Lösung „Eisbudendilemma“

Lars Müller, Teilnahme-ID 58886

Lösungsidee

Man beginnt mit einer naiven Brute-Force: Probiere alle möglichen Eisbudenstandorte; prüfe für jeden, ob er stabil ist - also gegen alle anderen möglichen Standorte in einer Abstimmung gewinnen würde.

Die Komplexität hiervon wäre entsprechend polynomiell $u^6 n$ für u Umfang und n Anzahl der Adressen: Das Probieren aller Standorte ist u^3; für jeden Standort müssten wieder alle Standorte durchgegangen werden, also u^6. Schließlich müssen jeweils die beiden Standorte verglichen werden. Hierfür müssen alle n Adressen iteriert werden.

Erste Optimierung: Gehe die Standorte i, j, k so durch, dass $i < j < k$ gilt. Anstatt von u^6 Schleifendurchläufen erhalten wir so $(\frac{u(u-1)(u-2)}{3!})^2$ Schleifendurchläufe.

Zweite Optimierung: Gehe i, j, k so durch, dass zwischen i und j sowie j und k jeweils mindestens ein Drittel der Häuser liegen. Beweis: Zwischen i und j (beide exklusive) befinden sich a Häuser, zwischen j und k b, sowie zwischen k und i c. Seien i, j, k so gewählt, dass $a \geq b \geq c$. Dann verlegen wir einfach i und j auf einander zu, so dass alle a Häuser für die Verlegung stimmen würden. Wir können nun k so verlegen, dass auch mindestens die Hälfte der $\lfloor \frac{b}{2} \rfloor$ Häuser für die Verlegung stimmen würden. Für $a > \lfloor \frac{n}{3} \rfloor$ ist die Summe der Stimmen offensichtlich eine Mehrheit, die Standorte i, j, k können nicht stabil sein.

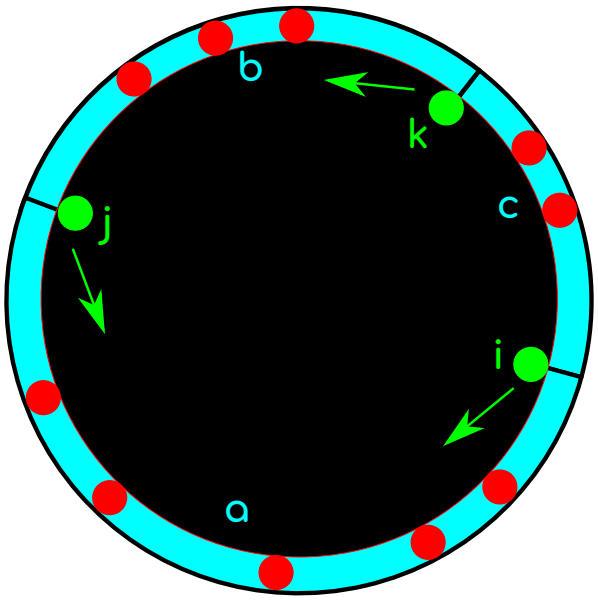

Abbildung 1: Instabile Standorte aufgrund ungleichmäßiger Verteilung der Häuser zwischen den Eisbuden: Pfeile geben Verlegungen / Verschiebungen an, die in einer Abstimmung - wenn alle kombiniert - eine Mehrheit bekämen

Wir haben allerdings Häuser, die exakt gleiche Adressen wie die Eisdielen besitzen, vernachlässigt; diese würden womöglich alle drei gegen die Verlegung stimmen. Entsprechend addieren wir einfach 3: Für $a > \lfloor \frac{n}{3} \rfloor + 3$ können die Standorte nicht stabil sein.

Die Prüfung, ob die Standorte stabil sind, lässt sich mithilfe einer Serie von Tests weiter optimieren. Sobald die zu testenden Standorte gegen einen anderen Standortvorschlag verloren haben, kann die Prüfung abgebrochen werden:

1. Probiere Verlegungen der Standorte um eine Adresse in oder gegen den Uhrzeigersinn
2. Probiere gegenüber von Häusern platzierte Eisbudenstandorte.
3. Probiere schließlich alle anderen Eisbudenstandorte gemäß der zweiten Optimierung.

Bestehen Standorte die dritte Prüfung, sind sie stabil.

Sobald man eine Lösung gefunden hat, ist man fertig und kann abbrechen. Möchte man allerdings die „beste“ Lösung - mit einer minimalen Summe der Abstände der Dorfbewohner zu den nächsten Eisdielen finden - müssen weiter Lösungen gesucht werden. Dennoch steigt die Laufzeit nicht zu sehr, denn man erhält die besagte Summe auch als Abbruchkriterium.

Umsetzung

Kompilierung

Go 1.13 oder neuer benötigt. `go build` (erzeugt eine ausführbare Datei namens `Eisbudendilemma`) oder `go build main.go` (nennt die Datei `main`). Die vorkompilierte ausführbare Datei ist für Linux.

Verwendung

`go run main.go <pfad> [beste-loesung] [png-ausgabe-pfad]` oder `./main <pfad> [beste-loesung] [png-ausgabe-pfad]`: Erstes Argument ist der Pfad zur Datei. Das zweite Argument ist optional und gibt an, ob die beste Lösung ermittelt werden soll; Standardwert ist `j` für „ja“, auch möglich ist `n` für „nein“. Das dritte Argument ist ebenfalls optional und gibt einen Pfad für eine Bildausgabe an; wird es weggelassen, wird kein Bild erstellt. Wird das dritte Argument angegeben, muss auch das Zweite angegeben werden.

Beispiel: `./main beispieldaten/eisbuden0.txt`

Ausgabeformat

```
Keine Lösung möglich
```

oder

```
Lösung: Adressen [...]
```

oder (wenn `beste-loesung` auf „ja“ gesetzt ist)

```
Lösung: Adressen [...] mit Summe der Abstände ...
```

Bibliotheken

Go-Sprachbibliotheken („builtins“): * `bufio`: `Scanner` * `fmt`: Ausgabe, Formattierung * `math`: Maxima * `os`: Argumente * `sort`: Sortierung der Adressen (keine Reihenfolge zugesichert), binäre Suche * `strconv`: Zahl-String-Konversion * `strings`: Stringverarbeitung („splitten“) * `sync`: Parallelisierung (`WaitGroup`, `Mutex`)

Externe Bibliothek (Installation über `go get github.com/fogleman/gg`): * `github.com/fogleman/gg`: Zeichenbibliothek

Parallelisierbarkeit

Als optimierte Brute-Force lässt sich das Verfahren gut parallelisieren: Für jede Start-Eisbude mit Adresse i wird eine parallel laufende „Goroutine“ (eine Art „leichtgewichtiger Thread“) gestartet, in der Eisbudenstandorte i, j, k probiert werden.

