiten, die auch in Abbildung 3.1 dargestellt sind:

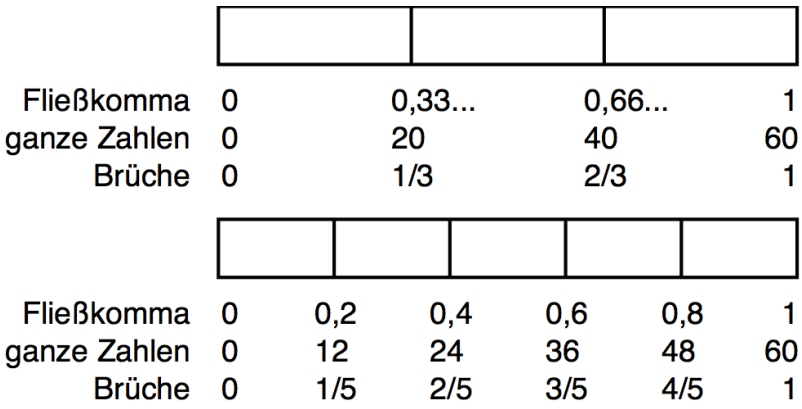


Abbildung 3.1: Dieses Bild zeigt einen Überblick über die verschiedenen Darstellungsformen bei einem Beispiel, bei dem wir davon ausgehen, dass maximal fünf Besucher an den Strand kommen. Daher wählen wir für die Darstellung bei ganzen Zahlen als Auflösung kgV(1,2,3,4,5)=60.

Fließkommazahlen: Am einfachsten ist es, die beiden Randpunkte der Strandintervalle als Fließkommazahlen zu berechnen. Dieses ist zwar sehr flexibel, birgt aber eine gewisse Ungenauigkeitsgefahr. Da wir nach optimalen Lösungen suchen, wäre es doch sehr ärgerlich, wenn wir uns durch einen Rundungsfehler etwas verschenken, auch wenn dieser Fall unwahrscheinlich sein mag.

Brüche: Am elegantesten, wenngleich etwas aufwändiger ist es, alle Randpunkte in Form von Brüchen darzustellen. Aufgrund der Beschaffenheit der Strandabschnitte können Randpunkte nur positive rationale Zahlen sein, die durch Zähler und Nenner bestimmt sind. Bei N Zelten ist der Nenner einfach als N zu wählen, und der Zähler liegt im Bereich $0, \ldots, N$. Die Darstellung ist ebenfalls recht flexibel, und die „Berechnung“ der Randpunkte ist offensichtlich.

Allerdings benötigt man einige Operationen; für unsere Algorithmen zum Beispiel den Vergleich zweier Brüche und Addition / Subtraktion. Möchte man dabei die Größe von Zähler und Nenner möglichst klein halten, sollte man auch Kürzen von Brüchen implementieren. Man kann (selbst ohne diesen Zusatz) mit Brüchen die Strandabschnitte für eine sehr große Anzahl von Zelten darstellen.

Ganze Zahlen: Dieser Punkt ist nur für diejenigen wichtig, die selbst versucht haben, eine Darstellung von Strandabschnitten mit ganzen Zahlen zu entwerfen. Er kann beim Lesen übersprungen werden, es sei denn, es besteht Interesse.

Auf jeden Fall zu ungenau ist es, die Randpunkte der Strandabschnitte in ganzen Metern anzugeben. Wir möchten vielmehr eine genaue Darstellung mit Hilfe von ganzen Zahlen. Die Idee hier ist, den Strand in so viele kleine Stückchen aufzuteilen, dass sich jeder Randpunkt als ganzzahlige Stückchenanzahl darstellen lässt. Dabei ist es wünschenswert, dass man die Stückchenanzahl nicht ständig ändern muss, sondern über die ge-

samte Berechnung die gleiche verwendet; die Darstellung ist sonst zu aufwändig (man möchte z.B. die Position eines Zeltes nicht ständig umrechnen müssen).

Um eine geeignete Stückchenanzahl zu ermitteln, legen wir fest, dass unser Programm maximal N Zelte erwartet. Dann können wir entweder eine Aufteilung in $N!$ Stückchen wählen oder das kleinste gemeinsame Vielfache aller Zahlen von 1 bis N benutzen. Beide Größen haben den Vorteil, dass sie ein Vielfaches aller Zahlen bis N sind. Bei z Zelten hat dann der $a.$ Strandabschnitt die ganzzahlige Größe $\frac{N}{z}$ bzw. $\frac{kgV(1...,N)}{z}$, den Startpunkt $(a-1)\cdot\frac{N}{z}$ bzw. $(a-1)\cdot\frac{kgV(1,...,N)}{z}$ und den Endpunkt $a\cdot\frac{N}{z}$ bzw $a\cdot\frac{kgV(1,...,N)}{z}$.

Dieses Verfahren ist zwar genau, jedoch leider nur für eine sehr kleine (nicht ausreichende!) Zeltanzahl durchführbar. Aus der folgenden Tabelle wird deutlich, dass $N!$, aber auch die Werte des kleinsten gemeinsamen Vielfachen der Zahlen bis N viel zu schnell wachsen. Zwar wird uns auch der später verwendete Algorithmus in Bezug auf die Zeltanzahl einschränken, eine Einschränkung bereits durch die Darstellungsform ist jedoch überhaupt nicht wünschenswert.

Zum Vergleich: $2^{32} = 4.294.967.296$, $2^{64} = 18.446.744.073.709.551.616$

N	N!	kgV(1,2,...,N)
5	120	60
10	3.628.800	2520
12	479.001.600	27.720
13	6.227.020.800	360.360
22	1.124.000.727.777.607.680.000	232.792.560
23	-	5.354.228.880
47	-	442.720.643.463.713.815.200

3.2.2 Darstellung von Zelten

Auch die Darstellung von Zelten ist interessant. Grundsätzlich kann man ein Zelt punktförmig oder als Intervall darstellen. Für eine vollständige Suche ist die Darstellung mit Intervallen, die sehr flexibel ist, auf jeden Fall vorzuziehen. Punktförmige Zelte sind bei unachtsamer Handhabung sogar der Grund für falsche Lösungen (siehe hierzu Abschnitt 3.3.3).

Bei Intervalldarstellung kann man zum Beispiel einem Zelt immer ein Intervall fester Breite zuweisen. Damit lässt sich die reale Ausdehnung eines Zeltes modellieren, andererseits ist die Darstellung nicht sehr flexibel. Zwei weitere Intervalldarstellungen entwickeln sich in der Diskussion der folgenden algorithmischen Überlegungen, zum einen die Stückchendarstellung in Abschnit 3.3.2 und zum anderen die variable Intervallarstellung, die in 3.3.3 beschrieben wird und den Kern dort geschilderten Algorithmen bildet.

3.3 Vollständige Suche (Teil 2)

In diesem Abschnitt versuchen wir, optimale Lösungen für das Problem zu finden. Wir werden dadurch auch Erkenntnisse über die grundsätzliche Lösbarkeit erlangen. Die Analyse erstreckt sich über mehrere Abschnitte, enthält dafür jedoch auch interessante Erkenntnisse.

3.3.1 Top-Down oder Bottom-Up

Vor der Besprechung der Algorithmen möchten wir kurz die beiden Begriffe Top-Down und Bottop-Up erwähnen. „Top-Down“ bezeichnet in der Regel einen Ansatz, bei dem ein Problem von der Problemstellung aus durch fortschreitendes Verfeinern und bearbeitet wird, „Bottom-Up“ meint eine Bearbeitung von potenziellen Lösungen oder Lösungsdetails her. Bei uns betrifft diese Unterscheidung die Reihenfolge, in der die Zelte platziert werden. „Top-Down“ setzt Zelte nacheinander auf den freien Strand und betrachtet dann die neue Situation für weitere Zelte. „Bottom-Up“ beginnt mit einer bestimmten Anzahl von Zelten, platziert diese in ihre Strandabschnitte, und entscheidet dann rückwärts, welches Zelt für den vorherigen Schritt gelöscht wird.

Zunächst erscheinen beide Ansätze gleichwertig. In Abschnitt 3.3.5 stellen wir jedoch fest, dass der Bottom-Up-Ansatz für die Aufgabe nur schlecht anwendbar ist.

