Bundeswettbewerb Informatik

31. Bundeswettbewerb Informatik, 2. Runde

Lösungshinweise und Bewertungskriterien

Allgemeines

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

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

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

Dokumentation Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und die Umsetzung in Software fließen lassen. Doch ohne eine gute Beschreibung der Lösungsideen, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine gute Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (die die verschiedenen bei der Lösung

des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung wenig wert. Bewerterinnen und Bewerter können die Qualität Ihrer Einsendung nur anhand dieser Informationen vernünftig einschätzen. Mängel können nur selten durch gründliches Testen der eingesandten Programme ausgeglichen werden – wenn diese denn überhaupt ausgeführt werden können: Hier gibt es häufig Probleme, die meist vermieden werden könnten, wenn Lösungsprogramme vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner getestet würden. Insgesamt sollte die Erstellung des schriftlichen Materials die Programmierarbeit begleiten oder ihr teilweise sogar vorangehen: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt keine saubere Umsetzung in welche Programmiersprache auch immer hin.

Bewertungsbögen Auf Ihrem Bewertungsbogen sind die Bewertungskriterien als Stichpunkte in den Tabellenzeilen aufgeführt. Kein Kreuz in einer Zeile bedeutet, dass das Kriterium den Erwartungen entsprechend erfüllt wurde. Vermerkt wird also in der Regel nur, wenn davon abgewichen wurde – nach oben oder nach unten. Ein Kreuz in der Spalte „+“ bedeutet Zusatzpunkte, ein Kreuz unter „–“ bedeutet Minuspunkte für Fehlendes oder Unzulängliches. Dabei haben die Marken nicht immer den gleichen Einfluss auf die Gesamtbewertung, sondern können je nach Einsendung und Kriterium unterschiedlich gewichtig sein. Die Schattierung eines Feldes bedeutet, dass die entsprechenden Zusatz- bzw. Minuspunkte in der Regel nicht vergeben wurden. Leider ist die Schattierung bei Aufgabe 2, Theoretische Analyse / Gedanken zur Laufzeit, fehlerhaft; bei diesem Kriterium konnten auch Zusatzpunkte erreicht werden.

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

Danksagung An der Erstellung der Lösungsideen haben mitgewirkt: Meike Grewing (Aufgabe 1), Martin Thoma (Aufgabe 2) und Maximilian Janke (Aufgabe 3). Die Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt, und zwar aus Vorschlägen von Jens Gallenbacher (Aufgabe 1), Torben Hagerup (Aufgabe 2) und Arno Pasternak (Aufgabe 3).

Aufgabe 1: Komplementär

1.1 Lösungsidee

Darstellung der Puzzleteile

Meike sollte für die Puzzleteile zunächst eine praktischere Darstellung wählen, welche es ihr ermöglicht, folgende Fragen leicht zu beantworten:

- Passen zwei Puzzleteile aneinander?
- Sind zwei Puzzleteile gleich?

Dies könnte z.B. eine Darstellung als Array mit vier Einträgen sein - jeder Eintrag repräsentiert dabei eine Seite. Dazu werden die Buchstaben a, b, c, d, A, B, C, D in dieser Reihenfolge z.B. durch die Zahlen von 0 bis 7 ersetzt. Im Folgenden bezeichnet der Begriff „Seite“ eine dieser Zahlen, die ja mögliche Ein- und Ausbuchtungen einer Puzzleteil-Seite repräsentieren. Mit dieser Darstellung kann Meike leicht prüfen, ob zwei Seiten x und y aneinanderpassen: Es muss $(x+4) \mod 8 = y$ gelten.

Die Array-Darstellung ermöglicht es ihr auch, die Puzzleteile einfach zu drehen und mit einem anderen Teil zu vergleichen; zählt sie beim Zugriff auf die Array-Felder eine Zahl i zum Index hinzu und rechnet modulo 4, so hat sie eine Rotation des Puzzleteils um $i \cdot 90°$.

Erste Teilaufgabe

Da Meike für die Puzzles jeweils feststellen möchte, ob es keine, genau eine oder mehrere Lösungen gibt, bleibt es ihr nicht erspart, alle Lösungsmöglichkeiten zu überprüfen. Das sind unter Umständen jedoch recht viele. Ein Ansatz wäre hier, für jede Möglichkeit, die $m \cdot n$ Puzzleteile in ein $m \times n$-Rechteck zu legen, zu überprüfen, ob die Puzzleteile jeweils aneinander passen. Da aber jedes Puzzleteil auf bis zu 4 Weisen gedreht werden kann, wären das im 5×5-Beispiel schon bis zu $25! \cdot 4^{25}$ zu überprüfende Anordnungen: für das erste Feld gibt es 25 Möglichkeiten, für das zweite 24 usw., und dann gibt es auf allen 25 Feldern noch 4 Möglichkeiten, die Teile zu drehen. Das ist eine Zahl mit 40 Stellen! Im Allgemeinen gibt es für ein $(m \times n)$-Puzzle $(mn)!4^{mn}$ Anordnungen.

Sie sollte daher nicht alle Anordnungen durchtesten, sondern ein optimiertes Backtracking verwenden, mit dem sie möglichst schnell feststellen kann, ob ein Lösungsansatz überhaupt zum Ziel führen kann. Nur dann muss sie diesen Ansatz weiterverfolgen. Die Laufzeit für Backtracking ist leider nicht polynomiell, was dazu führt, dass das Lösen des (5×5)-Puzzles deutlich länger dauert als bei den kleineren Puzzles.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

(a) Felder zeilenweise belegen.

1	2	5	10	17
3	4	6	11	18
7	8	9	12	19
13	14	15	16	20
21	22	23	24	25

(b) Belegung entlang der Diagonalen erweitern.

Abbildung 1: Reihenfolgen bei der Belegung der Felder eines (5×5)-Puzzles.

Backtracking Man beginnt in einer Ecke, z.B. der oberen linken Ecke, und speichert für diese eine Liste aller Puzzleteile, die auf dieses Feld passen. Das sind für das erste Feld natürlich alle Teile, in jeder möglichen Rotation. Von diesen Kandidaten wählt man einen aus und erstellt für das nächste Feld (z.B. das rechts daneben) eine Liste von möglichen Puzzleteilen. Dabei geht man davon aus, dass der gewählte Kandidat in die obere linke Ecke gelegt wurde. Hier kann man natürlich nur die Teile einsetzen, die unten an den ersten Kandidaten „passen“. So fährt man reihenweise fort, bis die letzte Ecke erreicht ist.

Pfade durch das Puzzle Man muss nicht unbedingt den oben beschriebenen Pfad zum Ausfüllen des Puzzles verwenden, sofern man sicherstellt, dass jedes Feld mit einem Puzzleteil belegt wird. Je mehr Puzzleteile man frühzeitig für ein Feld ausschließen kann, desto weniger Möglichkeiten müssen ausprobiert werden. Daher versucht man die Felder des Puzzles in solch einer Reihenfolge zu belegen, dass immer möglichst wenige Teile zur Auswahl stehen. Es bietet sich also an, nicht fest vorzugeben, in welcher Reihenfolge die Felder belegt werden sollen, sondern z.B. immer das Feld als nächstes zu wählen, auf das man die wenigsten Puzzleteile legen kann. Abbildung 1 zeigt mögliche feste Reihenfolgen zur Belegung der Felder: links die zeilenweise Belegung (s.o.) und rechts eine Belegung, die entlang der Diagonalen erweitert wird. In beiden Belegungsfolgen gibt es für alle 16 Felder, die nicht am oberen oder linken Rand liegen, zwei Einschränkungen für die Auswahl des nächsten Teiles. Bei der rechten Belegungsfolge werden diese Felder aber früher belegt als bei der reihenweisen Belegung. Für nicht-quadratische Puzzles sind gute feste Reihenfolgen nicht offensichtlich; es muss deshalb dargelegt werden, wie eine Lösung mit fester Reihenfolge sich bei nicht-quadratischen Puzzles verhält.

Listen verwalten Man kann sich für jede Ein- bzw. Ausbuchtung merken, wie viele Teile mit dieser Seite zur Fertigstellung des Puzzles definitiv noch benötigt werden bzw. wie viele Exemplare dieser Seite bei den bisher nicht verwendeten Teilen noch vorhanden sind. Sobald

von einer Sorte mehr Teile benötigt werden als vorhanden sind, kann der Lösungsansatz nicht mehr zum Ziel führen. Außerdem helfen diese Listen bei der Auswahl des Feldes, das als nächstes belegt werden sollte.

Insgesamt sollte man darauf achten, schon vor Beginn des eigentlichen Lösungsalgorithmus zu berechnen, welche Teile an welche Seiten angebaut werden können. Wenn man diese Information abspeichert und dann bei der Suche der Kandidaten für ein Feld nur noch nachschlägt, welche Teile überhaupt in Frage kommen, ist der Lösungsalgorithmus deutlich schneller, als wenn man jedes Mal wieder für jedes Puzzleteil überprüfen muss, ob es an eine bestimmte Stelle passt.