Ablauf

Das Ausprobieren erfolgt parallel. Die Funktion `istLoesung` bestimmt möglichst schnell, ob die übergebenen Standorte instabil sind; `getNaechsteAdresse` ermittelt zu einer Eisbuden-Adresse die nächste Adresse, bei der beginnend wieder eine Eisbude gesetzt werden kann, so dass die notwendigen Abstände eingehalten werden. Lösungen benötigen einen „Mutex“, um nacheinander verglichen zu werden; ein „paralleler“ Vergleich könnte zu „race-condition“ Fehlern führen. Soll die erstgefundene Lösung ausgegeben (und gezeichnet) werden, verhindert der Mutex die Ausgabe (und Zeichnung) weiterer Lösungen.

Quellcode

main.go

```
1  package main
2
3  import (
4          "bufio"
5          "fmt"
6          "math"
7          "os"
8          "sort"
9          "strconv"
10         "strings"
11         "sync"
12
13         "github.com/fogleman/gg"
14 )
15
16 // Liest Umfang und Adressen aus einer Datei
17 func einlesen(pfad string) (int, []int) {
```

```
18          file, err := os.Open(pfad)
19          if err != nil {
20                  panic(err)
21          }
22          defer file.Close()
23
24          scanner := bufio.NewScanner(file)
25          // Umfang einlesen
26          scanner.Scan()
27          umfang, err := strconv.Atoi(strings.Split(scanner.Text(), " ")[0])
28          if err != nil {
29                  panic(err)
30          }
31          // Adressen einlesen
32          scanner.Scan()
33          _addressen := strings.Split(scanner.Text(), " ")
34          addressen := make([]int, len(_addressen))
35          for index, addresse := range _addressen {
36                  var err error
37                  addressen[index], err = strconv.Atoi(addresse)
38                  if err != nil {
39                          panic(err)
40                  }
41          }
42          if err := scanner.Err(); err != nil {
43                  panic(err)
44          }
45          return umfang, addressen
46  }
47
48  const radius float64 = 300
49  const rand float64 = 50
50  const punktradius float64 = 2
51  func speicherePNG(pfad string, umfang int, adressen, eisbuden []int) {
52          zentrum := radius + rand
53          dc := gg.NewContext(int(zentrum * 2), int(zentrum * 2))
54          dc.DrawCircle(zentrum, zentrum, radius)
55          dc.SetRGB(0, 0, 0)
56          dc.Fill()
57          // Bestimmt die Koordinaten eines Kreispunkts
58          kreispunkt := func(adresse int, radius float64) (float64, float64) {
59                  winkel := float64(adresse) * 2 * math.Pi / float64(umfang)
60                  posX := math.Cos(winkel) * radius + zentrum
61                  posY := math.Sin(winkel) * radius + zentrum
62                  return posX, posY
```

```
63          }
64          // Adressen zeichnen
65          for _, adresse := range adressen {
66                  posX, posY := kreispunkt(adresse, radius + punktradius)
67                  dc.DrawCircle(posX, posY, punktradius)
68                  dc.SetRGB(1, 0, 0)
69                  dc.Fill()
70                  posX, posY = kreispunkt(adresse, radius + rand/4)
71                  dc.DrawStringAnchored(strconv.Itoa(adresse), posX, posY,
    ↪  0.5, 0.5)
72          }
73          // Eisbuden zeichnen
74          for _, adresse := range eisbuden {
75                  posX, posY := kreispunkt(adresse, radius - punktradius)
76                  dc.DrawCircle(posX, posY, punktradius)
77                  dc.SetRGB(0, 1, 0)
78                  dc.Fill()
79                  posX, posY = kreispunkt(adresse, radius - rand/4)
80                  dc.DrawStringAnchored(strconv.Itoa(adresse), posX, posY,
    ↪  0.5, 0.5)
81          }
82          // Speichern
83          dc.SavePNG(pfad)
84  }
85
86  func main() {
87          // Argumente
88          if len(os.Args) < 2 || len(os.Args) > 4 {
89                  println("Verwendung: <pfad> [beste-loesung]
                     ↪  [png-ausgabe-pfad]")
90                  return
91          }
92          ermittle_beste_loesung := true
93          if len(os.Args) >= 3 {
94                  if os.Args[2] != "j" && os.Args[2] != "n" {
95                          println("Verwendung: <pfad> [beste-loesung]
                             ↪  [png-ausgabe-pfad]")
96                          return
97                  }
98                  ermittle_beste_loesung = os.Args[2] == "j"
99          }
100         // Aufgabenstellung einlesen
101         umfang, adressen := einlesen(os.Args[1])
102         if len(adressen) <= 3 {
103                 // Triviale Lösung
```

```
104             fmt.Println("Lösung:", adressen)
105             return
106     }
107     // Es wird nicht garantiert, dass die Adressen sortiert sind.
108     sort.Ints(adressen)
109     // Drehe so, dass erste Adresse 0 ist
110     drehung := adressen[0]
111     for i := range adressen {
112             adressen[i] -= drehung
113     }
114
115     // Abstand zweier Adressen
116     abstand := func(a, b int) int {
117             abs := a - b
118             if abs < 0 {
119                     // Betrag
120                     abs = -abs
121             }
122             andersrum := umfang - abs
123             if andersrum < abs {
124                     // Weg ist kürzer in entgegengesetzter Richtung
125                     return andersrum
126             }
127             return abs
128     }
129
130     // Bestimmt den Abstand einer Adresse zur nächsten Eisbude
131     minAbstand := func(adresse int, eisbuden []int) int {
132             min_abs := math.MaxInt32
133             for _, eisbude := range eisbuden {
134                     abs := abstand(adresse, eisbude)
135                     if abs < min_abs {
136                             min_abs = abs
137                     }
138             }
139             return min_abs
140     }
141
142     // Summe der Abstände
143     summeMinAbstaende := func(eisbuden []int) int {
144             summe := 0
145             for _, adresse := range adressen {
146                     summe += minAbstand(adresse, eisbuden)
147             }
148             return summe
```

```
149         }
150
151         // Bestimmt, ob andere_eisbuden gegen eisbuden in einer Abstimmung
             ↪  gewinnen würde
152         istBesser := func(andere_eisbuden, eisbuden []int) bool {
153                 stimmen := 0
154                 for _, adresse := range adressen {
155                         if minAbstand(adresse, andere_eisbuden) <
                             ↪  minAbstand(adresse, eisbuden) {
156                                 // Verkürzung, Stimme dafür
157                                 stimmen++
158                         } else {
159                                 // Stimme dagegen
160                                 stimmen--
161                         }
162                 }
163                 // Mehr Ja- als Nein-Stimmen
164                 return stimmen > 0
165         }
166
167         // So viele Adressen müssen mindestens zwischen zwei Eisbuden liegen
168         min_adressen_zwischen_eisbuden := len(adressen) / 3 - 3
169         if min_adressen_zwischen_eisbuden < 0 {
170                 min_adressen_zwischen_eisbuden = 0
171         }
172         getNaechsteAdresse := func(adresse int) int {
173                 // Binäre Suche: Bestimmt, hinter der wievielten Häusern die
                     ↪  gegebene Adresse liegt