3.3.2 Algorithmus mit Strandstückchen

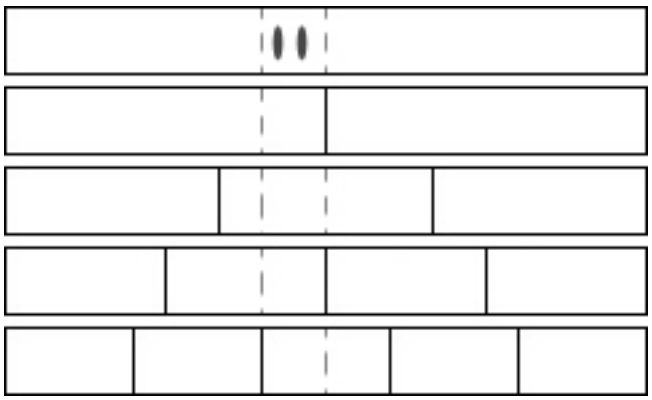


Abbildung 3.2: Beide (sogar alle) Positionen zwischen den gestrichelten Linien sind gleichwertig.

Versucht man eine „vollständige Suche“ für unser Problem zu entwerfen, wird schnell deutlich, dass es kaum möglich ist, „alle“ Positionen eines Zeltes auf dem Strand auszuprobieren. Selbst wenn man einem Zelt eine feste Breite gibt, hat man immer noch unendlich viele Möglichkeiten, an welcher Stelle man es aufbaut.

Wir stellen daher zunächst einige Vorüberlegungen an. Dazu betrachten wir die beiden grauen Markierungen in Abbildung 3.2, die zwei mögliche Positionen für ein Zelt darstellen sollen. In der Abbildung ist ebenfalls dargestellt, wie der Strand für bis zu fünf Besucher schrittweise aufgeteilt wird. Es wird klar, dass es egal ist, an welcher der beiden Positionen wir ein Zelt platzieren, da beide Positionen in jeder Aufteilung im gleichen Strandabschnitt liegen. Es sind sogar alle Positionen zwischen den gestrichelten Linien in diesem Sinne gleichwertig.

Abbildung 3.3: Die Aufteilungen des Strandes in gleiche Teile werden „übereinandergelegt“, um die Stückchen zu berechnen, die jeweils in allen Stufen nur in einem Strandabschnitt liegen.

Wir möchten daher nun solche Abschnitte berechnen, in denen zwei Zeltpositionen nie einen Unterschied machen. Bei einer Anzahl von N erwarteten Zelten „schieben“ wir dazu wie in Abbildung 3.3 alle Aufteilungen für 1 bis N Zelte übereinander. Wir erhalten so eine Aufteilung des Strandes in Stückchen, und nur Positionen in unterschiedlichen Stückchen machen beim Platzieren wirklich einen Unterschied.

Für die Berechnung der Stückchen geht man am einfachsten alle entstehenden Brüche durch und sortiert diese. Man benötigt entweder eine Kürzen-Funktion für Brüche oder eine Gleich-Funktion, die durch Erweitern Brüche auf Gleichheit testet. Dann erhält man für N Zelte die in der folgenden Tabelle angegebene Anzahl von Stückchen.

N	Stückchen	N	Stückchen
1	1	25	200
2	2	50	774
3	4	100	3044
4	6	150	6858
5	10	250	19024
10	32	500	76116
15	72	750	171018
17	96	1000	304192

„Vollständige Suche“ könnte nun bedeuten, alle Möglichkeiten durchzugehen, N Zelte auf die entstandenden Stückchen zu verteilen. Es müsste dann noch getestet werden, auf welcher Aufteilungsstufe man welche der N gesetzten Zelte hat bzw. wann welches Zelt hinzukommt. Leider ist die Anzahl der Zelt-Verteilungs-Möglichkeiten bereits für 15 Zelte größer als eine Billiarde (das Platzieren von N Zelten auf k freie Stückchen entspricht dem Ziehen von N Kugeln aus einer Urne mit k Kugeln ohne Beachtung der Reihenfolge und lässt sich mit Hilfe des Binomialkoeffizienten berechnen), so dass eine vollständige Suche nicht durchführbar ist:

N	Stückchen	Anzahl Möglichkeiten
1	1	1
2	2	1
3	4	4
4	6	15
5	10	252
10	32	64.512.240 $> 64 \cdot 10^6$
15	72	1.155.454.041.309.504 $> 10^{15}$
25	200	$> 10^{240}$

Alternativ kann man sich z.B. auch einen rekursiven Top-Down-Algorithmus vorstellen, bei dem für ein Zelt rekursiv eines der freien Stückchen ausgewählt wird. „Frei“ bedeutet hierbei, dass das Stückchen auf der aktuellen Stufe nicht im gleichen Strandabschnitt wie ein bereits besetztes liegt. Im Prinzip schließt man hierdurch ein paar nicht funktionierende Kombinationen aus, leider handelt man sich aber durch die Rekursion ein, dass die „Reihenfolge“ wieder eine Rolle spielt. Die gleiche Kombination von Zelt-Stückchen-Paaren lässt sich erzeugen, egal, mit welchem Zelt man angefangen hat, sie taucht also in der Berechnung immer wieder auf und ist sehr schwer vermeidbar. Durch dieses Problem geht ein solcher Algorithmus noch mehr als die in der Tabelle oben aufgeführten Fälle durch.

Führt man sich jetzt vor Augen, dass wir uns noch gar nicht damit beschäftigt haben, was passiert, sobald Zelte versetzt werden, kann man sich vorstellen, dass man es sich bei der Lösung dieser Aufgabe beliebig schwer machen kann.

3.3.3 Algorithmus mit variablen Zeltintervallen

Fast die spannendste Frage, die sich bei der Betrachtung der Aufgabe stellt, ist, ob man eigentlich durch eine optimale Berechnung das Versetzen von Zelten vollständig vermeiden kann. Glücklicherweise gibt es für die Beantwortung dieser Frage einen algorithmischen Ansatz, der mehr Erfolg verspricht als der in Abschnitt 3.3.2. Wir betrachten nun einen Algorithmus, der eine vollständige Suche mit einem rekursiven Bottom-Up-Ansatz durchführt.

Bottom-Up

Bei diesem Ansatz wird die Darstellung der Zelte eine ganz entscheidende Rolle spielen. Wir wählen eine Intervalldarstellung (und speichern alle Zelte und Strandabschnitte mit Brüchen).

Wegen des Bottom-Up-Ansatzes beginnen wir auf der untersten Stufe, auf der wir N Zelte haben. Der Algorithmus beginnt jetzt, indem er jedem Zelt ein Intervall zuweist, in dem es stehen kann, nämlich einen der N Strandabschnitte. Zelt eins steht also in $(0, \frac{1}{N})$, Zelt zwei im Intervall $(\frac{1}{N}, \frac{2}{N})$, Zelt N in $(\frac{N-1}{N}, 1)$. Das Intervall eines Zeltes stellt den Spielraum dar, der für das Setzen des Zeltes zur Verfügung steht. Können überhaupt N Besucher ihre Zelte auf dem Strand aufbauen, ohne Zelte zu versetzen, müssen die Zelte am Ende auf jeden Fall innerhalb der Intervalle stehen, die wir hier auch zugewiesen haben.

Wir wollen nun berechnen, wie der Strand einen Schritt früher aussah, d.h. wir müssen auf $N-1$ Zelte reduzieren, indem wir ein Zelt löschen. Der Spielraum, den wir haben, ist die Entscheidung, welches Zelt gelöscht wird. Unser Algorithmus probiert alle Zelte rekursiv durch und realisiert so die vollständige Suche. Ist eines der Zelte gelöscht, müssen wir berechnen, welche Konsequenzen dieses für unsere berechnete Lösung hat. Die $N-1$ verbleibenden Zelte müssen ja jetzt in die $N-1$ neuen Strandabschnitte passen.

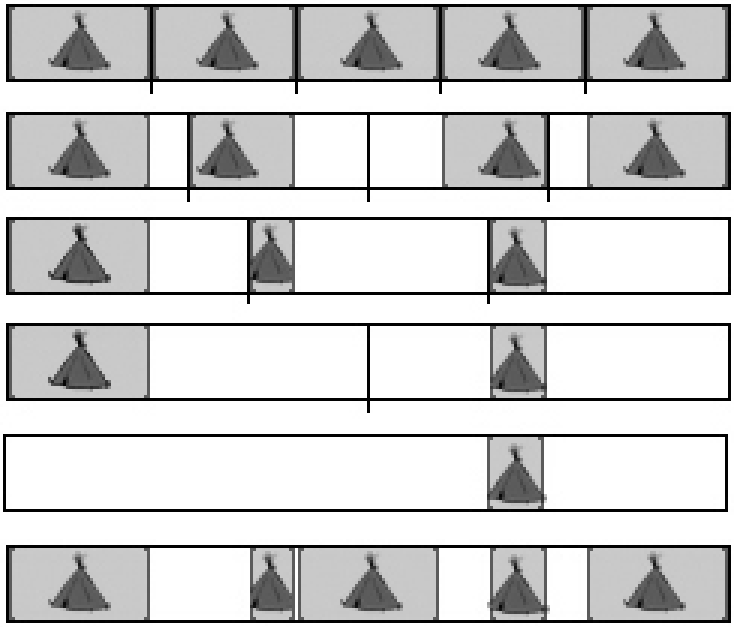


Abbildung 3.4: Illustration des Bottom-Up-Algorithmus' mit Zeltintervallen

Wir betrachten hierzu zunächst das Beispiel in Abbildung 3.4. Es soll eine Lösung für fünf Zelte berechnet werden. Zu Beginn entsprechen die Intervalle der Zelte den Zeltabschnitten. In der Abbildung wird dann das dritte Zelt gelöscht (beliebig gewählt). Dadurch muss jetzt das ursprünglich vierte Zelt in der nächsten Stufe im dritten Strandabschnitt liegen, damit keine Kollision auftritt. Das Intervall dieses Zeltes verringert sich also von $(\frac{3}{5}, \frac{4}{5})$ auf $(\frac{3}{5}, \frac{3}{4})$.