Zweite Teilaufgabe

Zur Erstellung zufälliger Puzzles kann man m und n zufällig wählen, z.B. aus dem Bereich von 2 bis 5, oder man lässt den Benutzer die gewünschten Werte eingeben. Diese sollten allerdings nicht zu groß sein, damit der Berechnungsaufwand im Rahmen bleibt.

Bei der Puzzle-Generierung sollte man im Auge behalten, dass kein Teil doppelt vorkommen darf und das Puzzle eindeutig lösbar sein muss.

Ein lösbares Puzzle zu erzeugen ist nicht schwierig. Man beginnt wieder z.B. in der linken oberen Ecke und belegt diese mit einem zufälligen Puzzleteil. Dann fährt man wieder zeilenweise fort und füllt das Puzzle direkt so, dass die Teile aneinander passen, wobei man „äußere Seiten", die also bisher an kein Puzzleteil angrenzen, zufällig wählen kann. Dabei sollte man nur sicherstellen, dass man die äußeren Seiten so wählt, dass kein Puzzleteil doppelt vorhanden ist. Das geht immer, da auf der beschriebenen „Route" durch das Puzzle nie mehr als zwei Seiten vorgegeben sind. Man kann also noch mindestens zwei Seiten frei wählen und hat damit $8^2 = 64$ Teile zur Auswahl – mehr, als für die Puzzle benötigt werden.

Leider kann man aber nur schlecht voraussagen, ob ein Puzzle nicht mehr als diese eine Lösung besitzt. Man könnte nun zunächst ein zufälliges lösbares Puzzle erstellen und dann mit dem Algorithmus aus Teilaufgabe eins überprüfen, ob es tatsächlich nur eine Lösung gibt. Ist dies nicht der Fall, erstellt man so lange neue zufällige Puzzles, bis man ein eindeutig lösbares gefunden hat.

Dabei stößt man auf das Problem, dass der Algorithmus aus Teilaufgabe eins vermutlich für kleine Puzzlegrößen sehr schnell alle Lösungen findet, zum Lösen eines 5×5-Puzzles jedoch einige Sekunden benötigt. Außerdem ist bei größeren Puzzles die Gefahr groß, ein Puzzle zu erzeugen, das mehr als eine Lösung besitzt. Man muss also voraussichtlich sehr viele große Puzzle auf eindeutige Lösbarkeit überprüfen, bis man wirklich eins findet.

Um dies zu umgehen, kann man wie folgt vorgehen: Man erstellt zunächst ein eindeutiges kleineres Puzzle und erweitert dieses dann zu einem größeren. Dabei ist die Wahrscheinlichkeit, ein eindeutig lösbares Puzzle zu finden, deutlich größer. Allerdings gibt es auch größere eindeutig lösbare Puzzles, die kein eindeutig lösbares kleineres enthalten, sowie eindeutig lösbare kleinere Puzzles, die sich nicht zu einem eindeutig lösbaren größeren Puzzle erweitern lassen. Diese Einschränkungen sollte man bei dieser "schrittweisen Vergrößerung" eindeutiger Puzzles beachten.

Prüfen, ob ein Teil schon vorhanden ist Da nur wenige Puzzleteile erstellt werden, ist es nicht zwingend nötig, effizient prüfen zu können, ob zwei Teile gleich sind. Man könnte also jedes der $m \cdot n$ Puzzleteile mit jedem anderen Teil vergleichen, was zu einer Laufzeit von $O\left((m \cdot n)^2\right)$ führt. In dieser Aufgabe ist das vernachlässigbar.

Hat man es mit einer großen Menge von Puzzleteilen zu tun, sollte man sich jedoch einer besseren Methode bedienen: Man könnte z.B. jedem Puzzleteil eine „Prüfsumme“ zuweisen, hier eine Zahl aus dem Bereich $[4, \dots, 4^8]$, die folgendermaßen berechnet wird: Setze `sum` auf 0. Für jede Seite des Puzzleteils addiere 4^i zu `sum` hinzu, wobei $i \in \{0, \dots, 7\}$ der Seite entspricht ($a = 0, b = 1, \dots$). `sum` ist also genau für solche Teile gleich, die die selben vier Ein- bzw. Ausbuchtungen haben. Wenn nun also ein neues Teil an das Puzzle angefügt wird, berechnet man seine Prüfsumme und schlägt (z.B. in einer Hash-Tabelle) nach, ob schon Puzzleteile mit der selben Summe vorhanden sind. Dies können höchstens 6 Stück sein, denn mehr verschiedene Puzzleteile mit den selben Ein- und Ausbuchtungen kann es nicht geben. Jetzt muss man nur noch das neue Teil mit diesen bis zu 6 anderen Puzzleteilen vergleichen und erhält so eine Laufzeit von $O(m \cdot n)$.

1.2 Beispiele

Wir geben Lösungen für die vorgegebenen Beispiele an. Links ist jeweils die Eingabedatei zu sehen, wobei die Zeilen mit den Teilen mit 0 beginnend nummeriert sind; die -1 neben der ersten Eingabezeile hat keine Bedeutung. Rechts daneben steht die textuelle Darstellung der (bis auf Drehung eindeutigen) Lösung: zuerst eine Darstellung der Teile mit ihren Seiten, daneben eine mit ihren Nummern. Außerdem ist eine grafische Darstellung der Lösung zu sehen. Achtung: grafische Darstellungen können in den Einsendungen anders aussehen, da eine Abbildung von den Buchstabencodes der Seiten auf die vier Formen nicht vorgegeben ist.

Beispiel 1: (2×2)-Puzzle

Eingabe: Lösung (textuell): Lösung (grafisch):

```
-1   2 2          c     D
 0   acBd       a   B b   d     0     2
 1   aDac         d     a
 2   abDd         D     A
 3   abAA       a   a A   a     1     3
                  c     b
```

Beispiel 2: (3×3)**-Puzzle**

Eingabe:

```
-1   3 3
0    acBd
1    aDac
2    abDd
3    abAA
4    adBA
5    Cbcc
6    abCD
7    dBbC
8    bdDa
```

Lösung (textuell):

```
  D     a     d
a   a A   d D   a    1     4     2
  c     B     b
  C     b     B
c   b B   C c   d    5     7     0
  c     d     a
  C     D     A
b   D d   a A   a    6     8     3
  a     b     b
```

Lösung (grafisch):

Beispiel 3: (4×4)-Puzzle

Eingabe:

-1	`4 4`
0	`acBd`
1	`aDac`
2	`abDd`
3	`abAA`
4	`adBA`
5	`Cbcc`
6	`abCD`
7	`dBbC`
8	`bdDa`
9	`bDcc`
10	`bAdA`
11	`bBbC`
12	`cADd`
13	`cdBA`
14	`dCCb`
15	`ACDC`

Lösung (textuell):

```
  c     d     c     A
a   B b   C c   b B   a    0     14    9     4
  d     C     D     d
  D     c     d     D
d   a A   d D   c C   C    8     13    12    15
  b     B     A     A
  B     b     a     a
b   b B   C c   D d   b    11    7     1     2
  C     d     a     D
  c     D     A     d
b   c C   a A   a A   A    5     6     3     10
  C     b     b     b
```

Lösung (grafisch):

Beispiel 4: (5×5)**-Puzzle**

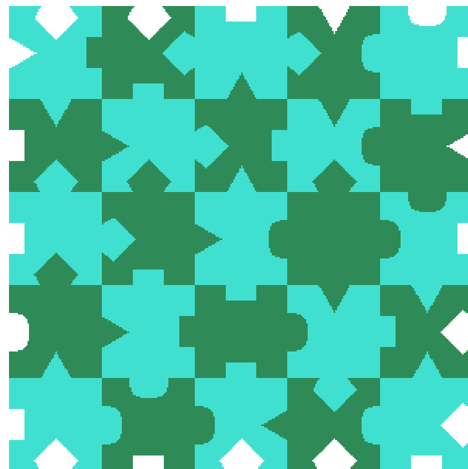
Eingabe:

```
-1   5 5
 0   acBd
 1   aDac
 2   abDd
 3   abAA
 4   adBA
 5   Cbcc
 6   abCD
 7   dBbC
 8   bdDa
 9   bDcc
10   bAdA
11   bBbC
12   cADd
13   cdBA
14   dCCb
15   ACDC
16   CcbB
17   dadb
18   Baab
19   abDC
20   adcD
21   aBdc
22   Dbda
23   baAD
24   dDcB
```

