Manganernte auf dem pazifischen Meeresgrund

Jan-Christoph Klie
jck@mrklie.com

Stephan Alaniz Kupsch
stephan.alaniz@gmail.com

DHBW Mannheim

15. Januar 2014

Inhaltsverzeichnis

1 Problembeschreibung und -analyse

1.1 Problembeschreibung

Aufgabe war es, Roboter zu simulieren, die allein auf sich gestellt auf dem pazifischen Meeresboden Mangan sammeln und sich nach einer gewissen Zeit an einem gemeinsamen Punkt treffen.

Im weiteren wird ein kurzer Überblick über mögliche Lösungen gegeben, welche warum implementiert wurde, und wie die Theorie dahinter aussieht. Schließlich wird noch diskutiert, wie gut der gewählte Ansatz schließlich war und wie es mit mehr Zeit (noch) besser gemacht werden kann.

1.2 Annahmen über das Problem

Es wurden Annahmen über die Eigenschaften des roblem gemacht, wo es keine Beschreibung/Limitierung gab. Diese sind im Folgenden:

- Roboter sind punktförmig
- Roboter können in der Bewegung saugen
- Es gibt keine Beschränkung, wie viele Roboter sich in einem Punkt befinden
- Kommunikation ist instantan und ohne Berechnungszeit (Senden wie Empfangen)
- Art der Kommunikation zwischen Robotern ist unbeschränkt (es kann alles übertragen werden)
- Roboter haben unbeschränkt viel Speicher

1.3 Disclaimer

Die Einschränkungen, die mit dem Problem kommen, wurden nicht simuliert. Nur deren Auswirkungen wurde berücksichtigt.

2 Vorgehensmodell und Entwicklung

Die Entwicklung hat sich in zwei große Phasen geteilt. Beide hatten ihr eigenens Vorgehensmodell, welche im Folgenden beschrieben werden:

2.1 Ideenfindung

Die erste Phase hat damit begonnen, mögliche Kanditaten für eine Lösung zu finden. Die Früchte dessen sind in Kapitel 3 zu finden. Wichtig war, sich nicht auf einen Ansatz zu versteifen und sofort zu implementieren, sonder Für und Wider abzuwägen. Im Laufe dessen sind immer neue Ideen gekommen, viele sind verworfen worden. Schließlich hat man sich für einen Ansatz entschieden, der möglichst schnell ein Ergebnis liefern sollte und eine Grundlage für alle späteren Entwicklungen bildet. Als sich dann herauskristallisiert hat, welcher Ansatz am geeignetsten war, wurde schließlich Phase Zwei eingeläutet, die Entwicklung an sich.

2.2 Design-Ziele

Bevor jedoch implementiert wurde, wurden einige Richtlinien und Ziele definiert, wie genau entwickelt werden soll. Diese werden im folgenden beschrieben:

Keep it simple
: Die Lösung sollte so einfach wie möglich gehalten sein. So wenig Abstraktion wie möglich, so viel wie nötig. Nichts implementieren, was nicht benötigt wird (z.B. Interfaces für womöglich benötigte Komponenten)

Rapid Prototyping
: Lieber schnell einen Prototypen schreiben und gucken, ob es überhaupt funktionieren kann. Im Laufe dieser Arbeit sind so vier oder fünf lauffähige Lösungen entstanden, und nur die letzte wurde so bereinigt und optimiert. Ziel war, Zeit zu sparen, was auch erreicht wurde.

Getrennt marschieren - vereint schlagen
: Es wurde versucht, Komponenten möglichst unabhängig zu entwickeln. Da viele der verwendeten Algorithmen ohne weiteres Stand-Alone getestet und implementiert werden konnten, wurde erst ganz am Schluss integriert. So konnte ohne Probleme Schnittstellen und Datenstruktueren vollkommen geändert werden, ohne Änderungen an anderer Stelle nach sich zu ziehen.

Not reinventing the square wheel
: Fast alles an Standardbibliotheken wurde bereits entwickelt, und zwar besser, als man es selbst könnte. Daher wurde lieber und ohne Zögern auf externe Module gesetzt.