Ebenso schrumpft das Intervall des zweiten Zeltes von $(\frac{1}{5}, \frac{2}{5})$ auf $(\frac{1}{4}, \frac{2}{5})$, da das zweite Zelt auf Stufe zwei nicht mit dem ersten kollidieren darf. In der Abbildung werden dann Zelt fünf, Zelt zwei und am Schluss Zelt eins gelöscht. Die Intervalle der Zelte zwei und vier schrumpfen dabei weiter. In der letzten Zeile der Abbildung sind noch einmal die Intervalle aller Zelte eingetragen. Die vom Algorithmus berechnete Lösung wäre nun, die Zelte innerhalb dieser Intervalle zu setzen, in umgekehrter Reihenfolge wie bei der Berechnung.

Allgemein beschrieben passiert nach dem Löschen eines Zeltes also folgendes: die verbleibenden Zelte müssen der Reihe nach in die Strandabschnitte passen. Ihre neuen Intervalle sind der Schnitt ihres alten Intervalls und des Strandabschnittes, in den sie jetzt gehören. Den Schnitt bildet man einfach, indem man das Maximum der beiden Intervallanfänge und das Minimum der beiden Intervallenden ermittelt. Natürlich führen nicht alle (sogar eher wenige) der Rekursionszweige tatsächlich zu einer Lösung. Man erkennt eine Sackgasse daran, dass eins der Intervalle eine negative bzw. keine Länge hat. In diesem Fall sind durch das Löschen und die neuen Strandabschnitte zwei Zelte in den gleichen Strandabschnitt gelangt. Wir brechen also ab und kehren auf die nächsthöhere Rekursionsebene zurück.

Wir haben jetzt einen funktionierenden Algorithmus beschrieben, der eine vollständige Suche für eine gegebene Zeltanzahl durchführt und, falls möglich, eine Lösung ohne Zeltversetzungen zurückliefert. Das Entscheidende bei diesem Algorithmus ist die Intervalldarstellung; versucht man, den Algorithmus mit festen Zeltpositionen zu konstruieren, hat man trotz Rekursion nur einen Greedy-Algorithmus programmiert, bei dem die Wahl der Zeltpositionen entscheidend ist.

Mit Hilfe des Algorithmus haben wir Lösungen für 1 bis 17 Zelte berechnet. Für 18 Zelte findet unser Programm keine Lösung. Es ist also gar nicht möglich, mehr als 17 Zelte zu setzen, ohne ein Zelt zu bewegen!

Top-Down

Der gerade in 3.3.3 beschriebene Ansatz lässt sich auch „vorwärts“, also Top-Down, realisieren. Auch hier benutzen wir kleiner werdende Zeltintervalle.

Wir beginnen mit einem einzigen Zelt, dem der gesamte Strand zur Verfügung steht. Im zweiten Schritt der Berechnung teilen wir den Strand in zwei Abschnitte auf und entscheiden uns (rekursiv), in welchen Strandabschnitt wir unser zweites Zelt setzen möchten. Das Zelt belegt diesen Abschnitt zunächst ganz. Für die alten Zelte (in diesem Schritt nur eins) berechnen wir den Schnitt ihres alten Intervalls und des Intervalls, in dem sie nun landen (dieses Intervall ist ja durch Positionierung des neuen Zelts festgelegt).

Das Schöne an diesem Ansatz ist nun, dass man ihn im Prinzip ohne vorherige Beschränkung der Zeltanzahl starten und dann abwarten kann, bis zu welcher Tiefe er Lösungen findet. Wie schon nach Abschnitt 3.3.3 klar war, existieren Lösungen bis zur Tiefe 17. In Abbildung 3.5

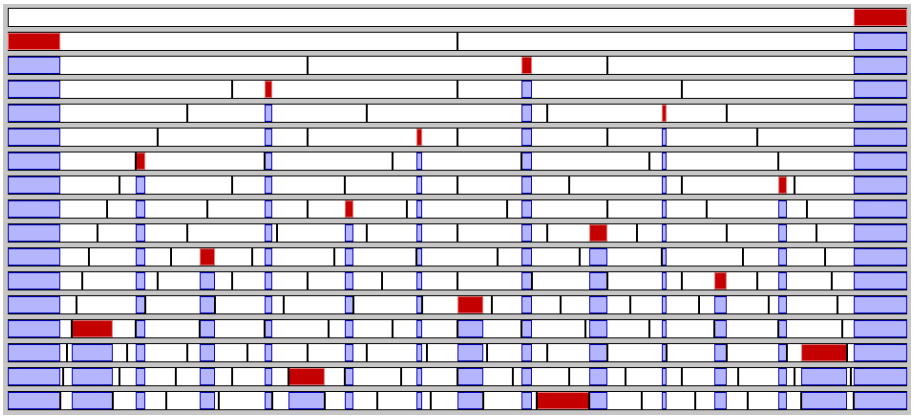


Abbildung 3.5: Die Abbildung zeigt die Intervalle, in denen die Zelte platziert werden dürfen, und die Stufen von einem bis zu 17 Zelten. Die neu hinzugefügten Zelte sind farblich hervorgehoben.

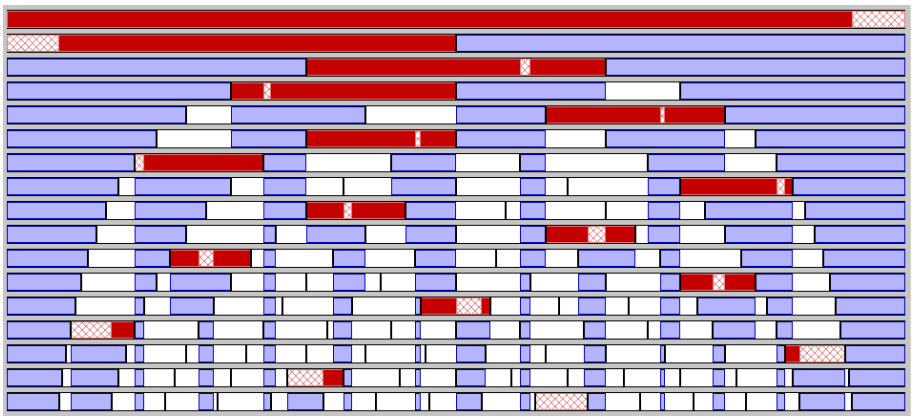


Abbildung 3.6: Diese Abbildung verdeutlicht, wie Abbildung 3.5 entstanden ist. Ein neues Zelt bekommt immer das ganze Intervall, wird aber schrittweise eingeschränkt. Man sieht zum Beispiel sehr schön, wie die Intervalle der äußeren Zelte sehr symmetrisch schrumpfen. Die schraffierte Fläche zeigt das Intervall an, das dem gerade eingefügten Zelt am Ende zur Verfügung stehen wird.

sieht man eine der von unserem Programm berechneten Lösungen für 17 Zelte grafisch dargestellt, in Abbildung 3.6 ist dargestellt, wie dieser Pfad des Algorithmus' die Lösung Schritt für Schritt berechnet hat.