Lösung (textuell):

```
   a     a     b     d     c
d   b B   a A   A a   c C   b     2     18    10    20    16
   D     b     d     D     B
   d     B     D     d     b
b   D d   A a   b B   c C   d     8     4     22    21    14
   a     a     d     a     C
   A     A     D     A     c
b   A a   D d   c C   C c   b     3     23    24    15    9
   a     b     B     D     D
   A     B     b     d     d
c   D d   b B   C c   B b   a     12    7     11    13    17
   d     C     b     A     d
   D     c     B     a     D
b   C c   C c   d D   c C   a     19    5     0     1     6
   a     b     a     a     b
```

Lösung (grafisch):



1.3 Erweiterungen

Diese Aufgabe ist in sich recht abgeschlossen, aber kleinere Erweiterungen sind doch möglich. Die Anzahl der Seitentypen, die hier vier (mal zwei) beträgt, kann erhöht werden; dies bedeutet aber keine besondere Steigerung der Schwierigkeit. Etwas interessanter ist die Verwendung anderer Formen wie Dreiecken oder Sechsecken für die Puzzleteile, was neue Überlegungen bzgl. fester Belegungsreihenfolgen erfordert. Spielraum für Kreativität ist ebenfalls vorhanden: Die zu belegenden Felder könnten unregelmäßig angeordnet werden (dazu müsste ein entsprechender Plan verwaltet werden), die Teile könnten gefärbt und dann zu Mustern gelegt werden, etc. Ein grundlegend anderer Lösungsansatz als Backtracking ergibt sich aber auch daraus nicht.

1.4 Bewertungskriterien

- Im Vergleich zur Lösungsstrategie spielt die Repräsentation der Puzzleteile eine weniger wichtige Rolle. Gute Ideen zu diesem Punkt sind deshalb nicht nötig, können aber belohnt werden; für besonders ungeeignete oder umständliche Darstellungen kann es Abzüge geben.
- Die Lösungen eines Puzzles müssen korrekt berechnet sein (mit passenden Seiten).
- Das Programm sollte die gegebenen Puzzles lösen und dafür zumindest bis 4×4 nicht (viel) mehr als eine Sekunde benötigen. Reine Brute-Force-Ansätze genügen nicht, besser als mit intelligentem Backtracking geht es wohl nicht. Einfaches Backtracking ohne weitere Verbesserung ist aber zu einfallslos. Eine optimierte Belegungsreihenfolge wurde häufig verwendet, immerhin ist ein entsprechender Vorschlag[1] leicht im Netz zu finden; mit einer optimierten Belegungsreihenfolge ist das (5×5)-Puzzle in etwa einer Minute lösbar. Interessanter als feste Reihenfolgen sind dynamische Reihenfolgen, also die Berechnung des als nächstes zu belegenden Feldes nach jedem Zug. Beachtenswert sind auch Maßnahmen zur geschickten Verwaltung von Puzzleteilen (z.B. Listen passender Puzzleteile) oder Ähnliches.
- Die Lösung sollte sich nicht auf quadratische Puzzlegrößen beschränken, wozu das Material eventuell verleitet.
- Im zweiten Teil sollten nur eindeutig lösbare Puzzles erzeugt werden; falls sich das nicht klar aus dem gewählten Verfahren ergibt, sollte begründet sein, warum die erzeugten Puzzles eindeutig sind. Eventuell getroffene Beschränkungen bei der Erzeugung von Puzzles sollten benannt und ebenfalls begründet werden.
- Bei dieser Aufgabe spielt die Laufzeit eine wichtige Rolle. Eine (nicht notwendiger Weise formale) Analyse der Laufzeit, etwa durch kombinatorische Überlegungen, wurde deshalb erwartet. Auch konkrete Laufzeiten sollten genannt sein; nicht unbedingt für jedes Beispiel, aber z.B. durchschnittliche Zeiten für die verschiedenen Puzzlegrößen.
- Für die vorgegebenen Beispiel-Puzzles müssen Lösungen angegeben sein. Außerdem werden eigene Beispiele erwartet, zumindest sollte ein nicht-quadratisches Puzzle mit Lösung angegeben sein.
- Lösungen sollten nachvollziehbar dargestellt sein. Eine grafische Darstellung ist nicht nötig, textuelle Darstellungen sollten aber die geometrische Anordnung der Teile bzw. der Ein- und Ausbuchtungen andeuten.

[1] http://www.abenteuer-informatik.de/PDF/Affenpuzzle_GeroScholz.pdf

Aufgabe 2: Marsch auf Mars

Der Bau einer Forschungsstation auf dem Mars mit 100 identischen autonomen Robotern ist eine große Sache. Dass diese nicht mit einander kommunizieren können und keine gemeinsame Karte – und daher auch kein gemeinsames Koordinatensystem – haben, macht die Sache nicht leichter. Immerhin können sie andere ihrer Art um Umkreis von 200m wahrnehmen. Da das Raumshuttle gewährleistet, dass nach dem Abwurf des ersten Roboters jeder weitere sich in einem Umkreis von 200m von einem früher abgeworfenen befindet, ist dies auch theoretisch machbar. Damit die Raumstation das gewährleisten kann, beginnen die Roboter erst nach dem Abwurf des letzten Roboters mit ihrer Programmausführung. Sie haben also einem Empfänger, der auf das Signal des Raumshuttles wartet und haben eine gemeinsame Zeit. Dies können die Roboter natürlich in ihrem Algorithmus nutzen.

2.1 Lösungsidee

Abwurf der Roboter

Die Modellierung des Abwurfes der Roboter kann durch eine Umrechnung ins Polarkoordinatensystem erfolgen. Der erste Roboter wird o.B.d.A. im Ursprung abgeworfen. Dann wird für jeden weiteren Roboter ein zufälliger Radius $r \in [0,200]$ und ein zufälliger Winkel $\alpha \in [0,360)$ gewählt. Der Abwurfort des $(i+1)$-ten Roboters ist dann, für ein $j \leq i$,

$$P_{i+1} = (x_{i+1} \mid y_{i+1}) = (x_j + r \cdot \cos(\alpha) \mid y_j + r \cdot \sin(\alpha))$$

Eine einfache Abwurfvariante, die auch einfache Lösungsstrategien ermöglicht, ist der „Kettenabwurf“, nämlich der Abwurf des Roboters R_{i+1} in die Nähe des unmittelbar vorher abgeworfenen Roboters R_i. Dadurch lassen sich aber nicht alle im Sinne der Aufgabenstellung gültige Endkonstellationen der Roboter erreichen, das Problem wird damit also eingeschränkt.

Treffpunktfindung über den Schwerpunkt

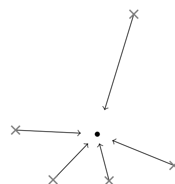


Abbildung 2: Treffen im Schwerpunkt

Jeder Roboter kann alle anderen Roboter wahrnehmen („sehen“), die sich in einem Umkreis von 200m um ihn herum befinden. Nun kann jeder Roboter sich auf den Punkt zubewegen, der den minimalen quadratischen Abstand zu jedem der wahrgenommenen Roboter hat. Dieser Punkt ist der Schwerpunkt. Man erhält ihn durch die Berechnung des Durchschnitts der x-

und y-Koordinaten.[2] Da durch die Bewegung der Roboter eventuell neue Roboter in den sichtbaren Bereich kommen, sollte die Schwerpunktberechnung regelmäßig durchgeführt werden. Dadurch ist es auch möglich, dass sich die Roboter nicht nur in eine Richtung bewegen. Abbildung 2 zeigt fünf Roboter, die sich alle sehen können und so direkt auf den Schwerpunkt zu fahren.

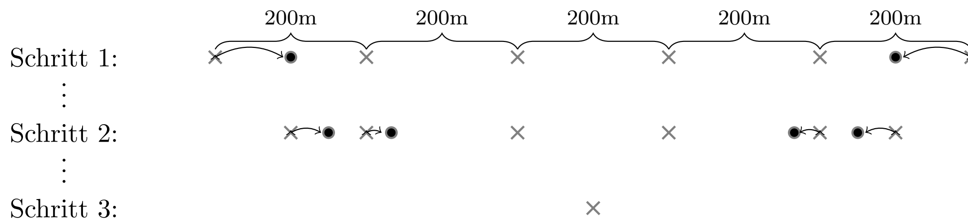


Abbildung 3: Alle Roboter auf einer Geraden

Das Verfahren funktioniert auch für stärker verteilte Roboterabwürfe wie z.B. 100 Roboter auf einer Geraden bei einem Abstand von 200m zwischen den Robotern. Diese Situation und das Verhalten des reinen Schwerpunkt-Algorithmus ist in Abbildung 3 dargestellt. Dabei sollte noch erwähnt werden, dass die Roboter sich in dieser Situation nur deshalb auf einem Punkt treffen, weil die Positionsangaben durch die `double`-Ungenauigkeit „gerundet“ werden.