Optimize last
: Da viel Code geschrieben wird, der am Ende nicht mehr benutzt wird, wurde erst in der letzten Iteration auf Performance geachtet.

90 % thinking, 10 % coding
: Die Zeit, die mit Programmierung an sich verbraucht wurde, hat nur einen geringen Anteil an der Gesamtzeit. Daher wurde viel mehr Wert auf gute Ideen zum Anfang hin gelegt.

Wenn unklar, abstrahiere
: Viele Detailentscheidungen wurden erst spät getroffen oder laufend geändert. Diese wurden so implementiert, dass sie leicht austauschbar sind. Beispiel dafür ist die Modellierung des Meeresbodens. Im letzten Schritt wurden diese Abstraktionen dann entfernt.

2.3 Nicht-Ziele

Es wurde keinen Wert darauf gelegt, eine ästethisch schöne Benutzeroberfläche zu entwickeln. Ziel war nur, dass sie Benutzerfreundlich ist.

2.4 Entscheidungen

Einige wichtige Entscheidungen wurden schon in der Beschreibung der Implementierung aufgezählt. Im Folgenden werden nun noch

Python
: Python als Programmiersprache hat viele Vorteile. Zum einen ist es extrem viel einfacher, Protoypen zu schreiben, da man im Allgemeinen viel weniger Code für Funktionalität braucht wie für vergleichbare andere Sprachen. Außerdem kommt Python "batteries included", d.h. es gibt sehr viele externe Pakete, die sich einfach installieren lassen. Schließlich kann Python als Duct Tape für nativen Code eingesetzt werden, was sich im Laufe des Projektes als äußerst praktisch erwiesen hat.

Matplotlib
: Matplotlib ist ein Paket, welches für das Plotten von Daten zuständig ist. Außerdem kann es diese auch animieren. Es ist kein vergleichbares Paket in anderen Programmiersprachen bekannt, was so vielseitig visualisieren kann, und somit ein weiterer Grund für Python als Grundlage

Numpy
: Numpy bietet schnelle, native Datenstrukturen für N-Dimensionale Arrays an. Es ist Grundlage für die gespeicherten Daten. Diese Arrays können von Python aus an C-und-C++-Code übergeben werden, wo dann performant Number Crunching betrieben werden kann. Außerdem ist es gut für Rechenoperationen auf Matrizen, vergleichbar da mit MATLAB.

Ipython notebook
: Zum Prototyping wurde eine Webanwendung benutzt, Ipython notebook genannt. Dort kann im Browser Python geschrieben werden und Matplotlib-Graphiken angezeigt werden. Somit kann das Arbeitsergebnis einfach geteilt werden. Beispiel dafür ist eine statische Version aus den Anfängen des Projektes: `http://goo.gl/z6B1N8`

Cython
: Cython ist ein Python-Modul, welches eine Schnittstelle zwischen Python und C/C++ bietet. So ist die Logik des Erntens in C++ geschrieben (Laufzeit ist so von 200s auf unter 10s gefallen), und wird nur von Python aufgerufen.

Git+Github
: Unverzichtbar mittlerweile ist die Versionsverwaltung von Sourcecode, vor allem, wenn mit mehreren Entwicklern zusammengearbeitet wird.

3 Problembetrachtung

Das Problem kann unter vielen verschiedenen Blickwinkeln betrachtet werden. Je nachdem, in welcher Domäne der Informatik es eingeordnet wird, gibt es unterschiedliche Lösungsansätze.

In diesem Abschnitt wird kurz beschrieben, welche Ansätze erdacht wurden, welches deren Vor- und Nachteile sind, und für welche schließlich ausgewählt wurden.

3.1 Graphenproblem

3.1.1 Graphendefinition

Aus der zugrunde liegenden Umgebung aus quadratischen Zellen (beschrieben in Kapitel 4 [ref?]) lässt sich ein Graph erstellen, bei dem jeder Knoten einer Zelle entspricht und über eine Kante zu jeder Nachbarzelle verbunden ist. Da sich ein Roboter in jede beliebige Richtung bewegen kann, handelt es sich dabei um einen bidirektionalen Graphen. Kanten zwischen zwei Knoten besitzen eine Gewichtung, je nachdem, ob ein Feld bereits besucht wurde oder nicht. Zu einem Knoten, der bereits besucht wurde, kann somit jede auf ihn gerichtete Kante die Gewichtung 0 besitzen. Ein noch nicht besuchter Knoten hat dahingegen Kanten mit einer Gewichtung von 1 die auf ihn zeigen.