174                 i := sort.SearchInts(adressen, adresse)
175                 if i >= len(adressen) {
176                         // Adresse liegt hinter dem letzten Haus
177                         return umfang
178                 }
179                 i += min_adressen_zwischen_eisbuden
180                 if i >= len(adressen) {
181                         return umfang
182                 }
183                 if adresse <= adressen[i] {
184                         return adresse + 1
185                 }
186                 return adressen[i]
187         }
188
189         // Ermittelt, ob eine Eisbudenverteilung stabil ist
190         // Um dies zu beschleunigen, werden zahlreiche "early-returns" genutzt:
```

```
191         // Es wird versucht, möglichst schnell Standorte zu finden,
192         // gegen die die gegebenen Standorte verlieren würden
193         istLoesung := func(eisbuden []int) bool {
194                 // Häufig bessere Standorte: Probiere simple Verschiebungen
195                 for i := -1; i < 2; i++ {
196                         for j := -1; j < 2; j++ {
197                                 for k := -1; k < 2; k++ {
198                                         if istBesser([]int{
199                                                 (eisbuden[0]+i) % umfang,
200                                                 (eisbuden[1]+j) % umfang,
201                                                 (eisbuden[2]+k) % umfang,
202                                         }, eisbuden) {
203                                                 return false
204                                         }
205                                 }
206                         }
207                 }
208                 // Eisbudenstandorte bei Adressen probieren
209                 // Häufig besser
210                 for i := 0; i < len(adressen); i++ {
211                         for j := i + 1; j < len(adressen); j++ {
212                                 for k := j + 1; k < len(adressen); k++ {
213                                         andere_eisbuden :=
    ↪  []int{adressen[i], adressen[j], adressen[k]}
214                                         if istBesser(andere_eisbuden,
                                             ↪  eisbuden) {
215                                                 return false
216                                         }
217                                 }
218                         }
219                 }
220                 // Eisbudenstandorte mit Abstand untereinander probieren
221                 for i := 0; i < umfang; i++ {
222                         for j := getNaechsteAdresse(i); j < umfang; j++ {
223                                 for k := getNaechsteAdresse(j); k < umfang;
                                     ↪  k++ {
224                                         andere_eisbuden := []int{i, j, k}
225                                         if istBesser(andere_eisbuden,
                                             ↪  eisbuden) {
226                                                 return false
227                                         }
228                                 }
229                         }
230                 }
```

```
231                 // Keine Standorte gefunden, gegen die die gegebenen Standorte
                     ↪  verlieren würden
232                 return true
233         }
234
235         var wg sync.WaitGroup
236         var mutex sync.Mutex
237         min_summe := math.MaxInt32
238         var beste_loesung []int
239         ausgabe := func() {
240                 // Zurück drehen
241                 for i := range adressen {
242                         adressen[i] += drehung
243                 }
244                 for i := range beste_loesung {
245                         beste_loesung[i] += drehung
246                 }
247                 if ermittle_beste_loesung {
248                         fmt.Println("Lösung: Adressen", beste_loesung, "mit
    ↪  Summe der Abstände", min_summe)
249                 } else {
250                         fmt.Println("Lösung: Adressen", beste_loesung)
251                 }
252                 if len(os.Args) > 3 {
253                         speicherePNG(os.Args[3], umfang, adressen,
    ↪  beste_loesung)
254                 }
255         }
256         // Probiert Eisbudenstandorte aus, bei denen die erste Bude Adresse i
             ↪  hat
257         probiere := func(i int) {
258                 defer wg.Done()
259                 // Probiere so aus, dass i < j < k gilt
260                 // und zwischen i und j sowie j und k ausreichend Häuser liegen
261                 for j := getNaechsteAdresse(i); j < umfang; j++ {
262                         for k := getNaechsteAdresse(j); k < umfang; k++ {
263                                 eisbuden := []int{i, j, k}
264                                 summe := -1
265                                 if ermittle_beste_loesung {
266                                         summe = summeMinAbstaende(eisbuden)
267                                 }
268                                 if summe < min_summe && istLoesung(eisbuden)
                                     ↪  {
269                                         // Mutex "sperren":
```

```
270                                             // Wird währenddessen eine weitere
                                                 ↪  Lösung gefunden,
271                                             // soll diese warten - die Lösungen
272                                             // sollen hintereinander
                                                 ↪  durchgegangen werden
273                                             mutex.Lock()
274                                             beste_loesung = eisbuden
275                                             min_summe = summe
276                                             if !ermittle_beste_loesung {
277                                                     // Erste gefundene Lösung
                                                         ↪  ausgeben
278                                                     ausgabe()
279                                                     // Fertig
280                                                     os.Exit(0)
281                                             }
282                                             // Mutex entsperren
283                                             mutex.Unlock()
284                                     }
285                             }
286                     }
287             }
288             for i := 0; i < umfang; i++ {
289                     wg.Add(1)
290                     go probiere(i)
291             }
292             // Warte auf das Ausprobieren aller infragekommender Standorte
293             wg.Wait()
294             if min_summe != math.MaxInt32 {
295                     // Lösung gefunden: min_summe hat nicht mehr den Maximalwert
296                     ausgabe()
297             } else {
298                     fmt.Println("Keine Lösung möglich")
299             }
300     }
```

Beispiele

Die verstrichene Zeit wurde nicht vom Programm selber, sondern von einem Shellscript gemessen.

eisbuden0.txt

Zusätzliches Beispiel:

```
10 9
0 1 2 3 4 5 6 7 8 9
```

```
Lösung: Adressen [3 6 9] mit Summe der Abstände 8

Zeit verstrichen: 0m0,081s
```