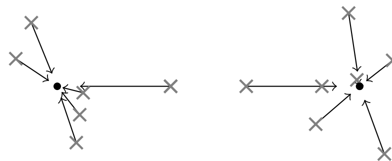


Abbildung 4: Bildung zweier Cluster

Dieser Algorithmus löst das Problem allerdings nicht. In Abbildung 4 ist eine Situation dargestellt, in der sich zwei Cluster bilden. Es können sich prinzipiell sogar beliebig viele Cluster bilden, wobei die Anzahl der maximal möglichen Cluster abhängig von der Roboteranzahl ist.

Ein sehr schwerer Fall ist die regelmäßige Anordnung der 100 Roboter in einem 10×10-Raster. Die Bewegung der Roboter mit dem Schwerpunkt-Algorithmus ist in Abbildung 5 dargestellt. Das letzte Bild zeigt auch die Sichtbereiche der Roboter, nachdem sich 13 Cluster gebildet haben.

Eine einfache und dennoch vielfach hilfreiche Ergänzung des Schwerpunktverfahrens ist es, den Roboter stehen bleiben zu lassen, falls er durch seine Bewegung einen anderen Roboter aus dem Blickfeld verliert.

[2] Bei der Durchschnittsberechnung spielt es keine Rolle, dass jeder Roboter ein eigenes Koordinatensystem hat. Egal wie das kartesische Koordinatensystem für jeden Roboter gewählt wird, der Schwerpunkt ist immer am gleichen Ort, auch wenn er eventuell mit anderen Koordinaten bezeichnet wird.

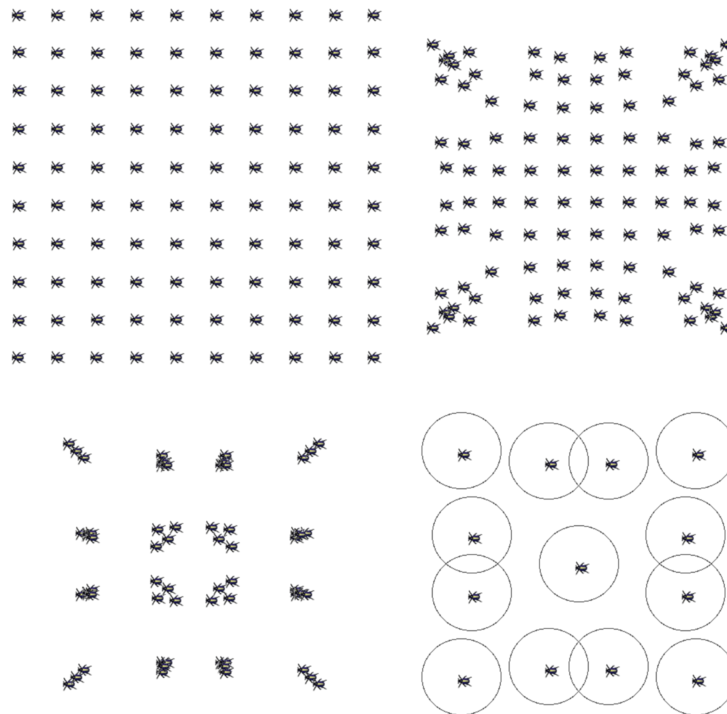


Abbildung 5: Bildung mehrerer Cluster

Clustervereinigung

Bei dem reinen Schwerpunkt-Algorithmus können sich mehrere Cluster von Robotern bilden. Es ist jedoch gewünscht, dass sich alle Roboter in einem Punkt treffen. Also müssen die Cluster zusammengeführt werden. Dazu muss man sich zuerst klar machen, dass nach $100 \cdot 200\text{m} \cdot 1\frac{\text{s}}{\text{m}} = 20000\text{s}$ entweder alle Roboter an einem Treffpunkt sind oder alle Roboter in Clustern sind. Wenn alle Roboter an einem Punkt sind, weiß das jeder Roboter, da das Raumshuttle auf jeden Fall 100 Roboter abwirft und den Robotern diese Zahl bekannt ist. Dann ist die Aufgabe gelöst.

Sollte es jedoch Cluster geben, geht jeder Roboter an seine Startposition zurück. Nach 20000s kann garantiert werden, dass jeder Roboter wieder dort angelangt ist. Nun geht jeder Roboter direkt auf das Zentrum des Clusters zu, in dem er zuletzt war. Roboter, die in dem gleichen Cluster waren werden sich auf diese Weise nicht aus dem Blick verlieren. Sind jedoch zwei Roboter zu Beginn in Sichtweite und verlieren dann den Kontakt, so wissen sie, dass sie in verschiedenen Clustern sind. Diese Roboter bewegen sich zurück zu der Position, wo sie zuletzt Kontakt hatten und werden für weitere Bewegungen gesperrt. Die Sperre wird erst aufgehoben, wenn andere Roboter sich auf der selben Position befinden.

Diese Methode vereinigt nun mindestens zwei Cluster, da die Sperre dafür sorgt, dass die Verbindung zwischen zwei Clustern weiter existiert. Der Cluster wird sich auf die Position des gesperrten Roboters zu bewegen. Sobald er ihn erreicht hat, wird die Sperre aufgehoben und gleichzeitig entweder der zweite Cluster oder ein weiteres Verbindungsglied sichtbar.

Da sich nicht unbedingt alle Cluster nun vereinigt haben, muss die Clustervereinigung so häufig wie nötig angewandt werden. Dies ist höchstens 98 mal der Fall. Damit finden sich die Roboter spätestens nach $20000\text{s} + 98 \cdot (2 \cdot 20000\text{s}) < 46$ Tage.

2.2 Weitere Lösungsideen

Entferntester Nachbar

Neben der Treffpunktfindung über den Schwerpunkt ist es auch vorstellbar, dass jeder Roboter zu dem sichtbaren Roboter geht, der am weitesten entfernt ist. Diese Strategie löst zufällige Verteilungen besonders gut. Es existieren jedoch ein paar Roboterverteilungen, bei denen diese Treffpunktfindungsstrategie eine Clustervereinigungsstrategie benötigt. Einer dieser Problemfälle ist in Abbildung 6 dargestellt. So können bei regelmäßigen Verteilungen viele Roboter mehrere entfernteste Nachbarn haben und diese ungünstig wählen, was zu mehreren Clustern führt.

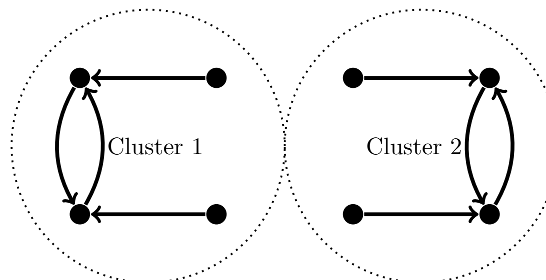


Abbildung 6: Mögliche Clusterbildung bei der Strategie „Entferntester Nachbar“

Bienentanz

Die Roboter können nicht direkt miteinander kommunizieren, aber die Bewegung anderer Roboter beobachten. Somit können sie ein „Bienentanz-Protokoll“ implementieren. Das Schwierige an dieser Lösung ist vor allem die Tatsache, dass die Roboter kein gemeinsames Koordinatensystem und insbesondere keine gleich ausgerichteten Achsen nutzen dürfen.

Diskretisierung

Jeder Roboter könnte sich die Koordinaten aller unbewegten Roboter in einer Liste L merken. Roboter bleiben nur auf Koordinaten aus L stehen und suchen unter diesen Punkten denjenigen aus, der den minimalen quadratischen Abstand hat. Dies ist sozusagen der diskrete Schwerpunkt. Falls Roboter in einem Cluster sind, bewegen sie sich zu einer zufälligen Position in L und bleiben kurz dort. Sobald ein Roboter alle 100 Positionen kennt, kennt er den diskreten Schwerpunkt. Nun kann es allerdings immer noch sein, dass nicht alle Roboter diese Position kennen. Dieses Problem muss für eine gültige Lösung auch behandelt werden.

Start direkt nach Landung

Die Aufgabenstellung schließt nicht aus, dass jeder Roboter direkt nach der Landung beginnt. Damit kann es grundsätzlich eine sehr einfache Strategie geben: Der zuerst gelandete Roboter R_1 erkennt sich als solchen, da er bei der Landung keinen anderen Roboter sieht, und bleibt stehen. Der zweite Roboter R_2 bewegt sich auf den ersten zu. Wenn ein weiterer Roboter abgeworfen wird, sieht er eine Menge von Robotern. Der abgeworfene Roboter wählt sich zufällig einen Roboter aus der Menge aus. Falls er auf einen anderen Roboter abgeworfen wurde, läuft der neu abgeworfene dem Roboter hinterher, auf den er abgeworfen wurde. Er bewegt sich zu diesem Roboter. Indirekt bewegt sich jeder Roboter auf R_1 zu.