3.1.2 Problembeschreibung

Es wird angenommen, dass die Position von einem Roboter und die Position des Ziels bekannt sind. Eine relative Position ist hier vollkommen ausreichend. Nun soll der Pfad gefunden werden, der in einer vorgegebenen Anzahl an Schritten (gleichzusetzen mit der vorhandenen Missionszeit) zum Ziel führt und dabei so viele unbesuchten Knoten wie möglich durchquert. Der Einfachheit halber wird für jeden Roboter sequentiell der beste Weg gesucht.

3.1.3 Lösungsansätze

Depth Limited Search Depth Limited Search (beschränkte Tiefensuche) ist ein Suchalgorithmus, bei dem Baum oder ein Graph erst in der Tiefe und anschließend in der Breite durchsucht wird. Zusätzlich besitzt der Algorithmus eine feste Tiefe. Das heißt, dass der Algorithmus alle Pfade durchsucht, die die maximale Tiefe nicht überschreitet. Von jedem Node kann so bei einem Schritt zu einem der vier Nachbarn gegangen werden. Der folgende Pseudocode beschreibt, wie Depth Limited Search funktioniert.

```
1 DLS(node, goal, depth) {
2   if ( depth >= 0 ) {
3     if ( node == goal )
4       return node
5
6     for each child in expand(node)
7       DLS(child, goal, depth-1)
8   }
9 }
```

Bezogen auf einen Roboter wird die aktuelle Position (`node`) , das Ziel (`goal`) und die Missionszeit (`depth`) dem Algorithmus übergeben. Da Depth Limited Search ursprünglich terminiert, sobald das Ziel gefunden wurde, muss der Algorithmus dahingehend modifiziert werden, sodass er alle möglichen Pfade durchsucht und sich den besten merkt.

```
DLS(node, goal, depth, path_so_far) {
  if ( depth >= 0 ) {
    if ( node == goal )
      path_so_far.append(node)
      check_for_path_score(path_so_far)
      path_so_far.pop()

    for each child in expand(node)
      path_so_far.append(node)
      DLS(child, goal, depth-1, path_so_far)
      path_so_far.pop()
  }
}
```

Die modifizierte DLS durchsucht nun alle möglichen Pfade, berechnet in `check_for_path_score()`, wie viele unbesuchten Knoten durchlaufen werden und speichert zugleich den besten Pfad, den er findet.

Der Hauptvorteil von Depth Limited Search ist, dass der Algorithmus definitiv den best möglichen Pfad zum Ziel im gegebenen Szenario findet. Der entscheidene Nachteil liegt allerdings in der Komplexität des Algorithmus. Diese ist geschätzt $\mathcal{O}(b^d)$.

Da die Rechenkomplexität exponentiell ist, wird schnell ein Schwellenwert erreicht, bei der die Berechnung aller Pfade zum Ziel schlicht zu lange dauert. Es ist möglich die Rechenkomplexität etwas zu verringen, indem nur die Pfade durchsucht werden, von denen das Ziel in der restlichen Zeit auch noch erreichbar ist. Dennoch bleibt es bei der theoretischen Grundkomplexität und verhilft gerade bei langen Pfaden (höhere Missionzeit) zu keinem entscheidenen Zeitgewinn.

Depth Limited Search wurde implementiert, allerdings auch schnell wieder verworfen, auf Grund der fehlenden Performance bei längeren Pfaden.

Bellman Ford Der Bellman Ford Algorithmus ist ein Suchalgorithmus der Graphtheorie um den kürzesten Weg zu berechnen. Eine Besonderheit des Algorithmus ist, dass er auch mit negativen Kantengewichtungen umgehen kann und erkennt, sobald es einen negativ gewichteten Zyklus gibt. Ein solcher Zyklus entsteht, indem sich in einem Pfad ein Kreis befindet, bei dem sich die Pfadkosten (Summe der Kantengewichtungen eines Pfades) immer weiter senken. Ist ein solcher Zyklus vorhanden, gibt es keinen kürzesten Pfad, da die Pfadkosten beliebig klein werden können.