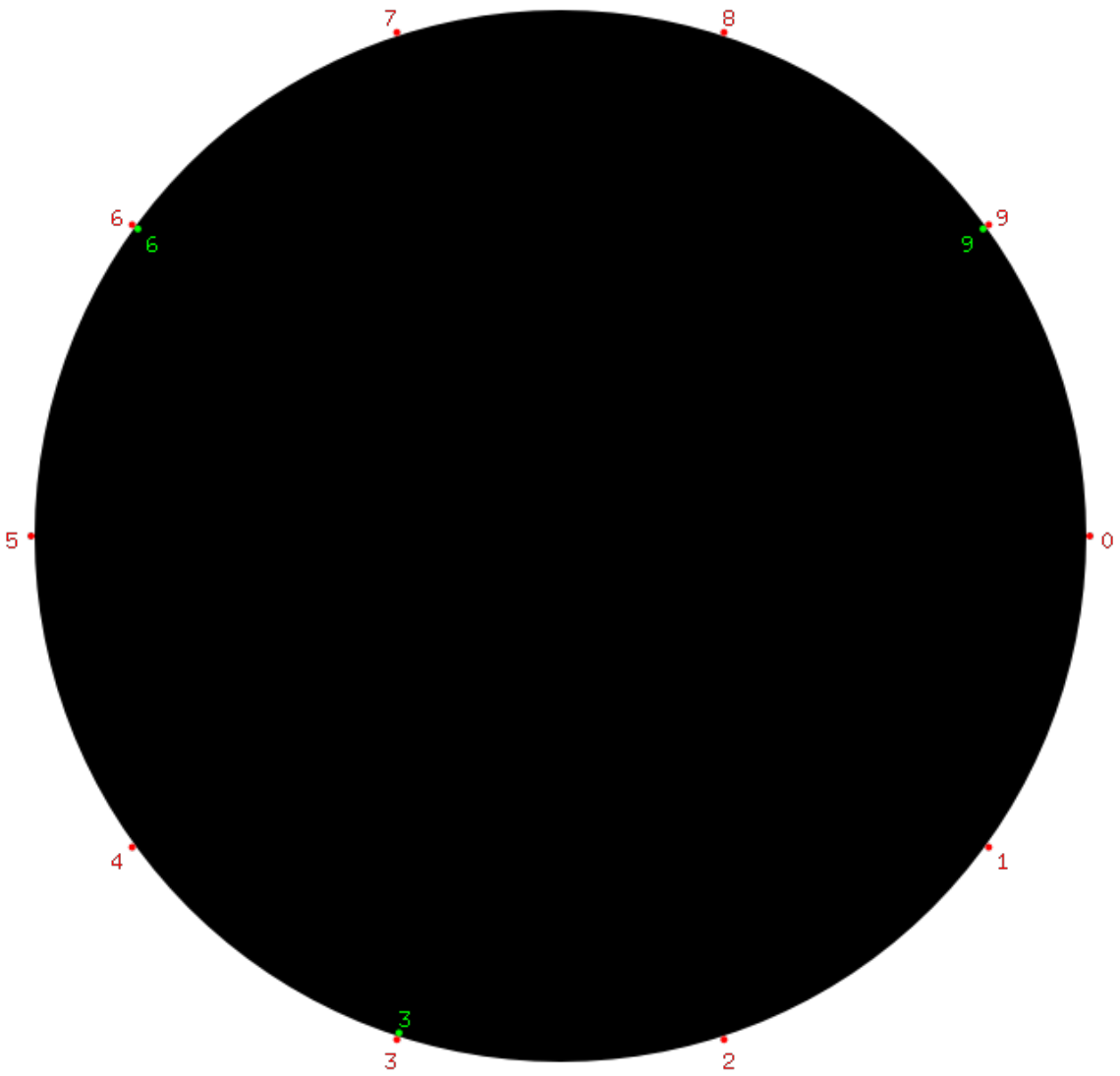

Abbildung 2: Eisbudenstandorte

Das Beispiel dient als „sanity-check“: Stabile Eisbudenstandorte liegen auf der Hand.

eisbuden1.txt

```
Lösung: Adressen [2 8 14] mit Summe der Abstände 6
```

```
Zeit verstrichen: 0m0,101s
```