Diese Strategie funktioniert aber nur unter bestimmten Randbedingungen, die klar beschrieben und begründet sein müssen: Zum einen müssen die Roboter eben unmittelbar nach der Landung mit ihren Operationen beginnen. Zum anderen dürfen die Zeitabstände zwischen zwei Abwürfen nicht zu klein sein, denn sonst könnte der zweite Roboter noch während des Ortungsprozesses des ersten landen und dessen „Selbsterkennung“ verhindern; das kann insbesondere passieren, wenn die Roboter getaktet operieren. Generell geht diese Strategie davon aus, dass ein Roboter einen anderen dauerhaft identifizieren kann; das ist aber nicht immer möglich, z. B. dann, wenn mit punktförmigen Robotern gerechnet wird, von denen mehrere den gleichen Standort einnehmen können.

2.3 Erweiterungen

Für die Lösung wurde angenommen, dass alle Roboter sich auf einer Ebene ohne Hindernisse befinden. Das ist genau genommen eine unrealistische Idealisierung. Nahe liegende Erweiterungen sind deshalb der Einbau von Hindernissen, die Berücksichtigung der Oberflächenkrümmung (wichtig für besonders kleine Planeten) oder die Annahme einer komplett anderen Topologie – wer weiß, was im Universum alles so anzutreffen ist.

2.4 Bewertungskriterien

- Die Lösung muss den zufälligen Abwurf korrekt implementieren. Es muss garantiert sein, dass ein Roboter nach seiner Landung mindestens einen anderen Roboter orten kann; ein „Kettenabwurf“ ist aber nicht in Ordnung. Abzüge gibt es auch, wenn der Bereich, in dem die Roboter abgeworfen werden, künstlich eingeschränkt wird. Außerdem muss klar sein, wie die Roboter sich nach dem Abwurf verhalten: Gemeinsamer Start nach Signal oder Start direkt nach der Landung.
- Das Verfahren zum Finden eines Treffpunkts darf nicht auf einem gemeinsamen Koordinatensystem bzw. gemeinsamen Nordpol beruhen, außer dieses/r wurde zur Laufzeit von den Robotern bestimmt. Auch eine zentrale Steuerung ist nicht erlaubt.
- Die Fähigkeiten der Roboter müssen der Aufgabenstellung entsprechen: Ortung nur relativ zur eigenen Position und Ausrichtung, keine Kommunikationsfähigkeit, Maximalgeschwindigkeit von 1 m/s.

- Unter der Bedingung, dass jeder Roboter beim Start seiner Aktivitäten mindestens einen anderen Roboter orten kann, sollte immer ein Treffpunkt gefunden werden, und zwar auch in den folgenden Extremfällen: Die Roboter befinden sich (a) auf einer Geraden im Abstand von 200m, (b) auf den Eckpunkten eines regelmäßigen 100-Ecks mit der Seitenlänge 200m oder (c) auf einem 10×10-Raster. Diese Extremfälle (oder zumindest einige davon) sollten erkannt worden sein. Falls sie nicht besonders behandelt werden (z.B. durch Clusterverbindung), sollte dies zumindest begründet sein, etwa durch Argumentation mit der geringen Wahrscheinlichkeit solcher Situationen.
- Die Probleme oder Randbedingungen einfacher Strategien – Clusterbildung bei „entferntester Nachbar“, unmittelbarer Aktionsbeginn nach der Landung und ausreichend großer Zeitabstand zwischen Abwürfen bei „folge dem ersten Roboter“ – müssen erkannt und auch behandelt werden.
- Da die Roboter in der Realität eine Ausdehnung haben, ist die Bedeutung von „Treffpunkt“ oder „gemeinsamer Ort“ unklar. Diese Problematik muss angesprochen und ihre Behandlung begründet werden. Es ist in Ordnung, sich das Leben zu vereinfachen und die Roboter als punktförmig anzusehen; dann können sie sich wirklich in einem Punkt treffen.
- Es sollte angesprochen werden, wie viel Zeit die Roboter benötigen, um einen Treffpunkt zu finden. Hier ist mit „Laufzeit“ also keine eingabeabhängige algorithmische Komplexität, sondern die Zeit bis zur Terminierung des Verfahrens gemeint. Besonders gute Überlegungen werden belohnt.
- Die Simulation der Roboterbewegung soll durch „kommentierte Augenblicksbilder“ dokumentiert werden. In der Regel sollte für mindestens drei verschiedene Ausgangssituationen das Finden eines Treffpunktes mit genügend Bildern von Zwischensituationen dokumentiert sein.

Aufgabe 3: Tourality

3.1 Das Spiel

Im Prinzip sind die Regeln von Tourality einfach und schnell erklärt, doch bald sieht man, dass eine Lösungsstrategie dies nicht ist. Selbst wenn man alleine auf dem Feld steht, ist es ein schweres Problem, alle Chips in möglichst geringer Zeit einzusammeln.[3] Außerdem gibt es noch den Gegenspieler, der einem das Leben nicht leichter macht, indem er im Weg steht und einem die Chips vor der Nase wegschnappt.

Tourality ist ein Nullsummenspiel mit perfekter Information. Dadurch, dass das Spiel aus zwei Runden mit jeweils vertauschten Startpositionen und -spielern besteht, ist es neutral, das heißt: kein Spieler kann von vornherein eine Gewinnstrategie[4] haben, zumindest aber ein Unentschieden garantieren. Da jeder Spieler im Turniersystem maximal 800 Züge (400 pro Runde) macht, kann man theoretisch stets einen optimalen Zug berechnen (siehe Minimax-Algorithmus). Da wir jedoch bezüglich der Rechenzeit beschränkt sind, werden wir die optimale Gewinnstrategie nicht berechnen können. Stattdessen müssen wir uns bemühen, eine möglichst gute Strategie in dem uns gegebenen Zeitrahmen zu finden.

Laut Aufgabenstellung besteht das Spiel aus zwei Runden, wobei wir insgesamt bemüht sind möglichst viele Punkte mehr als unser Gegenspieler zu haben. Dabei kann man die beiden Runden getrennt voneinander betrachten und in beiden Runden unabhängig versuchen, die Differenz zwischen unseren Punkten und denen des Gegners zu maximieren.

Spielplan Die Spielumgebung liefert uns eine Liste von Hindernissen und Chips (bzw. Blättern). Für die meisten Algorithmen ist eine solche Repräsentation des Spielplans jedoch nicht sinnvoll. Stattdessen empfiehlt es sich, den Zustand (leer, Hindernis, Chip, eigene KI, gegnerische KI) eines jeden der 20×20 Felder zu speichern. Für spätere Zwecke ist es nützlich, die Chips durchzunummerieren. Indem man das Spielfeld mit Hindernissen einrahmt, erspart man sich später die Betrachtung von „Randfällen“ bei der Implementierung.

3.2 Lösungsideen

Nearest-Neighbour-Heuristik

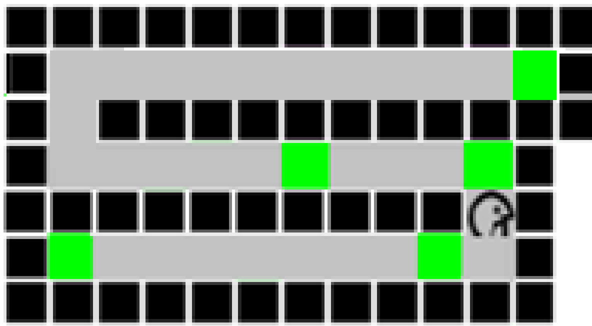
Als erster Ansatz bietet sich die Nearest-Neighbour-Heuristik an. Dazu wählt die KI ihre Bewegungsrichtung „gierig“ derart, dass sie den Abstand zum nächstgelegenen Chip minimiert. Effektiv steuert die KI somit einen der nächstgelegenen Chips an. Diese Richtung lässt sich effizient mit einer Breitensuche bestimmen[5].

---

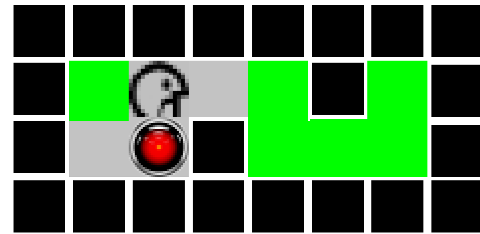
[3] Es entspricht einer NP-schweren Variante des Problems des Handlungsreisenden, wenn man die Beschränkungen für die Feldgröße und die Anzahl der Chips als variabel erachtet.

[4] Das ist eine Strategie, mit der er garantiert mehr als 40 Chips (insgesamt) bekommt.