Probleme bei punktförmigen Zelten

Wie in 3.2.2 angekündigt, haben wir auch ausprobiert, was passiert, wenn man punktförmige Zelte zulässt (bzw. mit punktgenauer Positionierung der Zelte arbeitet). Diese sind bei uns darstellbar durch Intervalle der Länge null. Da der Algorithmus versucht, eine Lösung mit

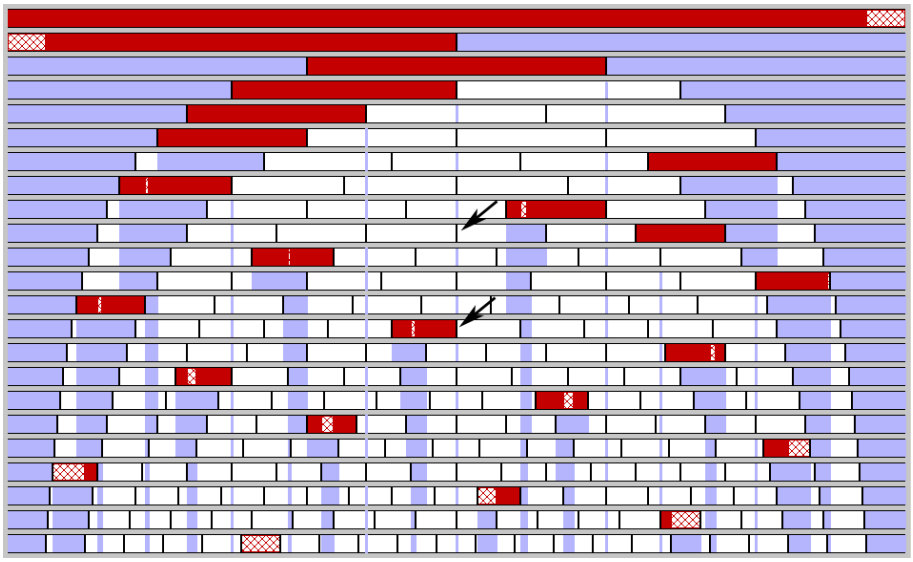


Abbildung 3.7: Die Abbildung ist genauso aufgebaut wie 3.6. Die sechs punktförmigen Zelte sind etwas verbreitert worden, damit man sie besser sehen kann.

möglichst vielen Zelten zu finden, nutzt er den Spielraum aus und produziert das in Abbildung 3.7 dargestellte Ergebnis. Man sieht sechs sehr schmale Zelte. Durch die Verwendung dieser punktförmigen Zelte steigert der Algorithmus die maximale Tiefe auf 23. Wir betrachten eines der punktförmigen Zelte, an der Stelle, auf die der obere Pfeil zeigt. Das Zelt liegt hier genau auf der Strandabschnittgrenze. Im Intervall rechts daneben ist bereits ein Zelt, unser Zelt gehört also in das links liegende Intervall. An der Stelle, auf die der zweite Pfeil zeigt, wird links von unserem Zelt ein neues eingefügt, und das rechts liegende Intervall ist frei. Hier gehört unser Zelt also in das rechte Intervall. Hätte das Zelt eine minimale Ausdehnung, würde also an einer der beiden Stellen eine Kollision entstehen. Daher ist diese Lösung falsch, und die hier beschriebene Verwendung von punktförmigen Zelten bei rekursiven Algorithmen führt zu falschen Lösungen.

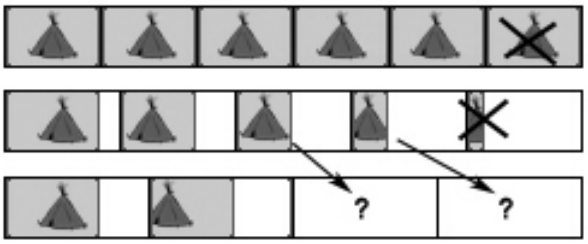
Man kann das Problem mit punktförmigen Zelten lösen, indem man einem Zelt eine „Orientierung“ zuweist, die angibt, ob es zum linken oder rechten Intervall gehört, wenn es auf einer Intervallgrenze landet.

3.3.4 Berücksichtigung von Zeltversetzungen

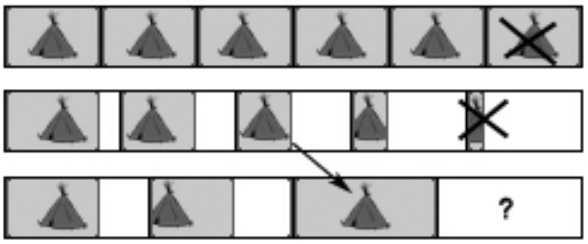
Als nächstes möchten wir wissen, wie sehr uns Zeltversetzungen helfen, unsere Zelt-Kapazität auf mehr als 17 zu erhöhen. Eine Frage dabei ist, wann man eigentlich Zelte versetzen soll. Intuitiv beginnt man damit, sobald eine Kollision aufgetreten ist. Theoretisch ist es aber auch möglich, durch eine viel frühere Zeltversetzung zu erreichen, dass Zelte so gesetzt werden können, dass mehr als eine Kollision vermieden werden kann. Bezieht man diese Möglichkeit mit ein, erhält man jedoch ein extrem komplexes Problem, bei dem ständig Zelte versetzt

werden dürfen. Wir halten es daher für legitim, nur Zelte zu versetzen, wenn Kollisionen auftreten. Dieses kann man den Campern auch viel besser erklären!

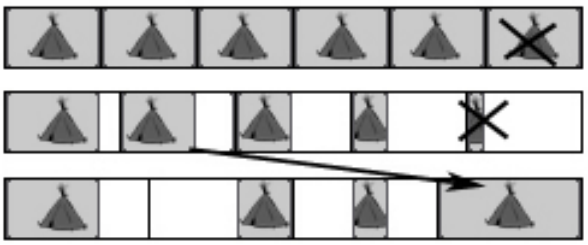
Probleme mit Zeltversetzungen



(a) Der Algorithmus hat hier zunächst Zelt sechs gelöscht, und will dann Zelt fünf löschen (das ist nicht besonders intelligent, liefert uns aber ein schönes Beispiel). Dadurch sind in der nächsten Stufe die beiden rechten Intervalle unbesetzt.



(b) Hier sieht man eine mögliche Versetzung des dritten Zeltes. Es wird genau eines der Intervalle gefüllt.



(c) Bei dieser Versetzung fällt das ursprünglich vierte Zelt automatisch in das dritte Intervall, wir sparen eine Versetzung.

Abbildung 3.8: Ein Beispiel für Zeltversetzungen beim Bottom-Up-Ansatz

Selbst nach dieser Einschränkung bleibt die Anzahl der Möglichkeiten, Zelte zu versetzen, enorm. Wir betrachten dazu zunächst Abbildung 3.8.

Durch (ungeschicktes) Löschen von Zelten tritt hier eine Situation auf, in der zwei Intervalle leer sind. Das Problem ließe sich einfach lösen, indem man die beiden Zelte, die im Moment keinen Platz haben, in die beiden Intervalle verschiebt. Dabei würde es keine Rolle spielen, welches Zelt man in welches Intervall versetzt, da ein versetztes Zelt ja das neue Intervall komplett ausfüllt und somit immer die gleiche Situation entstehen würde. Das unterste Bild zeigt jedoch, dass es auch ganz anders geht. Hier reicht eine Versetzung aus. Es kommt jedoch auch darauf an, dass wir unser Zelt in das rechteste Intervall versetzt haben, so dass die anderen Zelte in die richtigen Strandabschnitte einsortiert werden. Man sieht auch, dass wir ein eigentlich unbeteiligtes Zelt versetzt haben! Wir hätten zwar in diesem Beispiel auch das dritte statt dem zweiten Zelt versetzen können, aber zumindest wird plausibel, dass das Versetzen von unbeteiligten Zelten einen Unterschied macht.

Um wirklich alle Möglichkeiten zu überprüfen, muss man daher alle Kombinationen von Zelten und möglichen Zielintervallen, also bis zu N^2 Möglichkeiten, durchprobieren. Geht man all diese Fälle rekursiv durch, entsteht ein extrem großer Berechnungsbaum.

Das Beispiel 3.8 bezog sich auf den Bottom-Up-Ansatz, die Argumentation ist jedoch für beide Ansätze sehr ähnlich.

Erweiterung um Zeltversetzungen

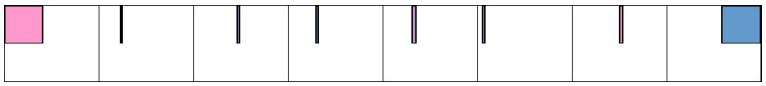
Trotz der Komplexität kann man versuchen, die Algorithmen zur vollständigen Suche um Zeltversetzungen zu erweitern.