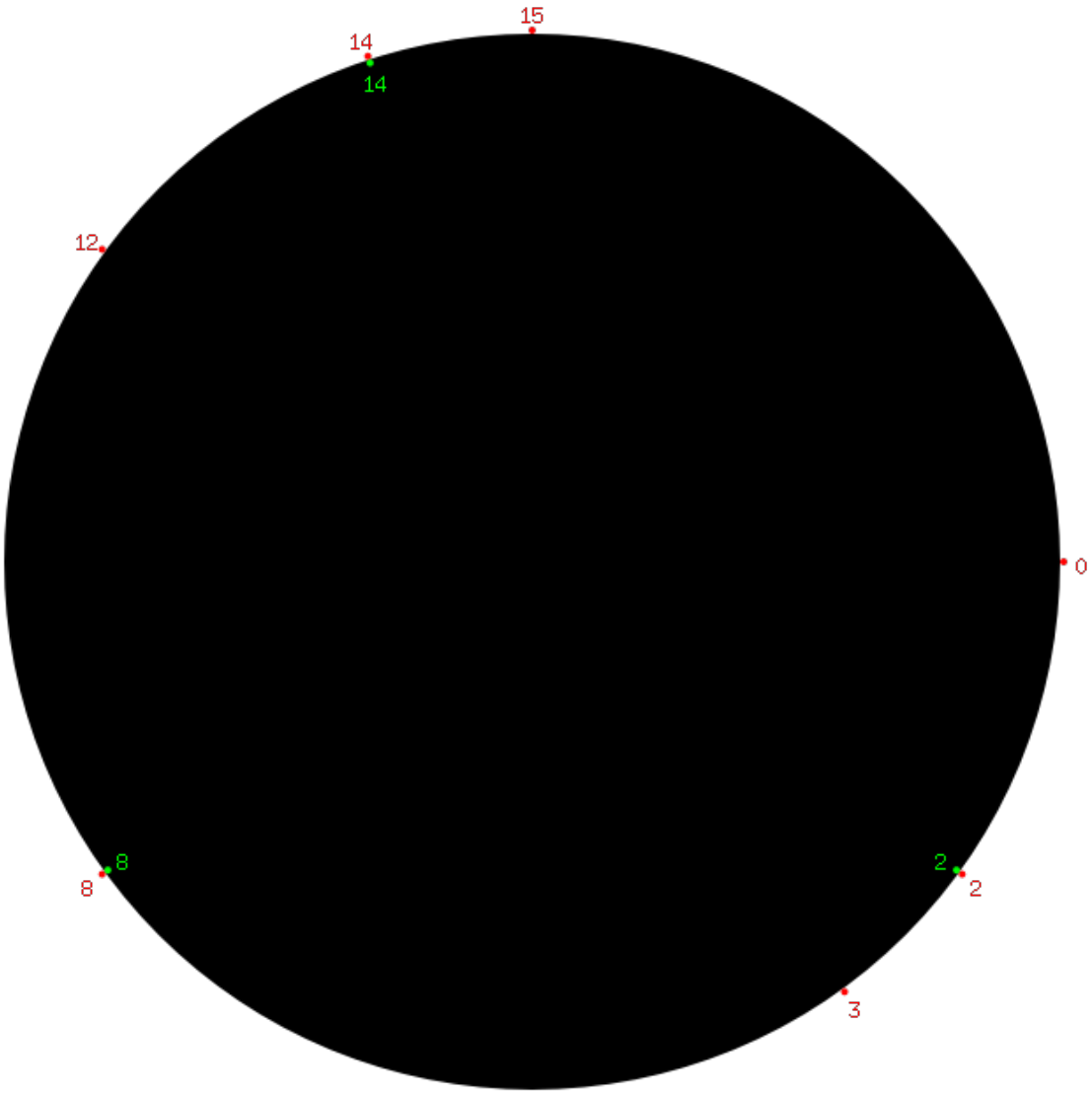

Abbildung 3: Eisbudenstandorte

eisbuden2.txt

```
Keine Lösung möglich

Zeit verstrichen: 0m0,018s
```

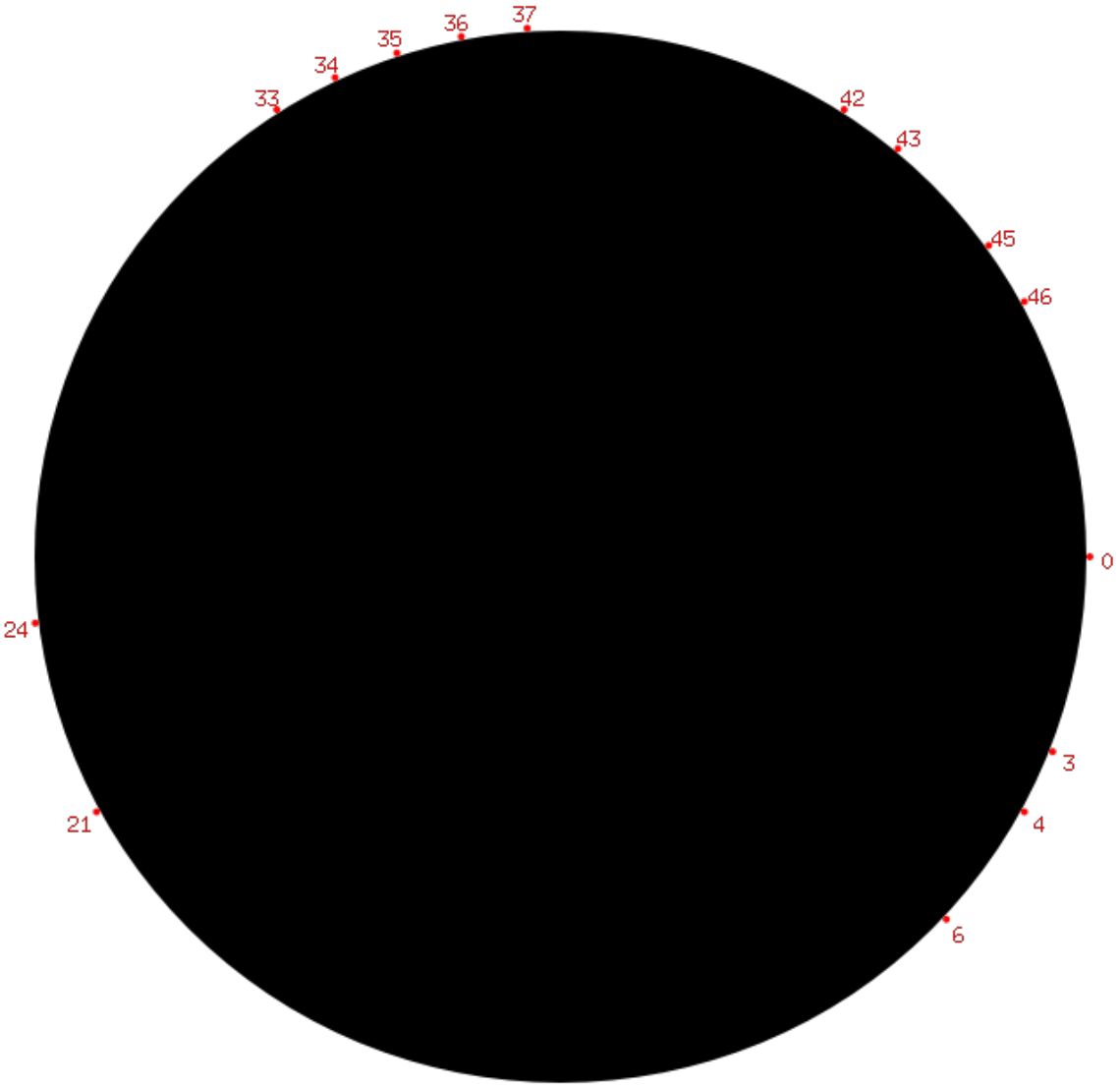

Abbildung 4: Adressen

eisbuden3.txt

```
Lösung: Adressen [17 42 50] mit Summe der Abstände 60

Zeit verstrichen: 0m0,121s
```