[5] Auf einem $n \times n$-Feld läuft die Breitensuche in $O(n^2)$; das lässt sich in unserem Fall ($n = 20$) mit einer Laufzeit von unter 1 ms realisieren.



(a) Optimal lassen sich die Chips in 42 Schritten einsammeln, die Heuristik benötigt jedoch 56.



(b) Die „rote KI“ (der Kreis unten) kann einem guten Gegner nicht mehr als einen Chip abringen; verwendet die andere KI jedoch die Nearest-Neighbour-Heuristik, vermag Rot alle bis auf einen Chip einzusammeln.

Abbildung 7: Ungünstiges Verhalten der Nearest-Neighbour-Heuristik.

Selbst im Einzelspiel garantiert die Nearest-Neighbour-Heuristik aber kein gutes Ergebnis. So lassen sich Spielpläne finden, auf denen der Weg, den die Heuristik wählt, um einen frei wählbaren Faktor länger als der kürzestmögliche Weg ist.[6]

Im Spiel zu zweit neigt die Nearest-Neighbour-Heuristik im ungünstigen Fall dazu, der anderen KI hinterher zu laufen. Entsprechend lassen sich Spielpläne mit n Chips konstruieren, in denen die Heuristik $n-2$ Chips weniger einsammelt als eine optimale KI[7] (vgl. Abbildung 7).

Weitere Heuristiken

Die Nearest-Neighbour-Heuristik lässt viele Varianten und Erweiterungen zu. So kann man sich beispielsweise überlegen, ob man bei der Berechnung des Weges zum nächsten Chip den Gegner als Hindernis betrachtet oder nicht. Gerade wenn man weit vom Gegner entfernt ist, kann es sinnvoll sein, das Feld, auf dem dieser steht, als frei zu betrachten, da zu erwarten ist, dass er nicht mehrere Züge auf dem gleichen Feld verweilt[8]. Diese Variante bietet zumindest einen Laufzeitvorteil, dann man muss nicht mehr jede Runde eine Breitensuche durchführen, sondern nur noch eine für jeden Chip, der aufgehoben wird, also 40 im gesamten Spiel.

Natürlich sind weitere Heuristiken denkbar. Bei der Wahl des Chips, den man ansteuert, muss man beispielsweise nicht immer den nächstgelegenen wählen, sondern kann andere Faktoren berücksichtigen wie die Nähe zum Gegner (und ob dieser den Chip eventuell vor einem ergattern kann) oder die Anzahl der Chips in der Umgebung. Man kann die KI vorausschauender gestalten, zum Beispiel indem sie ihren Weg so wählt, dass sie möglichst schnell zwei, drei,

[6] Tatsächlich liegt die Tour, die die Heuristik wählt, im schlechtesten Fall in $\Omega(l^2)$, wobei l die Länge einer optimalen Tour ist.

[7] Angenommen die gegnerische KI spielt jeweils optimal.

[8] Natürlich stellt diese Argumentation eine Vereinfachung dar, der Gegner könnte auf einem Feld verweilen, um einen zu blockieren oder aufgrund eines Fehlers, vielleicht ist ihm die Zeit ausgegangen. Im letzten Fall sollte man ihn einfach als Hindernis betrachten.

bzw. k Chips einsammelt. Mit steigendem k steigt jedoch auch die benötigte Berechnungszeit stark an.

In der Wahl der Heuristik gibt es viele Möglichkeiten, die verschieden gute Resultate liefern. Es kann sich lohnen, Zeit in eine gute Heuristik zu investieren, denn auf deren Basis lassen sich meist auch gute (heuristische) Bewertungsfunktionen konstruieren, wie wir sie später benötigen.

Minimax-Algorithmus

Rufen wir uns noch einmal unser Ziel ins Gedächtnis: In jeder Spielrunde versuchen wir die Differenz zwischen unseren Punkten und denen des Gegners zu maximieren. Für eine gegebene Spielsituation in der wir am Zug sind, wollen wir dabei die maximale Differenz, die wir gegen einen optimal spielenden Gegner erzwingen können, als den Wert der Spielsituation bezeichnen. Offensichtlich ist der beste Zug, den wir in einer gegebenen Situation machen können, derjenige, der diesen Wert maximiert. Gleichermaßen wird unser Gegner (wenn er gut spielt) den Zug wählen, der den Wert (aus unserer Sicht) minimiert.

In einer Spielsituation S stehen einem Spieler bis zu fünf Züge zur Verfügung: er kann in eine der vier Richtungen gehen oder stehen bleiben. Die Menge dieser Züge wollen wir mit $\mathbf{Z}(S)$ bezeichnen. Da uns immer die Option bleibt, stehen zu bleiben, ist $\mathbf{Z}(S)$ nicht leer.

Sei $z \in \mathbf{Z}(S)$, dann setze man

$$c_S(z) := \begin{cases} 0 & \text{mit dem Zug } z \text{ wird kein Blatt eingesammelt} \\ 1 & \text{mit dem Zug } z \text{ wird ein Blatt eingesammelt} \end{cases}$$

$S[z]$ sei die Spielsituation, die sich ergibt, wenn der Zug z in Spielsituation S ausgeführt wird, dabei betrachten wir das Spielende als eigene Spielsituation. Mit dieser Notation können wir eine rekursive Berechnungsvorschrift für den Spielwert angeben[9]:

$$\begin{aligned} \text{wert(Spielende)} &= 0 \\ \text{wert}(S) &= \max_{z \in \mathbf{Z}(S)} c_S(z) - \text{wert}(S[z]) \end{aligned}$$

$\text{wert}(S)$ gibt dabei den Wert an, den der Spieler, der in Spielsituation S anfängt, dem Spiel beimisst. Für den anderen Spieler hat das Spiel dann den Wert $-\text{wert}(S)$.

Damit ist im Prinzip auch bereits der Minimax-Algorithmus gegeben: Eine KI spielt genau dann optimal, wenn sie in einer Spielsituation S stets den Spielzug $z \in \mathbf{Z}(S)$ so wählt, dass $c_S(z) - \text{wert}(S[z])$ maximal ist.

In der Praxis ist die soeben gegebene Rechenvorschrift für wert() jedoch zu langsam. Im schlechtesten Fall hat $\mathbf{Z}(S)$ fünf Elemente, und es müssen für beide Spieler jeweils 400 Züge berechnet werden. Dann würde wert() sich ca. 5^{800}-mal selbst aufrufen, was fernab dessen liegt, was sich in der Praxis berechnen lässt.

[9]Hier wird davon ausgegangen, dass die Anzahl der Spielzüge wie auf dem Turnierserver auf 400 pro KI und Runde beschränkt ist.

Das Problem bei dieser naiven Methode besteht unter anderem darin, dass im obigen Fall viele Spielsituationen mehrfach betrachtet werden. Man kann die Laufzeit also entsprechend verbessern, indem man sich die für eine Spielsituation berechneten Werte merkt und sie dann nachschlagen kann, wenn die gleiche Situation wieder eintritt. Dazu bedarf es eines assoziativen Arrays[10] d, in welches sich Spielsituationen einfügen lassen:

```
function WERT(S)
    if der Wert d[S] existiert nicht then
        d[S] ← −∞
        for all z ∈ Z(S) do
            d[S] ← max(d[S], c_S(z) − WERT(S[z]))
        end for
    end if
    return d[S]
end function
```

Dieser Pseudocode verdeutlicht freilich nur das allgemeine Verfahren zur Berechnung von wert(); in der Praxis sollte wert(S) iterativ berechnet werden, da die Entwicklungsumgebung die Rekursionstiefe beschränkt[11].

Die durch Memoisierung erreichte Verbesserung reicht zudem noch nicht, um wert(S) in den vorgegebenen Zeitschranken zu berechnen, wie eine Abschätzung über die Anzahl der Spielsituationen zeigt. Jede Spielsituation ist nämlich eindeutig durch folgende Spielinformationen festgelegt:

- Position der Spielers, der gerade am Zug ist
- Position des anderen Spielers
- Anzahl der verbleibenden Züge
- Menge der vorhandenen Chips

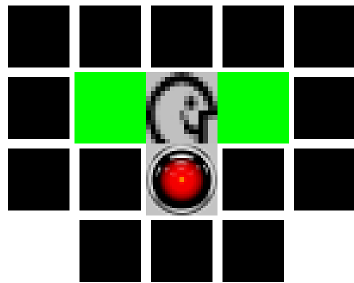
Dabei kann der Spieler, welcher gerade am Zug ist, maximal 360 Positionen einnehmen, für den zweiten Spieler verbleiben 359. Es gibt 800 mögliche Werte für die Anzahl der verbleibenden Züge. Um die Menge der vorhandenen Chips zu charakterisieren, muss man sich für jeden der n vorhandenen Chips[12] merken, ob er bereits eingesammelt wurde. Insgesamt kann man so abschätzen, dass es ca. $10^8 2^n$ mögliche Spielsituationen gibt. Indem man den Wert für die einzelnen Spielsituationen memoisiert, erreicht man also einen Vorteil gegenüber dem naiven Ansatz wert(S) zu berechnen, jedoch ist das noch nicht genug. Man ist an einem Punkt angelangt, wo man auf die Optimalität der KI zugunsten der Laufzeit verzichten muss.