Beim Top-Down-Ansatz hat man es beim Einbauen relativ leicht. Wir haben es so gemacht, dass wir nach einem gescheiterten Test das Setzen des letzten Zeltes rückgängig gemacht, uns die Einfügenummer aber gemerkt haben. Dann werden alle Kombinationen von Zelten und möglichen Zielen durchprobiert, wobei die Zielintervalle die Strandabschnitte sind, die entstehen, wenn wir das nächste Zelt erneut platzieren. Nur das Intervall, in das wir sowieso das nächste Zelt setzen wollen, wird übersprungen. Haben wir ein Paar aus Zelt und Ziel, müssen wir nicht nur die Intervallgrenzen des Zeltes ändern, sondern auch das Zelt zwischen den anderen Zelten neu einsortieren. Haben wir uns dafür entschieden, es auf der nächsten Stufe in Intervall i zu setzen, muss es auch an dieser Position zwischen den anderen Zelten stehen, um tatsächlich in dieses Intervall zu fallen. Abschließend setzen wir das neue Zelt erneut und überprüfen wie im normalen Algorithmus, ob jetzt jedes Zelt mit seinem Intervall vereinbar ist. Ist dieses der Fall, kann die Rekursion fortgeführt werden. Natürlich werden alle möglichen Zelt-Ziel-Paare rekursiv durchprobiert. Es kann auch notwendig werden, mehrere Zelte in einem Schritt zu versetzen. Für diesen Fall merkt man sich die berechneten Zeltintervalle vor dem Neuplatzieren des neuen Zeltes und führt rekursiv eine weitere Zeltversetzung durch.

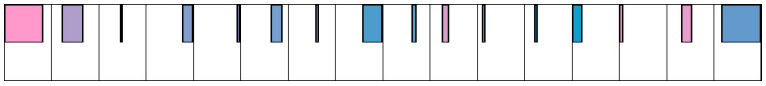
In Abbildung 3.9 ist eine Lösung für 19 Zelte dargestellt, bei der zwei Versetzungen stattfinden.

(a) Die ersten sieben Zelte wurden ohne Probleme platziert.

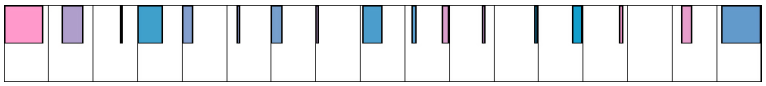
(b) Beim achten Zelt tritt ein Fehler auf.



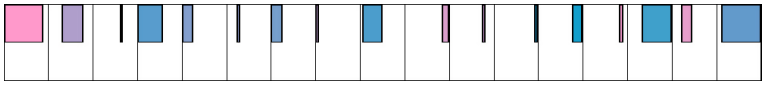
(c) Ein Zelt wurde versetzt, so dass die Zelte nun wieder in den Intervallen liegen.



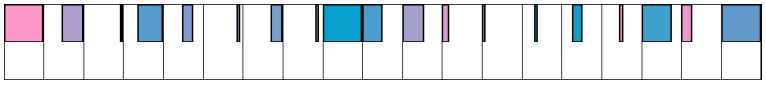
(d) Die nächsten Zeltsetzungen sind problemlos. Hier sind bereits 16 Zelte gesetzt.



(e) Beim 17. Zelt gibt es wiederum ein Problem.



(f) Das Zelt, das vorhin bereits versetzt wurde, wird erneut versetzt, um den Fehler zu beheben.



(g) Endpositionen aller 19 Zelte

Abbildung 3.9: Ein Beispiel mit 19 Zelten und zwei Versetzungen. Es wird hier zweimal das gleiche Zelt versetzt.

Beim Bottom-Up-Ansatz funktioniert die Platzierung ähnlich. Man kommt hier leichter durcheinander, insbesondere, was die Ausgabe der Versetzungen anbelangt. Man muss außerdem aufpassen, dass man beim Umsortieren und Neunummerieren der Zelte schon gelöschte Zelte (die ja für die weitere Berechnung irrelevant, aber wahrscheinlich noch in der Datenstruktur vorhanden sind) ignoriert etc. Abgesehen von solchen kleineren Schwierigkeiten ist die Umsetzung praktisch gleich.

Leider bringt eine Implementierung dieses Ansatzes insgesamt keine großen neuen Erkenntnisse. Wir haben nach und nach berechnet, dass man es mit einer Zeltversetzung schafft, eine Lösung für 18 Zelte zu finden, mit zwei Zeltversetzungen kommt man mindestens bis zu 20

Zelten, und mit dreien mindestens bis 23 Zelte. Aufgrund der doch recht hohen Laufzeit des Algorithmus ist darüber hinaus recht wenig zu sagen. Es ist eventuell möglich, auch für viele Zelte vernünftige Ergebnisse zu erzielen. Insgesamt stößt man jedoch langsam auf eine Grenze, was die Rechenzeit anbelangt.

3.3.5 Umgang mit sturen Besuchern

Leider sind wir noch immer nicht ganz in der Lage, mit unserem Algorithmus die Aufgabenstellung zu erfüllen. Wir müssen uns noch darauf einstellen, dass Besucher unsere Vorschläge eventuell nicht befolgen. Selbst, wenn unser Besucher sich an den Vorschlag hält, ergibt sich eine Änderung für unseren Algorithmus. Er braucht nun nur noch einen eingeschränkten Suchraum durchzugehen, ausgehend von den bereits gesetzten Zelten.

Anpassung des Bottom-Up-Verfahrens

Mit dem rekursiven Bottom-Up-Ansatz wird es nun schwierig. Prinzipiell wäre es möglich, Zelte als nicht löschbar zu markieren, und dann nach einer Lösung zu suchen, bei der nur noch diese Zelte stehen, so dass man die neue Lösung an die alte Lösung „ankleben“ kann. Das Problem entsteht, sobald man auch Zeltversetzungen betrachten will. Denn dann kann man im Prinzip nicht mehr vorgeben, wo die Zelte am Ende stehen, da sie eventuell versetzt werden müssen, um eine Lösung zu finden. Man müsste also irgendwie festlegen, dass Zelte während der Berechnung auf die gewünschten Positionen verschoben werden, und handelt sich damit eine Extraschwierigkeit ein, deren Lösung uns derzeit nicht klar ist.

Anpassung des Top-Down-Verfahrens

Der Top-Down-Algorithmus hat es hier wesentlich leichter. Wir geben schlicht eine durch die Wahl des Benutzers festgelegte Situation vor, setzen die Tiefe entsprechend hoch und lassen den Algorithmus versuchen, die restlichen Zelte zu platzieren. Es ergeben sich jedoch noch zwei Probleme:

Das erste Problem betrifft die Zeltpositionen gesetzter Zelte. Übernehmen wir eine berechnete Lösung und möchten die Zelte tatsächlich setzen, müssen wir davon absehen, ihnen ein ganzes Intervall zuzugestehen, welches weiter verkleinert werden kann. Wir müssen uns vielmehr für eine tatsächliche Position des Zeltes entscheiden. Wie bei den später in 3.4 diskutierten Verfahren kann man ein Zelt zum Beispiel in die Mitte seines bisherigen Intervalls, an den linken oder rechten Rand des Intervalls oder nach einem anderen Verfahren platzieren. Anschließend müssen wir das Zelt anders repräsentieren. Durch einen Punkt (also ein Intervall, dessen Grenzen gleich sind) können wir es nicht darstellen, da dadurch unser Algorithmus durcheinander kommt. Wir möchten eigentlich auch keine punktförmigen Zelte benutzen, siehe 3.3.3. Wir haben uns stattdessen dafür entschieden, Zelte hier durch schmale Intervalle

einer festgelegten Breite darzustellen, die während der weiteren Ausführung nicht verkleinert werden dürfen.

Das zweite Problem ist grundsätzlicher. Wir haben bisher keinen Mechanismus, zu entscheiden, welche Lösung wir jetzt eigentlich nehmen, wenn wir ein Zelt wirklich setzen wollen! Die Algorithmen berechnen viele Lösungen, und diese sind nur bis zur gerade berechneten Tiefe optimal. Treffen wir eine Auswahl, entscheiden wir zwangsläufig greedy.

Man könnte davon ausgehen, dass die Erweiterung unserer Algorithmen es nun auch erlaubt, wirklich mehr als 20 oder 22 Zelte zu betrachten. Das ist im Prinzip richtig, da wir zwischendurch Zelte festlegen und dann von dort aus weiterrechnen und einen großen Teil des früheren Suchraums ignorieren können. Leider kommen wir damit auch nicht sonderlich weit, wir haben zum Beispiel eine Lösung für 19 Zelte festgelegt, konnten aber trotzdem nicht berechnen, ob es eine Lösung für 23 Zelte mit drei weiteren Versetzungen gibt, weil es zu lange dauert.