Im folgenden ist der Bellman Ford Algorithmus mit Pseudo Code beschrieben. Die Idee des Algorithmus besteht darin in jeden Knoten zu speichern mit welcher Pfadlänge der Knoten erreicht werden kann und welcher Vorgänger zurückverfolgt werden muss, um dem Pfad mit der gespeicherten Länge zu erhalten. Dabei wird iterativ immer nach dem kürzesten Pfad zu jedem Knoten gesucht und die Werte aktualisiert.

Konkret wird im ersten Schritt der Ausgangsknoten mit der Pfadlänge 0 initialisiert und alle anderen Knoten bekommen eine undefinierte oder unendliche Länge zugeordnet.

Im zweiten Schritt wird über alle Kanten iteriert und überprüft, ob über eine Kante ein Knoten erreicht werden kann, sodass der Pfad, der über den Ursprung der Kante führt, kürzer ist als die bisherig gespeicherte Pfadlänge im Zielknoten. In diesem Fall wird die Pfadlänge angepasst und der Vorhänger neu gesetzt. Dies geschiet solange bis sich die Informationen in den Knoten nicht mehr ändern (oder maximal so oft wie es Knoten gibt). Schließlich wird im dritten Schritt geprüft, ob ein negativ gewichteter Zyklus vorhanden ist.

```
BellmanFord(list vertices, list edges, vertex source)
   // This implementation takes in a graph, represented as lists of vertices and edges,
```

```
    // and fills two arrays (distance and predecessor) with shortest-path information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := infinity
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
```

Bezogen auf einen Roboter, der den Pfad zum Ziel sucht, der am höchsten gewichtet ist, kann man einfach den Algorithmus dahingehend anpassen, dass er nach der längsten Pfadlänge sucht. In der Praxis ergibt such allerdings das Problem, dass Zyklen kaum zu vermeiden sind und Roboter ihre eigenen Pfade niemals kreuzen dürfen. In beiden Fällen gelangt der Algorithmus nicht zu einer Lösung. Daher wurde dieser Algorithmus zwar implementiert, jedoch schließlich auch wieder verworfen.

3.2 Maschinenlernen

Ähnlich wie bei der künstlichen Intelligenz, wird beim Machine Learning die Umgebung als Input aufgenommen und daraus eine Output, die Entscheidung, erzeugt. Der wesentliche Unterschied ist, dass beim Machine Learning der Agent durch Trainieren und das damit verbundene Lernen seine Entscheidungen fällt. Die Entscheidung wird mit dem kontinuierlichen Training besser.

Machine Learning lässt sich in viele verschiedene Disziplinen unterscheiden. Ein konkreter Ansatz ist das so genannte Reinforcement Learning mit einem Artificial Neural Network. Reinforcement Learning beschreibt das Lernverhalten, bei dem der Agent Feedback zu seinen Entscheidungen bekommt und daraus lernen kann. Z.B. lassen sich die Entscheidungen des Agenten durch die konkrete Anzahl an geerntetem Mangan messen und dieses Ergebnis als Feedback geben. Der Agent nutzt nun ein Artificial Neural Network und passt dieses durch das gegebene Feedback an. Das Artificial Neural Network ist, angelehnt an das menschliche Gehirn, ein Netzwerk von Neuroen die ein Input verarbeiten können. Komplexe verknüpfungen zwischen den Neuronen werden so angepasst, dass Inputs ein gewünschtes Ergebnis liefern.

Wichtig ist hierbei auch die Wahl des Inputs. Es ist möglich den kompletten von Roboter erfassbaren Bereich als Input zu überliefern. Dies würde jede Zelle im Umkreis von 200 Metern, sowie deren Zustand beinhalten (geerntet, ungeerntet, anderer Roboter). Zusätzlich ist es sicher hilfreich zu erfassen in welcher Richtung sich das Ziel bzw. der Mittelpunkt aller Roboter befindet.