Im Folgenden wollen wir versuchen, wert(S) anzunähern. Dazu behalten wir im Grunde die Rechenvorschrift bei, sparen jedoch einiges aus, indem insgesamt weniger Spielsituationen betrachtet werden. Dazu müssen wir sowohl den Faktor 10^8 als auch den Faktor 2^n geeignet verkleinern.

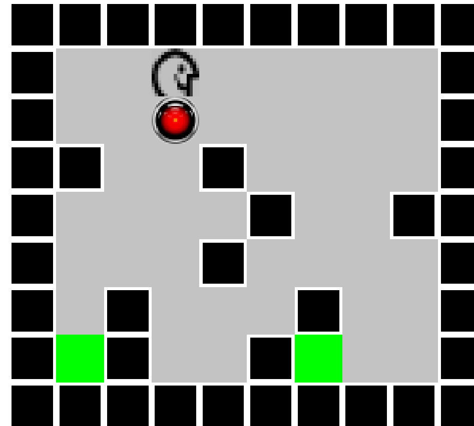
[10] Assoziative Arrays (auch Dictionarys, Maps genannt) erweitern Arrays auf im Allgemeinen nichtnumerische Indexmengen. Da sie sowohl von Java und Python unterstützt werden, betrachten wir sie einfach als zur Verfügung stehend und gehen nicht auf ihre Umsetzung ein.

[11] Zumindest, wenn für die Implementierung Java verwendet wurde, ist eine Rekursionstiefe von über 12.000 nicht möglich.

[12] Bei Spielbeginn gilt $n = 40$.



(a) Hier sollte sich die obere KI erst gegen Spielende bewegen, damit Rot nicht mehr genug Züge verbleiben, um den anderen Chip einzusammeln.



(b) Ein komplizierteres Beispiel, in dem die rote KI den Sieg nur aufgrund der Begrenzung der Zugzahl erzwingen kann.

Abbildung 8: Einfluss einer Begrenzung der Zugzahl auf das Spielresultat.

Um den Faktor 10^8 klein zu bekommen, kann man bei der Betrachtung der Spielsituationen die Anzahl der verbleibenden Züge ignorieren. Wir gehen also davon aus, dass uns beliebig viele Züge zur Verfügung stehen. Damit reduziert sich die Anzahl der möglichen Spielsituationen auf weniger als $130.000 \cdot 2^n$. Es sei jedoch betont, dass diese Laufzeitersparnis auf Kosten der Optimalität geht. So lassen sich Spielsituationen konstruieren, in denen der Spielwert anders ist, wenn die Anzahl der verfügbaren Züge begrenzt ist (vgl. Abbildungen 8(a) und 8(b)). In der Praxis scheint das jedoch nicht allzu ausschlaggebend zu sein, zumindest legt das die Beobachtung von Spielen verschiedener KIs nahe.

Mit der soeben vorgeschlagenen Verbesserung ist aber auch der Algorithmus anzupassen, denn nun kann es vorkommen, dass sich WERT() rekursiv auf die gleiche Spielsituation aufruft (z.B. wenn für beide Spieler die Option „Stehenbleiben“ betrachtet wird). In diesem Fall können wir für die Spielsituation den Wert 0 annehmen: wenn es für beide Spieler adäquat ist, dass sich eine Spielsituation wiederholt, dann kann offensichtlich keiner mehr Chips als der andere ergattern.

Damit wird aber der Spielwert in den Unteraufrufen von WERT(S) nicht mehr notwendigerweise korrekt berechnet, was sich jedoch nicht auf die Korrektheit des Rückgabewertes des initialen Aufrufs auswirkt. Wenn wir die fehlerhaften Werte später fortlaufend weiterverwenden, garantiert das sogar ein optimales Spiel.[13]

Falls n klein ist, kann wert(S) nun bereits berechnet werden. Bei großen n ist das jedoch noch immer nicht möglich. Statt einer häufig bei Minimax-Algorithmen üblichen Beschränkung der Suchtiefe bietet es sich hier an, die betrachteten Mengen von Chips zu beschränken. Beispielsweise kann man sich auf Spielsituationen beschränken, in denen nur eine bestimmte Anzahl an Chips eingesammelt wurde.

[13] Das gilt jedoch mit den nun folgenden Änderungen nicht mehr.

Hier bedarf es nun einer heuristischen Bewertungsfunktion, also einer Funktion, mit welcher wir den Wert einer Spielsituation abschätzen können, anstatt ihn weiter mit wert() zu berechnen. Eine naive Bewertungsfunktion würde für jede Spielsituation den Wert 0 schätzen. Da die Qualität des Minimax-Algorithmus jedoch essentiell von der Qualität der Bewertungsfunktion abhängt, sollte man sich an dieser Stelle etwas Mühe geben. Besser wäre beispielsweise, das Ergebnis, welches zwei einfach zu berechnende KIs (beispielsweise mit Nearest-Neighbour-Heuristik) in einer gegebenen Spielsituation erbringen, als Schätzwert zu nehmen.

Der Minimax-Ansatz lässt sich durch Pruning-Techniken wesentlich verbessern. Beispielsweise vermag man mit Alpha-Beta-Pruning die Laufzeit zu steigern, es bewähren sich aber durchaus auch spekulativere Techniken. Ein solcher Ansatz, welcher die Leistung der KI in der Praxis stark steigert, sei nun vorgestellt.

Ein Chip-basierter Ansatz

Die bis jetzt erarbeitete KI stellt eine erhebliche Verbesserung gegenüber einfachen Heuristiken dar. Vor allem im Endspiel und in Konfliktsituationen, in denen die Spieler nahe beieinander sind, schlägt sie sich gut. Sind jedoch viele Chips vorhanden und die Spieler weit voneinander entfernt, bewährt sie sich nur mäßig und ist nicht vorausschauend genug, um wirklich gute Entscheidungen zu treffen.

Wir betrachten also im Folgenden den Fall, in dem die Spieler „weit weg“ voneinander sind und noch viele Chips zur Verfügung stehen: Es stellt sich heraus, dass es in diesem Fall meist das Beste für beide Spieler ist, auf direktem Wege Chips einzusammeln. Man kann also vereinfachend davon ausgehen, dass sich ein Spieler für einen Chip entscheidet, den er dann auf einem kürzesten Weg anfährt. Erst wenn er den Chip erreicht hat, trifft er die nächste Entscheidung. Da die Gegner weit voneinander entfernt sind, ist die Wahl des kürzesten Weges (es gibt gewöhnlich mehrere) nicht ausschlaggebend, es sei denn, es liegen weitere Chips auf einem kürzesten Weg. Man kann jedoch davon ausgehen, dass der Spieler nur Ziele auswählt, bei denen letzteres nicht der Fall ist; liegt auf dem Weg zum Chip c der Chip b, gibt es keinen Grund den Chip c als Ziel zu wählen, bevor man Chip b (oder evtl. einen anderen Chip) angefahren hat.

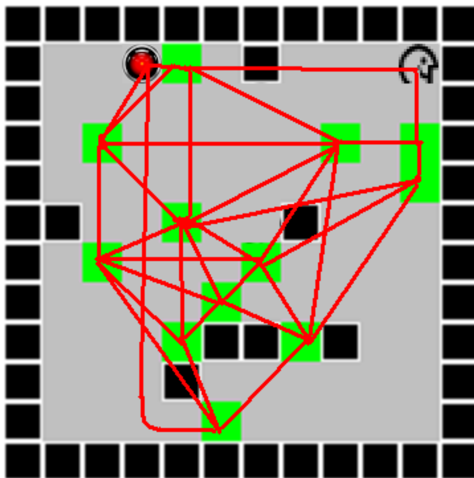
Es lässt sich dementsprechend die Menge G_K der zu betrachtenden Ziele des Spielers K folgendermaßen charakterisieren: Sei C die Menge aller Chips, man definiere für $a, b \in C$:

$$a \prec_K b :\Leftrightarrow a \text{ liegt auf einem kürzesten Weg von der Position von } K \text{ nach } b$$

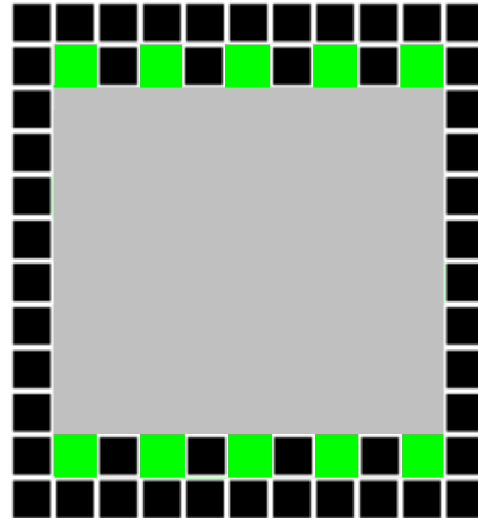
$\prec_K$ ist eine partielle Ordnung auf C. G_K ist nun genau die Menge aller Chips, welche bezüglich $\prec_K$ minimal sind[14].