3.4 Greedy-Algorithmen

Wir wenden uns nun den Greedy-Algorithmen zu. Diese versuchen möglichst gute Lösungen zu erzielen, legen aber mehr Wert darauf, dieses schnell und für viele Zelte zu tun. Sie sind vom Ansatz her einfacher, und ihre Qualität kann man auch mit Ausprobieren ermitteln. Auch Greedy-Algorithmen lassen sich Top-Down oder Bottom-Up programmieren. Implementiert man den Algorithmus aus Abschnitt 3.3.3 mit festen Zeltpositionen (entsprechend der folgenden Verfahren), so erhält man einen Bottom-Up-Greedy-Algorithmus, der leider zusätzlich Rekursion benötigt. Diese kann man ebenfalls aus dem Algorithmus streichen (indem man z.B. einfach festlegt, dass immer das linke von zwei Zelten entfernt wird).

Alternativ, und intuitiver, programmiert man einen Top-Down-Greedy-Algorithmus, indem man nacheinander Zelte nach einem festgelegten Schema in freie Intervalle setzt. Dabei ist es logisch, bei nötigen Zeltversetzungen das Zelt stehenzulassen, was dem Kriterium, nach dem man auch Zelte setzt, mehr entspricht. Wir haben drei Varianten ausprobiert:

3.4.1 Zelte an Intervallgrenzen setzen

Bei dieser Variante setzt man ein Zelt immer an die Grenze eines freien Intervalls. Es macht keinen Unterschied, ob man immer die linke oder immer die rechte nimmt, da das Problem grundsätzlich symmetrisch ist. Man sollte jedoch darauf achten, dass man Zelte auf Grenzen auch dem entsprechenden Intervall zuordnet; es ist recht dumm, immer an die linke Grenze zu setzen, ein auf der Grenze liegendes Zelt hinterher aber wiederum dem linken Intervall zuzuordnen. Muss man ein Zelt versetzen, so nimmt man dasjenige, das von der Intervallgrenze weiter entfernt ist, und setzt es an die Intervallgrenze eines freien Intervalls.

Dieser Ansatz funktioniert sehr schlecht. Die erste Zeltversetzung tritt beim vierten Zelt auf, und für 20 Zelte werden bereits 34 Versetzungen benötigt. Besonders störend ist, dass immer

die gleichen Zelte umziehen müssen, für die der Algorithmus einfach keinen Platz zu finden scheint. ;)

3.4.2 Zelte auf Intervallmitten setzen

Etwas besser ist es, die Mitte eines freien Intervalls zu berechnen und das Zelt dort zu setzen. Wie bei 3.4.1 hat man auch hier den Vorteil, dass Zeltpositionen abhängig von der Zeltanzahl Brüche mit relativ kleinen Zählern und Nennern sind, so dass man mit dieser Darstellungsform keine Probleme bekommt. Muss man ein Zelt versetzen, so rechnet man den Abstand der beiden kollidierten Zelte zur Intervallmitte aus und versetzt das Zelt, das von der Intervallmitte weiter entfernt ist.

Der Algorithmus muss erst beim 13. Zelt versetzen und schafft es, die ersten 20 Zelte mit 19 Zeltversetzungen auf dem Strand zu verteilen. Bei sehr vielen Zelten (ab 500) wird der Algorithmus jedoch ungefähr so schlecht wie der erste.

3.4.3 Maximale Zeltentfernungen

Wesentlich intelligenter ist es, den Abstand eines Zeltes zu seinen beiden Nachbarzelten zu maximieren. Hierfür muss man ein wenig mehr Aufwand betreiben, da man z.B. eine sortierte Liste der Zeltpositionen braucht, in der man die beiden Zelte nachschauen kann, die links und rechts von einem freien Intervall liegen.

Man berechnet dann die Mitte zwischen diesen beiden Nachbarn. Liegt diese Mitte innerhalb des freien Intervalls, kann man sein Zelt dort setzen. Problematisch ist, wenn die Mitte außerhalb liegt. Man kann dann das Zelt auf die entsprechende Intervallgrenze setzen (um es so nah wie möglich an der Mitte zu haben), hat aber ein Problem, falls man Zelte auf Grenzen immer einer Seite zuordnet. In diesem Fall muss man es „leicht entfernt“ von der gefährlichen Grenze plazieren. Muss man ein Zelt versetzen, sucht man sich natürlich das aus, das von der so zu berechnenden Mitte weiter entfernt ist.

Der Algorithmus versetzt bereits beim 11. Zelt, schafft dafür jedoch 20 Zelte mit 7 Versetzungen. Bei großen Zeltanzahlen schlägt dieses Verfahren die beiden ersten deutlich.

3.4.4 Leistungsvergleich

Die folgende Tabelle gibt eine Übersicht über die Leistungsfähigkeit der drei Greedy-Strategien. Hier ist festgehalten, bei welchem Zelt die erste Zeltversetzung auftritt, und wie viele Zeltversetzungen für das Platzieren einer gegebenen Anzahl von Zelten nötig werden.

	Greedy 1	Greedy 2	Greedy 3
erste Versetzung	4	13	11
bis 20 Zelte	34	19	7
bis 50 Zelte	241	176	126
bis 100 Zelte	966	877	558
bis 500 Zelte	25509	26457	13944
bis 1000 Zelte	101263	96297	54927

3.4.5 Vorausschau

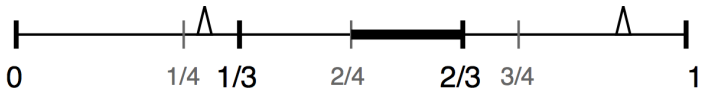


Abbildung 3.10: Vorausschau: Die Platzierung des dritten Zeltes im markierten Teil des freien mittleren Drittels verhindert einen Konflikt im nächsten Schritt, also beim Platzieren des vierten Zeltes.

Als weitere Heuristik (ein auf intuitiven Ideen aufbauendes Lösungsverfahren, das aber nicht gesichert zu optimalen Ergebnissen führt) kann eine Vorausschau eingesetzt werden. Wenn man bei n Zelten für ein (neu zu setzendes oder auch zu versetzendes) Zelt ein freies n-tel hat, kann man den Bereich innerhalb dieses n-tels, in den man das Zelt setzen sollte, durch Vorausschau einschränken: Ein n-tel überschneidet sich mit mehreren $n+1$-teln. Man setzt nun das Zelt in denjenigen dieser Schnittbereiche, in dem keines der anderen schon gesetzten $n-1$ Zelte sitzt (falls das möglich ist). Dann hat man im Schritt $n+1$ ein Problem weniger. Man betrachtet also zukünftige Aufteilungen, aber nur im Hinblick auf Konflikte mit schon gesetzten Zelten. Abbildung 3.10 illustriert die Idee am Beispiel der Platzierung des dritten Zeltes. Es kann auch weiter vorausgeschaut werden: im Schritt n auch schon $n+1$-tel, $n+2$-tel, … $n+k$-tel berücksichtigen. Gegenüber der besten der oben geschilderten Strategien bringt die Vorausschau aber keine wesentlichen Verbesserungen.

3.4.6 Zeltdarstellung

Es sei noch erwähnt, dass wir alle Greedy-Algorithmen zunächst mit Brüchen als Zeltdarstellung programmiert haben. Beim dritten Ansatz gerieten wir jedoch mit den Brüchen in Schwierigkeiten, weil sehr große Nenner auftraten. Wir haben dann mit Fließkommazahlen programmiert. Am Ende wollten wir dann doch noch wissen, ob es vielleicht reicht, im alten Programm eine Kürzen-Funktion zu implementieren und stellten fest, dass das so angepasste Programm andere Ergebnisse lieferte als unser Fließkommazahlenprogramm. Das lag daran, dass die Vergleiche, welches Zelt näher an einer Mitte lag, bei der genaueren Darstellung anders ausfielen. Und tatsächlich kamen auch bessere Ergebnisse heraus: für 50 Zelte brauchte

unsere Brüche-Version nur 118 Versetzungen (im Gegensatz zu 126 bei der Fließkomma-Variante).

3.4.7 Beispielausgabe mit 20 Besuchern

Zwei Ausgaben sind im Aufgabentext explizit verlangt, zum einen ein Ablaufprotokoll mit 20 Besuchern, die sich an die Vorgaben des Programms halten, und zum anderen ein Ablaufprotokoll mit 20 Besuchern, wobei sich der n-te Besucher, falls n ungerade ist, nicht an das Programm hält, sondern sein Zelt nach vorgegebener Regel aufstellt.