Es war geplant mithilfe der Machine Learning Library PyBrain `www.pybrain.org` ein solches System aufzubauen. Zeitliche Einschränkungen haben schließlich dazu geführt, diese Idee nicht weiter auszuführen.

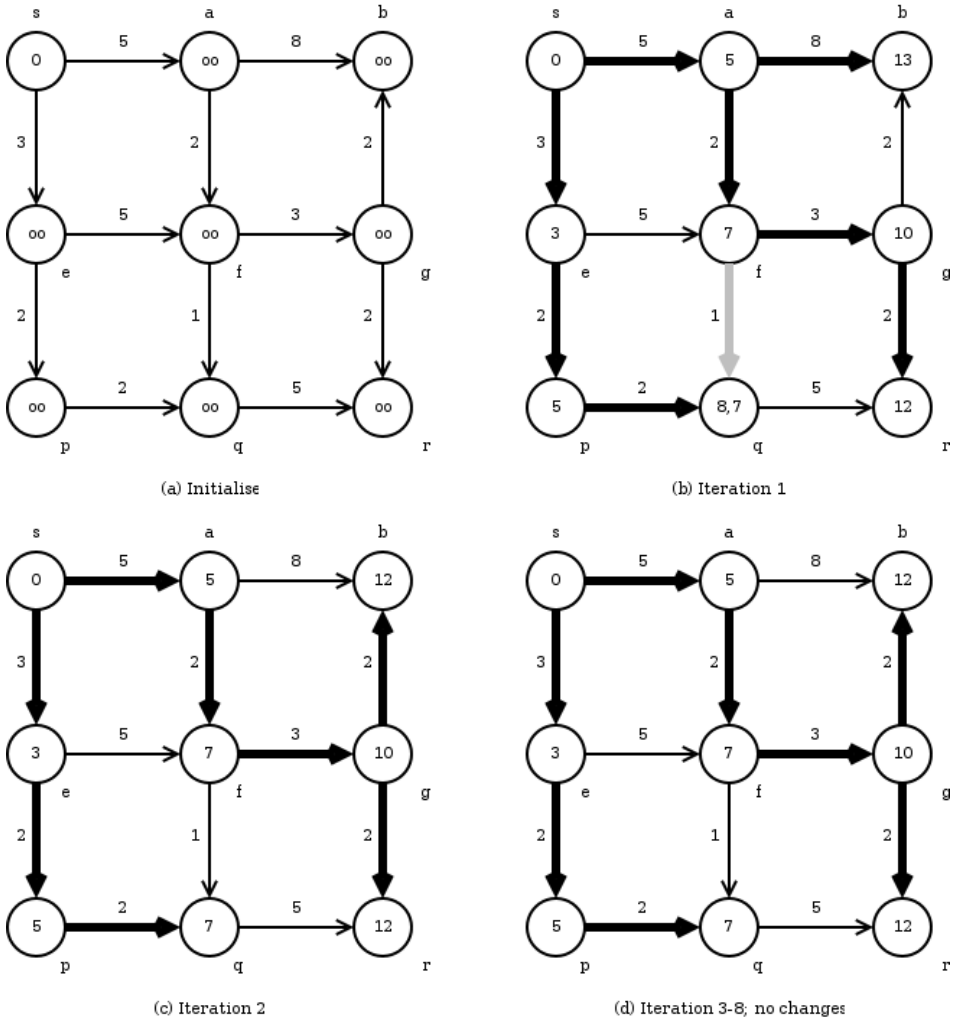


Abbildung 1: Bellman Ford Algorithmus, Iterationen aus Schritt 2

3.3 Künstliche Intelligenz

Das Problem kann auch so betrachtet werden, dass Agenten genannte Akteure nach jedem Zug entscheiden, wie der nächste Zug aussehen soll. Worauf diese Entscheidung fußt, wird in der Implementierung beschrieben, da diese Lösungsidee tatsächlich und schlussendlich realisiert wurde.