Man erhält teilweise eine erhebliche Beschleunigung des Minimax-Algorithmus, wenn man sich für jeden Spieler K auf Züge beschränkt, welche auf einem (a priori fest wählbaren) Weg zu einem Chip aus G_K liegen. Wie oben bereits erwähnt, ist diese Beschränkung aber nur dann gerechtfertigt, wenn die Spieler „weit weg“ voneinander sind. Es gilt also ein Kriterium für „weit weg“ zu finden. Hier bieten sich viele Möglichkeiten. Ein sinnvolles Kriterium ist, dass

[14] Ein Chip c ist minimal bezüglich $\prec_K$, wenn für jeden Chip b mit $b \prec_K c$ bereits $b = c$ gilt.



(a) Eine zufällige Karte, auf der für jeden Chip und die beiden Spieler die bezüglich der obigen Ordnung $\prec_K$ minimalen Chips eingezeichnet wurden.



(b) Eine Karte, auf der $|G_K|$ unabhängig von der Spielsituation für beide Spieler maximal ist, $\prec_K$ ist leer. Der Pruningansatz funktioniert hier also nicht.

Abbildung 9: Spielsituationen beim Chip-basierten Ansatz.

G_{Spieler1} und G_{Spieler2} disjunkt sind. Dabei bewährt es sich, diese Bedingung nicht für jeden Zug zu prüfen, sondern erst dann, wenn ein Spieler einen Chip einsammelt.

Das obige Verfahren hat sich auf zufällig generierten Spielplänen als sinnvoll erwiesen. Hier besteht G_K meist aus wenigen Elementen[15] und die Abstände zu diesen Chips sind nicht allzu groß, weswegen tatsächlich wenige Spielsituationen berücksichtigt werden müssen. Es gibt jedoch Spielpläne, in denen die oben definierte Relation $\prec_K$ leer ist und folglich jeder Chip in G_K liegt (unabhängig von der Position des Spielers K).

Abschließende Bemerkung

Da es eine Fülle von Methoden zur Realisierung von Spiele-KIs gibt, sind die hier vorgestellten Verfahren vor allem als Beispiele und nicht als einzig richtige Lösungen zu sehen. Tatsächlich bietet sich eine breite Auswahl von Möglichkeiten; neben der großen Variabilität, die sich bereits in der Wahl von Heuristiken und Pruningansätzen zeigt, sind auch grundlegend andere Herangehensweisen vorstellbar. Beispielsweise lässt sich über Monte-Carlo-Verfahren oder andere randomisierte Algorithmen nachdenken. Man kann auch versuchen die Beschränkung der Spielzüge zu nutzen um beispielsweise den Spieler im Endspiel zu blockieren, wie sich das in den in Abbildung 8 beschriebenen Spielsituationen anbietet. Oftmals lässt sich die Qualität verschiedener Ansätze erst im praktischen Spiel entscheiden.

[15] Bei fast allen zufälligen Spielplänen lag $|G_K|$ für alle Positionen von K unter 15, für die meisten Positionen bestand G_K aus 5 bis 10 Elementen.

3.3 Erweiterungen

Die Beschränkungen für den Spielplan lassen sich verallgemeinern, beispielsweise kann man größere Spielpläne mit mehr Chips und Hindernissen betrachten. Statt 20×20 quadratischen Feldern lassen sich dem Spielplan auch allgemeinere Graphen zugrundelegen. Hierbei sollte die Erweiterung jedoch einen neuartigen Ansatz für die KI erfordern und nicht nur die ohnehin schon entwickelten Methoden verallgemeinern.

Man kann auch einen nichtdiskreten Spielplan betrachten, indem sich der Spieler einen Schritt in eine beliebige Richtung bewegen kann (also nicht nur in vier Raumrichtungen). Die Hindernisse könnten in dem Fall allgemeine Polygone sein, und auch die Blätter und Spieler könnten eine feste Form und Größe haben.

Die Anzahl der Spieler lässt sich variieren. Beispielsweise können Teams, die jeweils aus mehreren Spielern bestehen, gegeneinander antreten. Dabei steuert entweder eine KI alle Spieler eines Teams (sozusagen zentral), oder jeder Spieler hat eine autonome KI, welche nicht mit ihren Mitspielern kommunizieren kann.

3.4 Bewertungskriterien

- Die verwendeten Verfahren sollten über einfache heuristische Ansätze hinausgehen. Auf jeden Fall sollte die KI in irgendeiner Weise auf ihren Gegner Bezug nehmen, dessen Position berücksichtigen und auf seine Züge reagieren.
- Es muss auf die zur Verfügung stehende Berechnungszeit geachtet werden. Insbesondere sollte vermieden werden, dass eine KI aus Zeitmangel stehenbleibt, was meist fatal für den Spielverlauf ist. Dazu müssen die grundlegenden Verfahren, etwa zur Berechnung der Abstände zwischen Spieler und Chips bzw. zwischen den Chips, ausreichend effizient sein. Überlegungen zur Laufzeit solcher Verfahren und der gesamten Strategie sind also erforderlich.
- Die Qualität der verwendeten Verfahren muss gut begründet und an ausreichend vielen Beispielen diskutiert werden. Dazu ist eine visuelle Darstellung von geeigneten Spielsituationen erforderlich.
- Die KI sollte sich in der Praxis (d.h. im Abschlussturnier) gut schlagen. Auf jeden Fall sollte sie einer einfachen, heuristischen KI, beispielsweise mit Nearest-Neighbour-Heuristik, überlegen sein.

Perlen der Informatik – aus den Einsendungen

Allgemeines

Wort des Wettbewerbs: Anti-Bit

Deshalb habe ich mich entschlossen, den kompletten Delphi-Code, den ich für den Thread-Algorithmus entworfen habe, auf Assembler umzustellen. *Er hat es wirklich getan!*

Das Schwierige an dieser Erweiterung ist zu begründen, warum sie schwierig ist.

Der ausgelassene Quelltext ist nicht etwa auf einer fröhlichen Feier, sondern im Anhang zu finden.

Aufgabe 1: Komplementär

Allerdings ist diese Methode, wie beinahe alle Brute-Force-Methoden, sehr ineffizient.

Um einen Algorithmus für dieses Puzzle zu finden, muss man zunächst die Komplexitätsklasse des Puzzles finden.

Um das erste und auch das zweite Problem zu löschen, ... *Ja ja, BwInf-Probleme sind nun mal besonders brenzlig.*

Aufgabe 2: Marsch auf Mars

Dieses Konstrukt ist zwar fragil, aber dank seiner begrenzten Komplexität nicht zusammengebrochen.

Die Funktion `destroy` bringt den Roboter zum Stehen.

Falls dieses Dokument am Rechner gelesen wird, ist eventuell noch zu beachten, dass es sich nicht bei allen Punkten um Roboter, sondern manchmal auch einfach nur um Dreckflecken auf dem Schirm handelt.

Danach kehrt der Roboter zum Startpunkt zurück und fährt mit dem Bit fort (so wie in „fortfahren“, nicht wie in „fort fahren“).

Das Programm stellt Roboter als Instanzen der Klasse `Toaster` dar.

Wer weiß, was für Technik im Roboter verbaut ist, wenn das Geld nicht einmal für eine Funkverbindung gereicht hat.

Wer zur Hölle gibt [...] Milliarden an Euro aus, um Roboter [...] auf den Mars zu schießen, [...] hat aber dann kein Geld oder keine Lust, die Roboter mit einem simplen WLAN oder einem anderen Kommunikationssystem auszustatten?

Aufgabe 3: Tourality

Dieses Verfahren habe ich durch mehrere Verfahren optimiert.

… aus einer soziopathischen Einstellung gegenüber dem Turnierserver mache mir übrigens keine Gedanken zur Speicherplatzkomplexität.

Implementiert habe ich den Algorithmus in der Klasse `ASTAR` (bitte nicht mit der Blumensorte „Aster“ verwechseln, die schreibt man mit „e“).

Da ich ein Gegner der Vorratsdatenspeicherung bin, berechne ich die Funktionsparameter jede Runde neu.

Der Käfer hat anschließend Bauchschmerzen, die er aber gerne erträgt, um sie dem anderen Käfer zu ersparen.