Zwanzig brave Besucher

Eine Textausgabe kann in diesem Fall z.B. so aussehen („Zelt bei X aufstellen“ bedeutet dabei, dass das Zelt X Meter von der Westgrenze des Strandes entfernt aufgebaut wird):

```
Zelt 1 bei 500.00 aufstellen.
Zelt 2 bei 750.00 aufstellen.
Zelt 3 bei 250.00 aufstellen.
Zelt 4 bei 875.00 aufstellen.
Zelt 5 bei 125.00 aufstellen.
Zelt 6 bei 625.00 aufstellen.
Zelt 7 bei 375.00 aufstellen.
Zelt 8 bei 937.50 aufstellen.
Zelt 9 bei 62.50 aufstellen.
Zelt 10 bei 562.50 aufstellen.

Zeltversetzung: Zelt 10 abbauen und bei 312.50 aufstellen.
Zelt 11 bei 687.50 aufstellen.

Zeltversetzung: Zelt 11 abbauen und bei 562.50 aufstellen.
Zelt 12 bei 812.50 aufstellen.

Zeltversetzung: Zelt 10 abbauen und bei 187.50 aufstellen.
Zelt 13 bei 437.50 aufstellen.

Zeltversetzung: Zelt 13 abbauen und bei 312.50 aufstellen.
Zelt 14 bei 687.50 aufstellen.
Zelt 15 bei 437.50 aufstellen.
Zelt 16 bei 968.75 aufstellen.
Zelt 17 bei 31.25 aufstellen.
Zelt 18 bei 531.25 aufstellen.

Zeltversetzung: Zelt 18 abbauen und bei 343.75 aufstellen.
Zelt 19 bei 656.25 aufstellen.
```

```
Zeltversetzung: Zelt 13 abbauen und bei 296.88 aufstellen.
Zeltversetzung: Zelt 19 abbauen und bei 531.25 aufstellen.
Zelt 20 bei 781.25 aufstellen.

Es wurden insgesamt 7 Zelte versetzt
```

Abbildung 3.11 stellt diese Lösung grafisch dar.

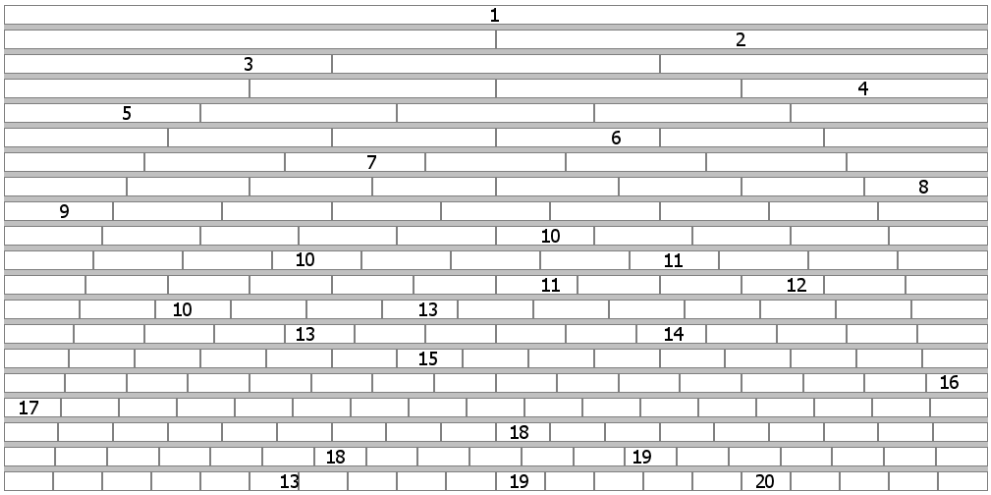


Abbildung 3.11: In jeder Zeile ist die Nummer des neuen Zeltes an der Position des neuen Zeltes zu sehen. Wurde ein Zelt versetzt, so ist dessen Nummer an der neuen Position zu finden. Insgesamt gibt es sieben Versetzungen.

Zehn brave und zehn sture Besucher

Eine Textausgabe kann in diesem Fall z.B. so aussehen:

```
Zelt 1 wird auf Wunsch des Besuchers bei 7.00 aufgestellt.
Zelt 2 bei 503.50 aufstellen.
Zelt 3 wird auf Wunsch des Besuchers bei 673.67 aufgestellt.

Zeltversetzung: Zelt 3 abbauen und bei 255.25 aufstellen.
Zelt 4 bei 751.75 aufstellen.
Zelt 5 wird auf Wunsch des Besuchers bei 807.00 aufgestellt.

Zeltversetzung: Zelt 5 abbauen und bei 379.38 aufstellen.
Zelt 6 bei 875.88 aufstellen.
Zelt 7 wird auf Wunsch des Besuchers bei 578.43 aufgestellt.
```

```
Zeltversetzung: Zelt 2 abbauen und bei 131.13 aufstellen.
Zelt 8 bei 665.09 aufstellen.

Zeltversetzung: Zelt 8 abbauen und bei 937.94 aufstellen.
Zelt 9 wird auf Wunsch des Besuchers bei 451.44 aufgestellt.
Zelt 10 bei 665.09 aufstellen.

Zeltversetzung: Zelt 9 abbauen und bei 478.90 aufstellen.
Zelt 11 wird auf Wunsch des Besuchers bei 279.73 aufgestellt.

Zeltversetzung: Zelt 11 abbauen und bei 193.19 aufstellen.
Zelt 12 bei 708.42 aufstellen.

Zeltversetzung: Zelt 12 abbauen und bei 813.81 aufstellen.
Zelt 13 wird auf Wunsch des Besuchers bei 391.62 aufgestellt.

Zeltversetzung: Zelt 5 abbauen und bei 323.43 aufstellen.
Zelt 14 bei 528.67 aufstellen.

Zeltversetzung: Zelt 9 abbauen und bei 708.42 aufstellen.
Zelt 15 wird auf Wunsch des Besuchers bei 407.00 aufgestellt.

Zeltversetzung: Zelt 15 abbauen und bei 69.06 aufstellen.
Zelt 16 bei 453.13 aufstellen.

Zeltversetzung: Zelt 4 abbauen und bei 968.97 aufstellen.
Zelt 17 wird auf Wunsch des Besuchers bei 595.24 aufgestellt.

Zeltversetzung: Zelt 17 abbauen und bei 388.89 aufstellen.
Zelt 18 bei 736.11 aufstellen.

Zeltversetzung: Zelt 17 abbauen und bei 503.13 aufstellen.
Zeltversetzung: Zelt 7 abbauen und bei 619.62 aufstellen.
Zeltversetzung: Zelt 9 abbauen und bei 789.47 aufstellen.
Zelt 19 wird auf Wunsch des Besuchers bei 270.16 aufgestellt.

Zeltversetzung: Zelt 19 abbauen und bei 224.22 aufstellen.
Zeltversetzung: Zelt 17 abbauen und bei 437.44 aufstellen.
Zelt 20 bei 562.50 aufstellen.

Es wurden insgesamt 17 Zelte versetzt.
```

Abbildung 3.12 stellt diese Lösung grafisch dar.

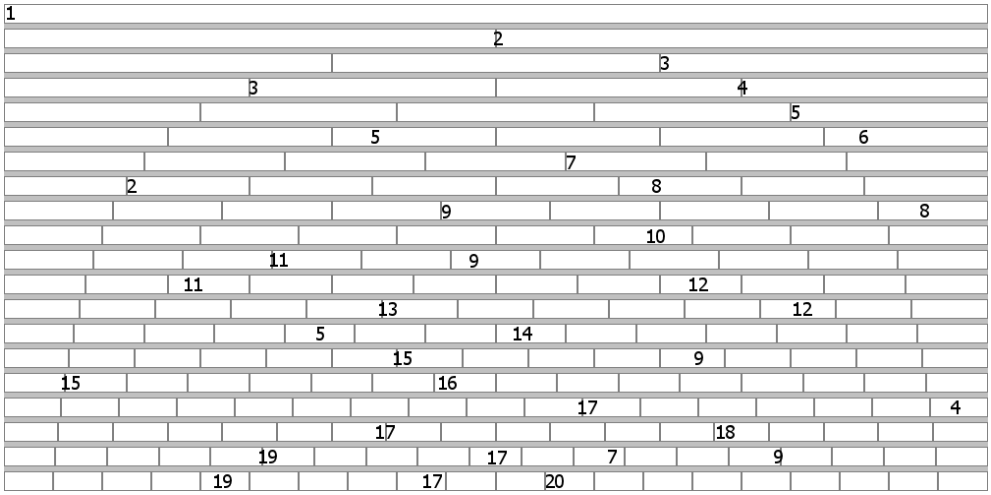


Abbildung 3.12: Dieses Bild ist genauso aufgebaut wie 3.11, allerdings gibt es sehr viel mehr Versetzungen.

3.5 Probleme mit Zeltversetzungen (Teil 4)

Die algorithmischen Probleme, die bei einer vollständigen Suche durch Zeltversetzungen entstehen, haben wir bereits in Abschnitt 3.3.4 diskutiert.