4 Umgebung

Die Modellierung des Meeresbodens hat zum Ziel, die folgenden Fragen zu beantworten oder einen guten Kompromiss zu finden:

1. Wie lassen sich eine begrenzte Anzahl an Kreisen auf einer unendlichen Fläche anorden, sodass die nicht von Kreisen bedeckte Fläche minimal ist (vergleichbar mit dem Ausstechen von Kreisen aus Keksteig)?
2. Wie lässt sich der Meeresboden unterteilen, sodass eine Simulation möglichst einfach wird?

Geplant ist, dass die Missionsdauer in Zeitschritt von 1s aufgeteilt wird. In jedem Zeitschritt bewegen sich die Roboter einen geometrischen Schritt auf dem Meeresboden weiter und säubern ihn dabei von Manganknollen. Der Vorteil darin besteht, dass die Optimierung der Ausbeute darauf reduziert wird, den nächsten Schritt möglichst geeignet auszuwählen.

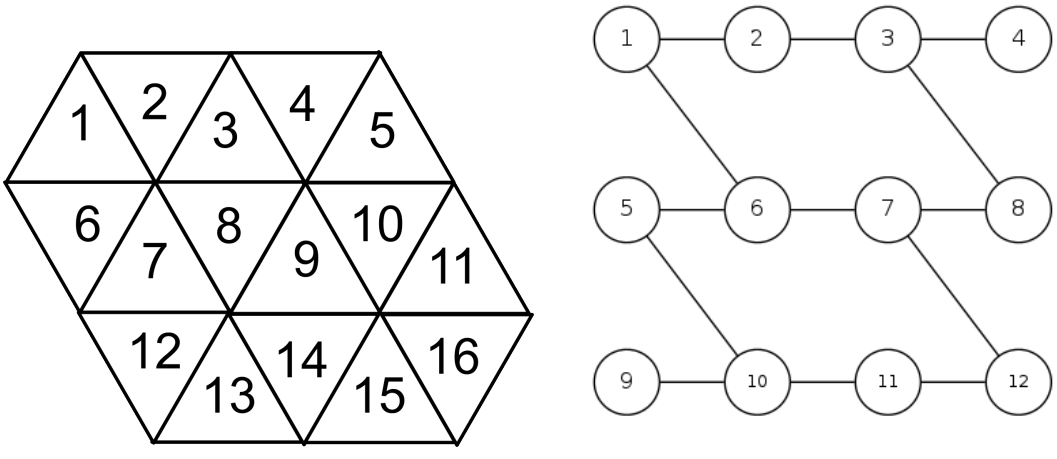
Dazu werden unendlichen Weiten des pazifischen Meeresgrundes in Abschnitte in Form von Polygonen eingeteilt (parkettiert), damit der Wertebereich der nächsten möglichen Schritte diskretisiert wird.

Die Wahl, mit welchem Polygon gearbeitet wird, hat direkte Auswirkungen auf Genauigkeit und Leistungsfähigkeit der Simulation, wie im Folgenden gezeigt wird.

4.1 Konkrete Modellierung des Meeresbodens

Der Meeresboden kann dann als Graph gesehen werden, bei denen Knoten als Zellen gesehen werden und geometrisch einem Polygon entsprechen. Angeordnet werden diese so, dass zwischen den Mittelpunkten der Innenkreise zweier benachbarter Zellen exakt 1m Abstand ist. Dies hat den Grund, dass ein Roboter in jedem Zeitschritt in die Mitte der nächste Zelle wechseln kann.