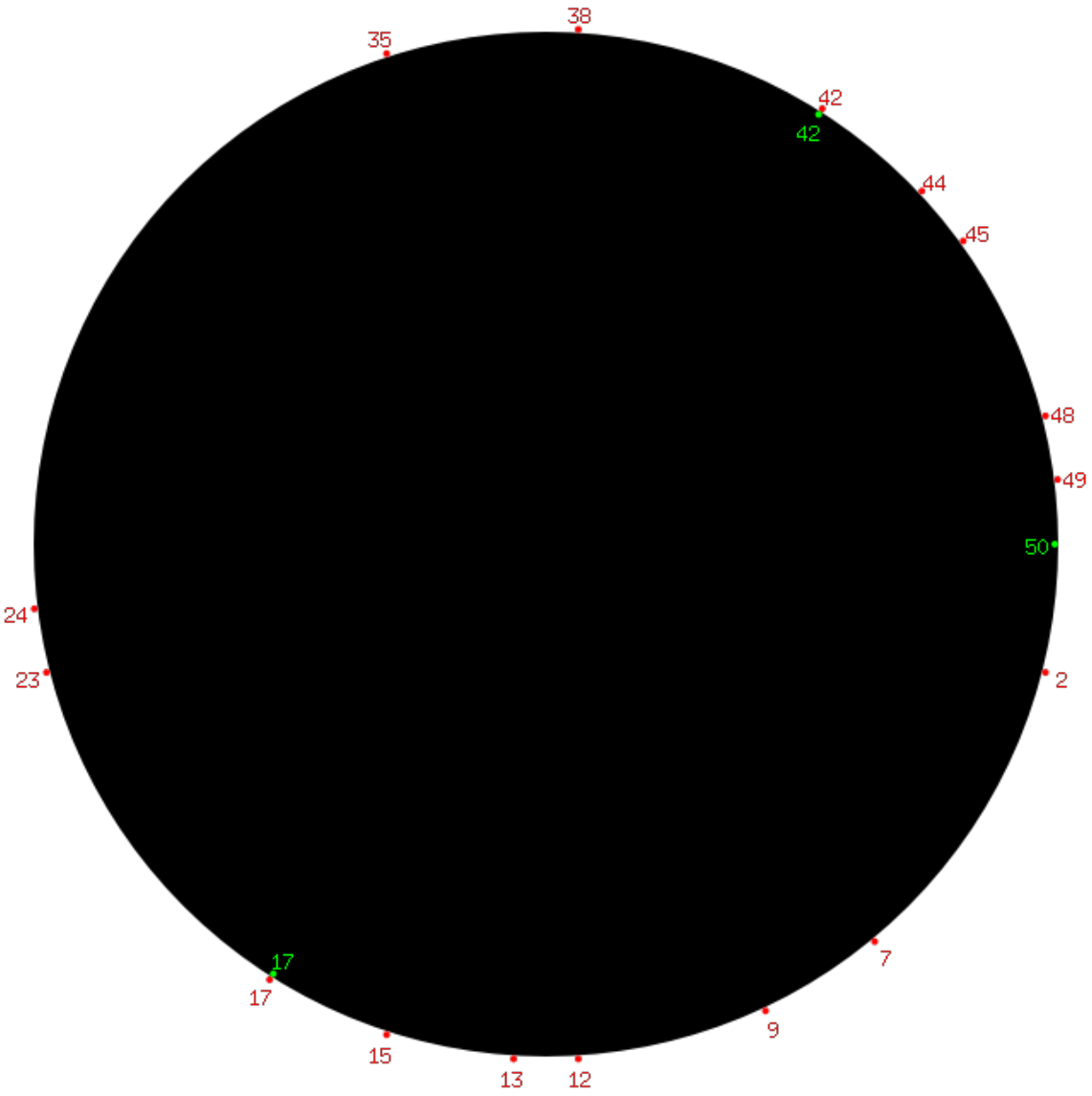

Abbildung 5: Eisbudenstandorte

eisbuden4.txt

```
Keine Lösung möglich
```

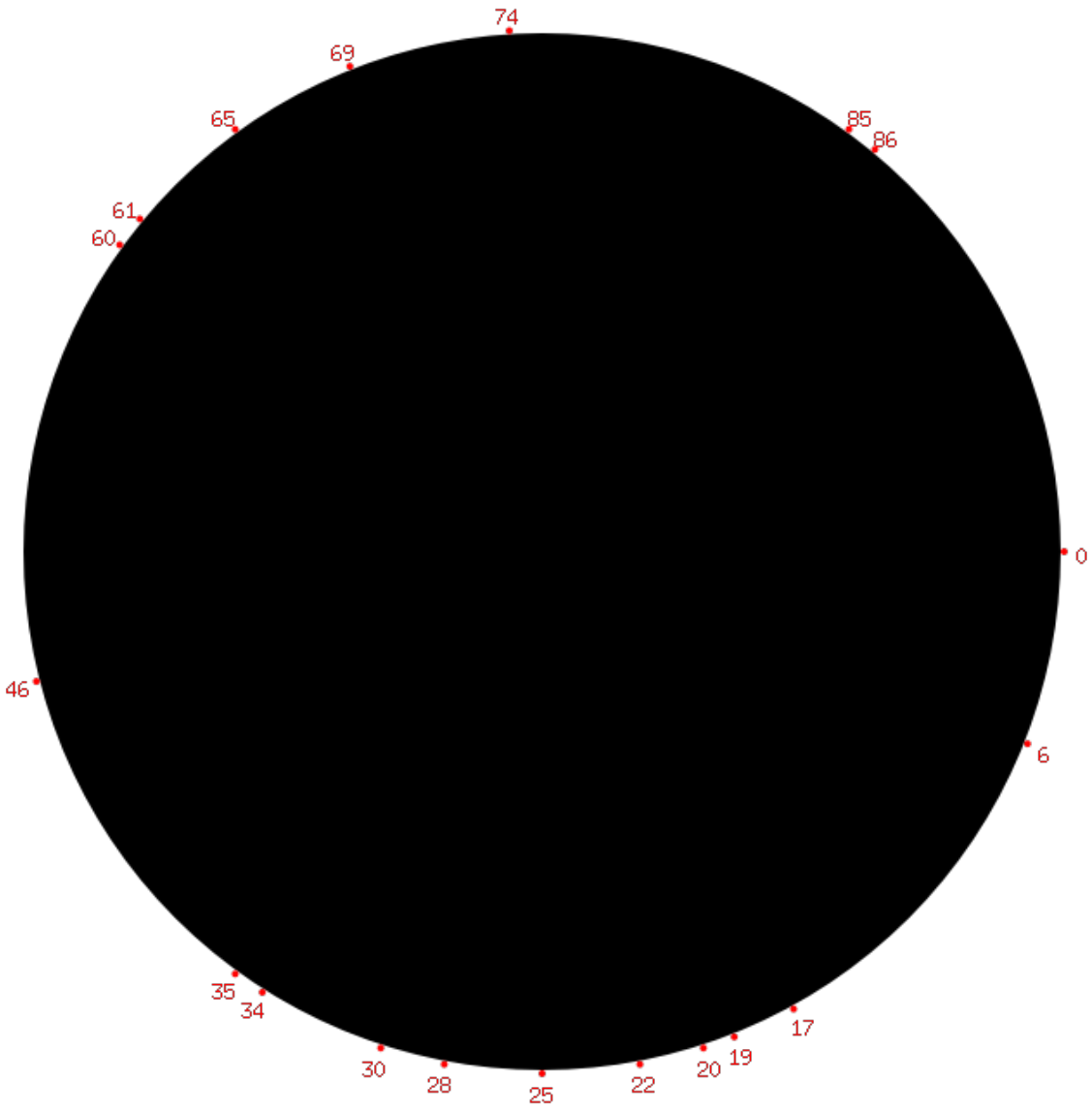

Abbildung 6: Adressen

Zeit verstrichen: 0m0,120s

eisbuden5.txt

Lösung: Adressen [83 128 231] mit Summe der Abstände 406