Algorithmische Probleme waren in diesem Teil jedoch nicht gemeint. Es sollten eher praktische Probleme beim Einsatz der Software diskutiert werden, die mit Zeltversetzungen zu tun haben. Wie die Aufgabenstellung sagt, werden Zeltversetzungen gar nicht gerne gesehen. Es wäre also hilfreich, wenn das Programm begründen könnte, warum es am besten oder gar die einzige Möglichkeit ist, ein bestimmtes Zelt zu versetzen. Denkbar ist auch, Stammgäste nach Möglichkeit mit Zeltversetzungen zu verschonen. Im Sinne der prinzipientreuen Gäste von Norderfrierum wäre es wohl, möglichst die zuletzt platzierten Zelte zu versetzen. Wenn man über die Jahre (bzw. Programmläufe) hinweg denkt, können (zumindest bei Stammgästen) Versetzungen der Vorjahre berücksichtigt werden. Dies verlangt aber eine Identifizierung von Zelten mit Gästen und Unterscheidung zwischen Stamm- und anderen Gästen.

3.6 Erweiterungen

Es ist schwierig, diese ohnehin anspruchsvolle Aufgabe noch sinnvoll zu erweitern. Es könnte interessant sein, bei der Lösung des Problems mit randomisierten Algorithmen zu arbeiten, also unter Verwendung des Zufallsprinzips. Der (durchdachte) Einsatz von Zufallsverfahren zusätzlich zu einer „normalen“ Lösung wird als Erweiterung gewertet. Außerdem wurden bei der Aufgabenstellung Abreisen der Zeltbesucher völlig außer Acht gelassen. Die Berücksichtigung von Abreisen ist eine naheliegende, sinnvolle und schwierige Erweiterung.

3.7 Bewertungskriterien

Prinzipiell ist zu sagen, dass das Problem sehr kompliziert ist und funktionierende Lösungen bereits eine Leistung sind. Die Bewertung richtet sich nach den folgenden Kriterien:

In *Aufgabenteil 1* wird eine klare Beschreibung der Anforderungen an das Programm erwartet. Dazu gehört auch eine übersichtliche Auflistung, insbesondere aber eine Unterscheidung nach harten Anforderungen, die unbedingt erfüllt werden müssen, und den wünschenswerten Eigenschaften, also Anforderungen, die nur weniger präzise angegeben werden können. Die beschriebenen Anforderungen und Eigenschaften sollen natürlich sinnvoll sein; insbesondere muss eigenmächtiges Handeln der Besucher berücksichtigt werden.

Die meisten Bewertungspunkte betreffen *Teil 2* bzw. werden anhand der Ausgaben von *Teil 3* überprüft. Für die Bewertung spielt eine Rolle, ob

- ... die Darstellung von Zelten bzw. Strandabschnitten geeignet ist, also insbesondere nicht mit großen Ungenauigkeiten (Gleitkommazahlen) oder hoher Komplexität (Aufteilung in $N!$ Abschnitte) verbunden ist, nicht zu falschen Ergebnissen führt bzw. dazu, dass nur für recht wenige Zelte eine Lösung gefunden werden kann. Solche Effekte werden negativ bewertet. Auch wenn ein Algorithmus mit vollständiger Suche punktförmige Zelte erlaubt und dadurch falsche Lösungen berechnet, ist dies ein Mangel.
- ... Alternativen zur Darstellung von Zelten bzw. Strandabschnitten diskutiert werden. Dies wird nicht erwartet, kann aber zu Pluspunkten führen.
- ... wirklich immer N Zelte sauber auf alle N-tel des Strandes verteilt werden.
- ... der Algorithmus aus anderen Gründen fehlerhafte Lösungen liefert. Alle Lösungen, die mehr als 17 Zelte ohne Versetzungen setzen können, sind (mit großer Wahrscheinlichkeit) falsch.
- ... das Programm ausreichend effizient arbeitet; mindestens 20 Zelte waren zu platzieren.
- ... die Qualität der Lösungen stimmt. Greedy-Verfahren wie die ersten beiden beschriebenen sind zu einfach, der dritte Greedy-Ansatz ist akzeptabel.

In *Teil 3* müssen mindestens die Ausgaben für die beiden geforderten Fälle vorhanden sein. Diese sollten vollständig und übersichtlich sein. Für weitere Beispiele kann es Pluspunkte geben, wenn diese geeignet sind, das Funktionieren des Programms weiter zu verdeutlichen.

In *Aufgabenteil 4* sollten insbesondere praktische Probleme mit Zeltversetzungen im Zusammenhang mit dem Einsatz der Software erläutert werden. Hierzu werden die Schilderung einiger plausibler Probleme und die Beschreibung gut begründeter Lösungen erwartet.

Perlen der Informatik – aus den Einsendungen

Allgemeines

Worte des Wettbewerbs: fünfseitenlang, radikaler Einschlag, empirische Wahrscheinlichkeit, Optimizing Algorithem, eingestelltbar, irrelwand, Komplexizität

Arbeit mit Pointern ist wie Stricken im Dunkeln.

. . . dass ein solches Verfahren ein fakultatives Wachstum besitzt.

Ich bin mir vollkommen bewusst, dass dieses Vorgehen irgendwo zwischen unsauberer Mathematik und schwarzer Magie liegt.

Liste der Erweiterungen, also known as „Betteln um Zusatzpunkte“

„Object-oriented programming is an exceptionally bad idea, which could only have originated in California.“ *(angebliches Zitat von Dijsktra, in der 63. von 71 Fußnoten der Einsendung wiedergegeben)*

Quellcode (leider ohne Syntax, da Jbuilder die nicht exportieren wollte)

Windows Binaries kann ich Gott sei Dank nicht erzeugen.

Aufgabe 1

Riechungsvektor

Peripheriezentriwinkelsatz

Dabei wähle immer eine existierende Kugel und einen Punkt auf ihrem Radius.

Der Zeitaufwand steigt grob quadratisch, sind es bei der 10. hinzugefügten Kugel 10 Überprüfungen, so hat man bei der 1000. schon stolze 1000 Stück.

Bei diesem Ansatz treten aber aufgrund dessen, dass Pixel natürlich ganzzahlig sind, und Radien, die sehr viele Pixel groß sind, das Gesamtbild sehr schnell sehr viel größer machen.

Die Klasse ’even’ stellt die unendlich große zweidimensionale Ebene dar.

Natürlich ist eine exakte Bestimmung dank der begrenzten Variablengenauigkeit niemals wirklich exakt. Deshalb bezieht sich „exakt“ eher auf mathematisch-ästhetische Exaktheit.

Aufgabe 2

Meistermasteranschlussknoten

GOTE SAN seine Unterstruktur *Ja, ja: Der Dativ ist dem Genitiv sein Tod.*

Die Kanten eines Knotens sind dabei aufsteigend in Adjutanz-Listen gespeichert.

Es gab dabei einige Dinge zu beachten: Webseiten können unterschiedliche Namen haben.

Das Passwort ist fest zugewiesen und lautet: Gote_San_seine_Nachricht.

... die Anwendung des folgenden Satzes der Zahlentheorie, der schon vor mehreren hundert Jahren bekannt war: Man kann eine gegebene Zahl mit Hilfe von n so manipulieren, dass die Veränderung nur wieder rückgängig gemacht werden kann, wenn man die Zahl kennt.

```
p4r4n0!d3 könn3n 31337-5<hr!ft v3rw3nd3n. D!3 N4<hr!<h7 !57 d4dir<h
n!ch7 w3n!93r gu7 135b4r, 4b3r d3u71!<h 5<hw3r3r 2u 3nt5<h1ü5531n.
```

GOTE SAN lässt die verschlüsselten 4 Bytes von einem Boten auswendig lernen.

$3 = 0$ ist die Beschreibung für ein Dreieck, $4 = 0$ für ein Viereck etc.

Einen Algorithmus zu finden, der dieses Problem in polynomialer Zeit löst, würde implizieren, dass P=NP gilt. Selbst meine Arroganz ist nicht so groß, dass ich ernsthaft versuchen würde, einen solchen Algorithmus zu finden.

Aufgabe 3

Zeltler

Das Versetzen von Zelten ist natürlich eine komplizierte Angelegenheit. Die meisten Menschen haben ja schon beim Aufstellen eines Zeltes Probleme.

... und alle möglichen Umsetzmöglichkeiten gleichzeitig betrachten.

Insgesamt werden 0 Besucher verrückt.

Schon allein die Tatsache, dass man nicht vor Saisonende abreisen kann, ist für jeden Feriengast eine Zumutung.

Schlecht sitzende Rentner werden aussortiert.

Denkt man weiter, so stellt man fest, dass für das zweite Zelt müssen daher nur alle Primzahlen von 2 bis n getestet werden.