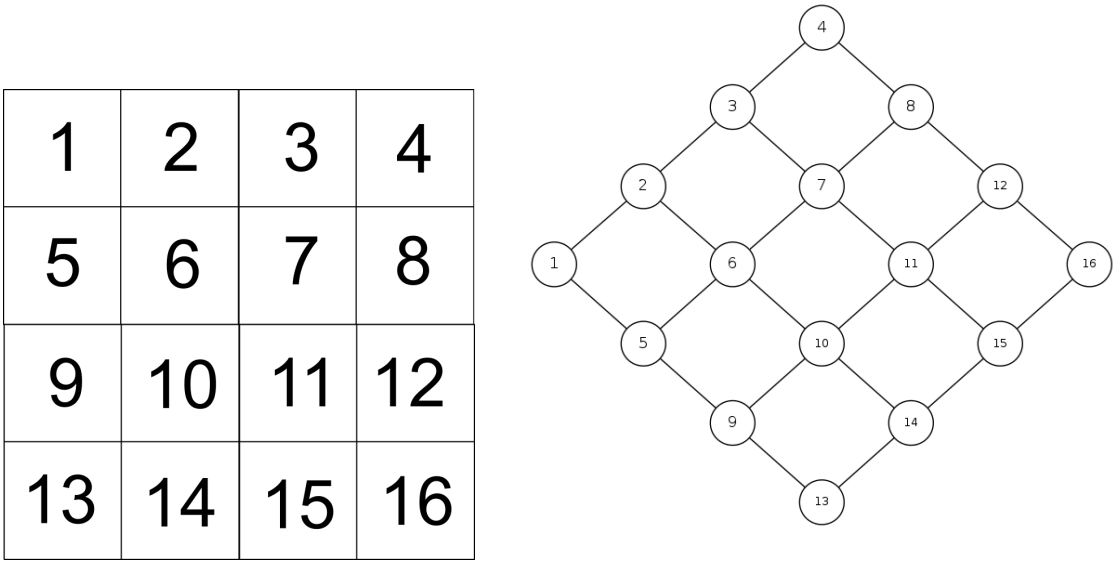
Der Zusammenhang zwischen realer Umgebung und Modellierung ist in Fig. 2 - 11 visualisiert.



(a) (b)

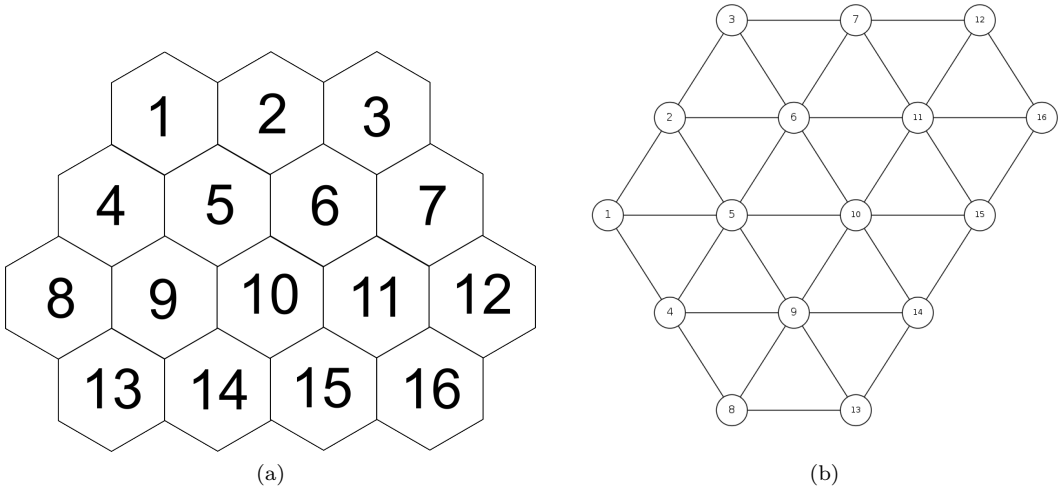
Abbildung 2: Parkettierung mit gleichseitigen Dreiecken

Roboter werden so simuliert, dass die Bewegungen nur über die Kanten einer Zelle möglich sind. Daher ist Anzahl der Ecken identisch mit den möglichen Bewegungsrichtungen. Somit gilt: je mehr Ecken,



(a) (b)

Abbildung 3: Parkettierung mit Quadraten



(a) (b)

Abbildung 4: Parkettierung mit regelmäßigen Sechsecken

desto mehr Freiheitsgrade. Die Kanten geben an, von welchem Knoten zu welchen Nachbarn gewechselt werden kann. Somit hat jeder Knoten auch so viele Kanten wie das gewählte Polygon Ecken hat.