Zeit verstrichen: 0m3,980s

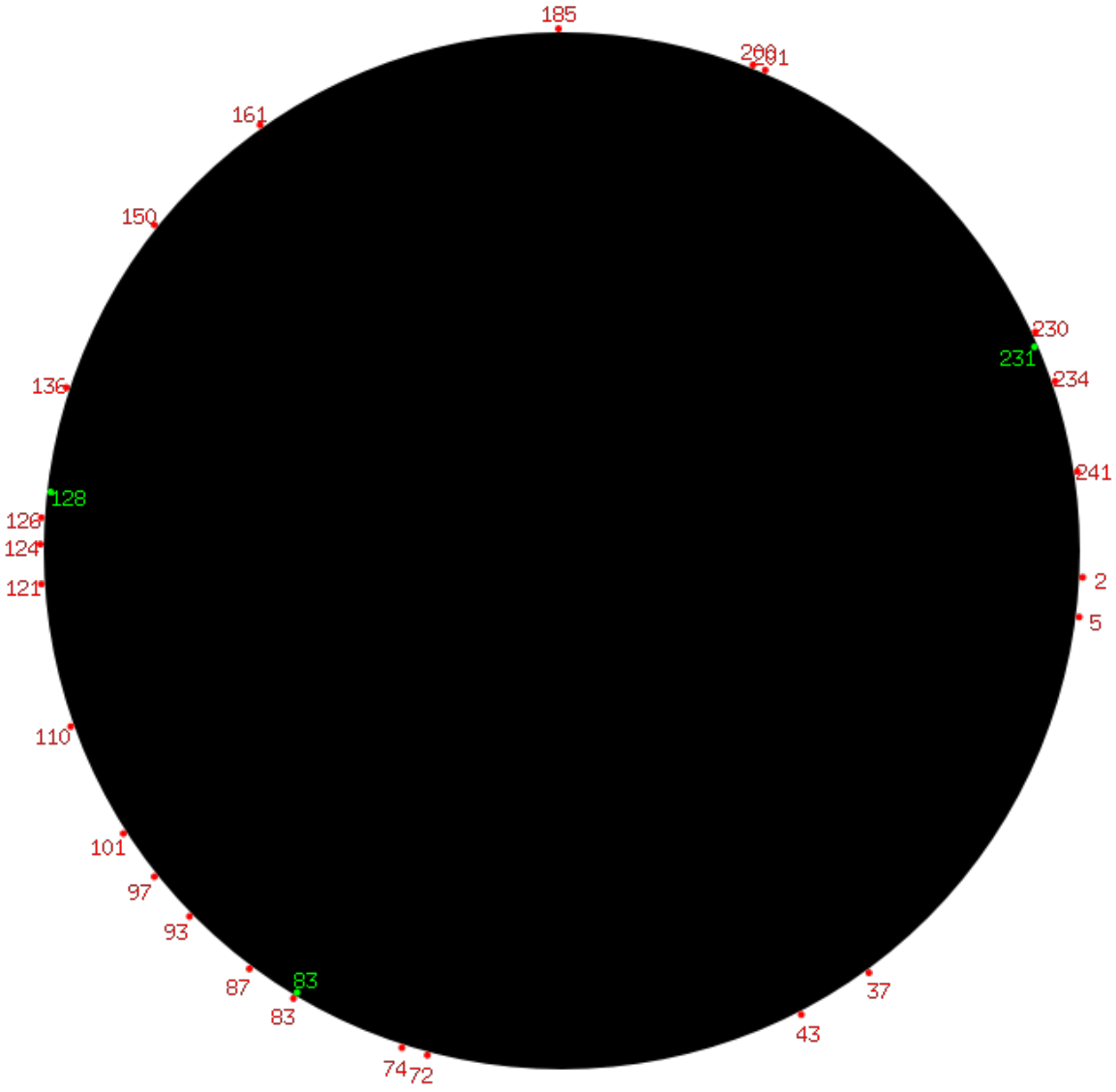

Abbildung 7: Eisbudenstandorte

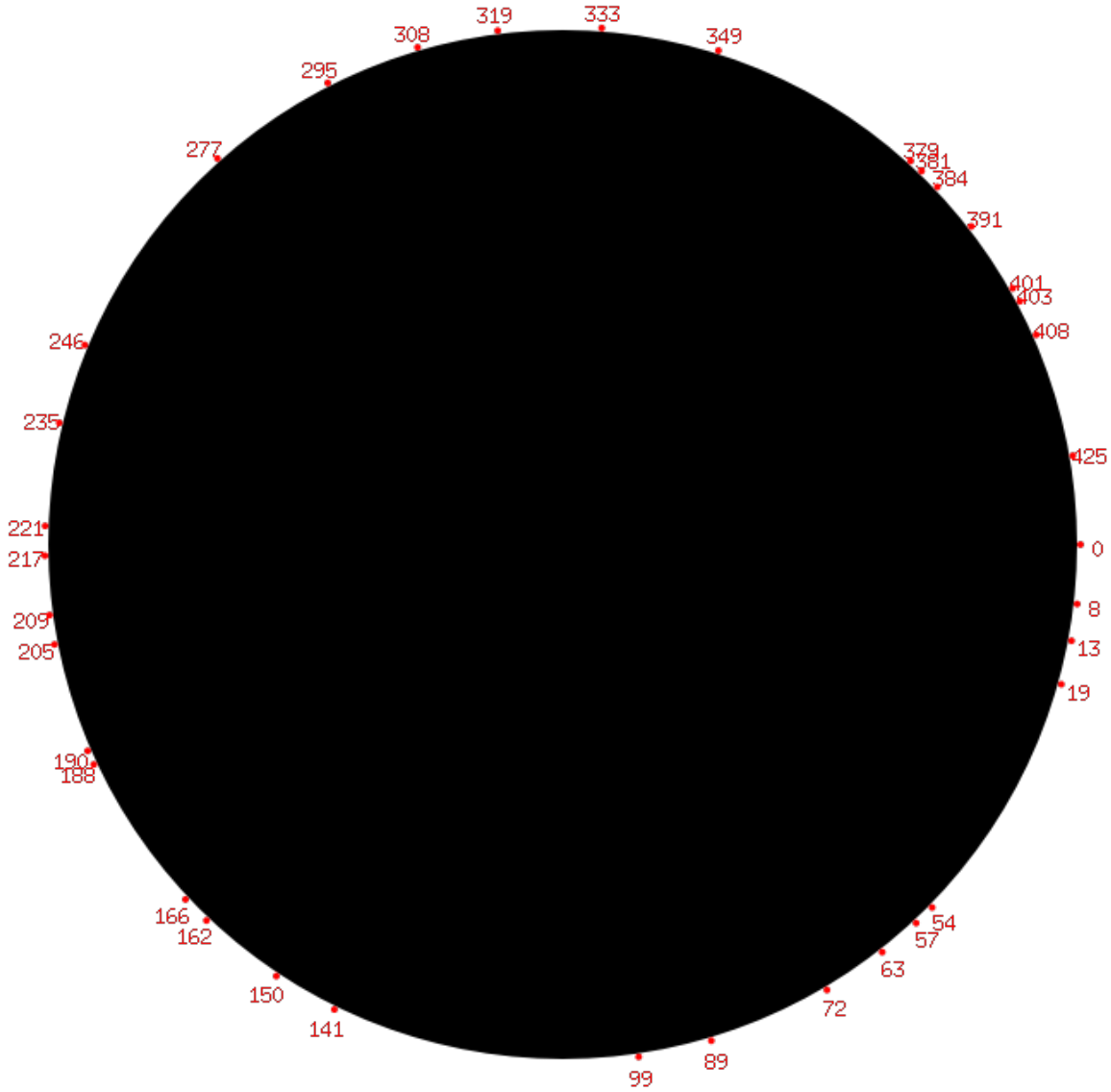

Abbildung 8: Adressen

eisbuden6.txt

```
Keine Lösung möglich

Zeit verstrichen: 0m54,536s
```

eisbuden7.txt

```
Lösung: Adressen [114 285 420] mit Summe der Abstände 1362

Zeit verstrichen: 2m52,899s
```

Die Laufzeiten reichen in meiner Umgebung von kurzen Laufzeiten bei kleinen Beispielen im Sekundenbereich bis hin zum Minutenbereich bei den größeren Beispielen.

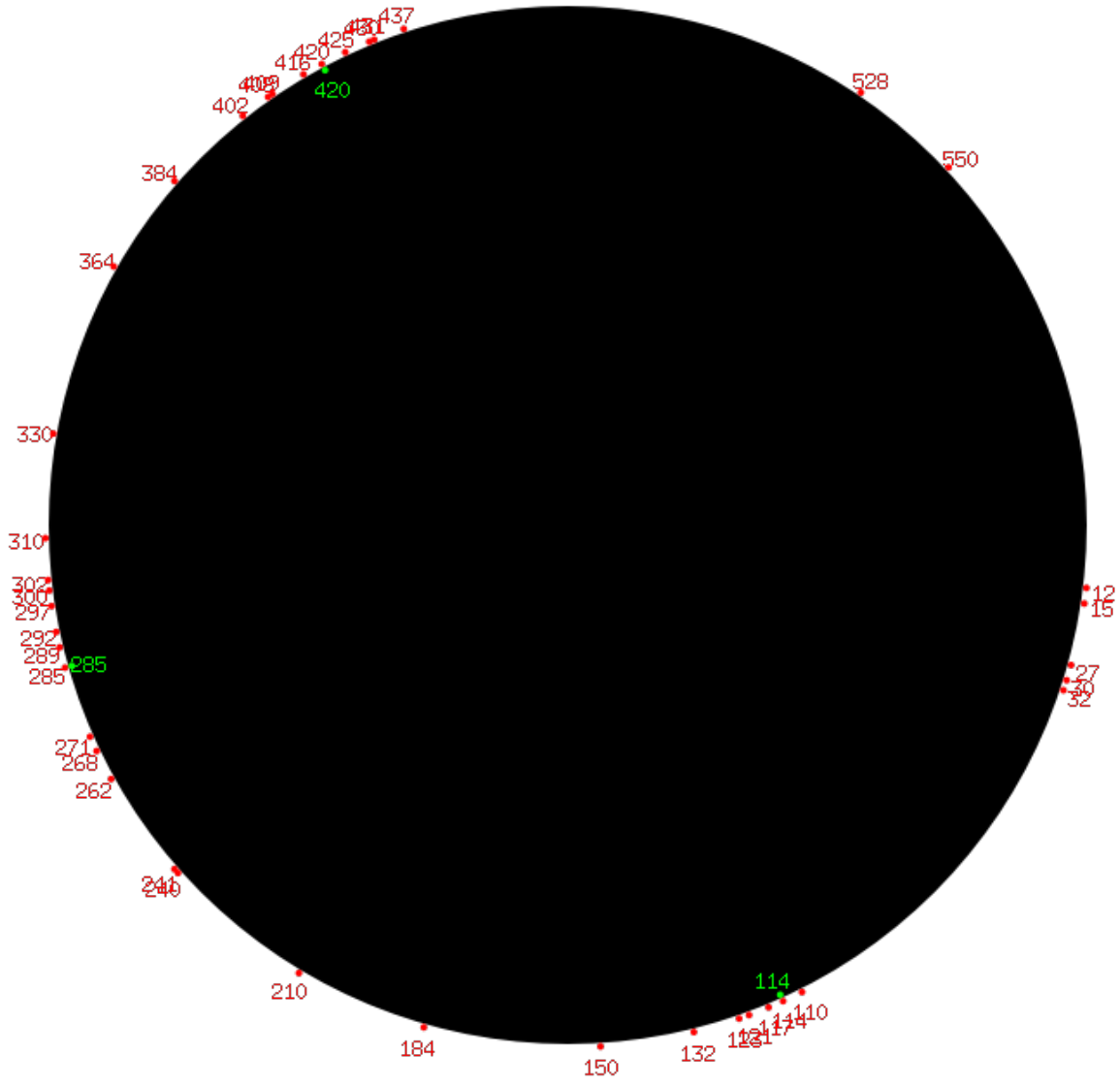

Abbildung 9: Eisbudenstandorte