Nach [HH02, S. 12] gilt:

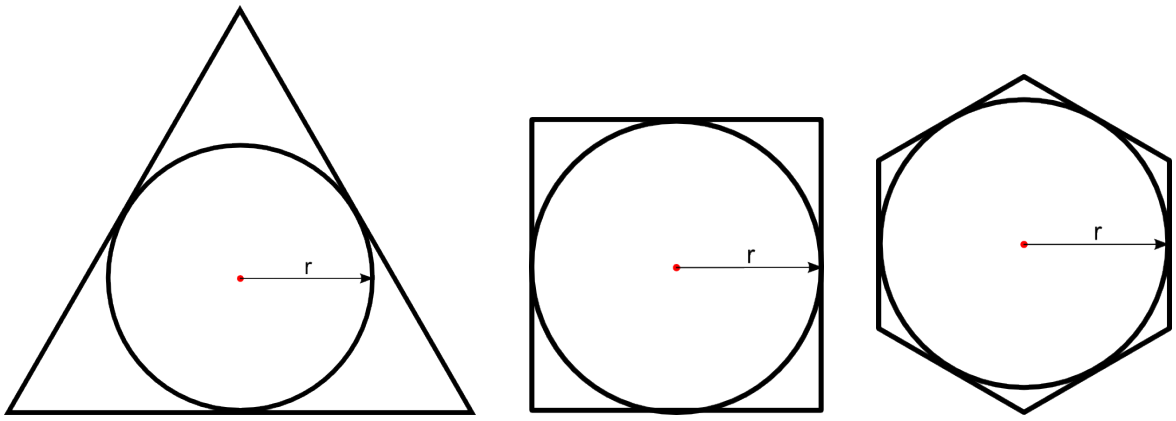
Satz 1. *Eine lückenlose Parkettierung* ***ohne*** *Überschneidungen ist nur mit den regulären n-Ecken für n = 3,4,6 möglich.*

Da eine Simulation mit unregelmäßiger Parkettierung unnötig komplex erscheint, entscheidet es sich zwischen gleichseitigem Dreieck, Quadrat und regelmäßigem Sechseck.

4.2 Polygone und Innenkreis

Der Arbeitsbereich eines Roboters ist kreisförmig, die Zellen, in denen er sich befindet, jedoch ein Polygon. Daher gibt es in den Ecken der Zelle Bereiche, die nicht (unmittelbar) gesaugt werden. Berechnen lässt sich der Verlust Q über das Verhältnis von der Fläche des Innenkreises des Polygons zum Flächeninhalt des Polygons selbst:

$$Q = \frac{A_{\text{Kreis}}}{A_{\text{Polygon}}}$$



(a) (b) (c)

Abbildung 5: Regelmäßiges 3,4,6-Eck mit eingezeichnetem Innenkreis und Radius

Die folgende Tabelle zeigt das Verhältnis von den drei Polygonen.

n	Q
3	60,46 %
4	78,50 %
6	90,69 %

Abbildung 6: Verhältnis Q der Flächeninhalte von Innenkreis eines n-Ecks und dessen gesamten Flächeninhalts

Somit würde zum Beispiel bei Modellierung durch Zellen in Form von regelmäßigen Dreiecken ein Roboter nur 60 % des Mangans darin einsammeln. Je größer n, desto besser wird die unmittelbare Ausbeute.

4.3 Entscheidung

Die Auswahl fiel schließlich leicht, da die Simulation mit einem Quadrat als zugrundeliegendem Polygon folgende Vorteile bietet:

- Einfache Datenstruktur
- Einfaches Berechnen der Nachbarn
- Visualisierung einfach, da eine Zelle einem Pixel entspricht, somit keine Umrechnung nötig

- Weniger Verlust als Dreieck
- Weniger Freiheitsgrade als Sechseck (weniger Auswahl bedeutet weniger Rechenaufwand)
- Einfache Berechnung von Entfernungen zwischen zwei Zellen 5.4

Die tatsächliche Modellierung ist in Fig. 7 dargestellt. Zu beachten ist, dass sich das Gitter unendlich weit in alle Richtungen erstreckt, aber in der Implementierung fast nur der erste Quadrant genutzt wird.

Bewegungen können nur in Nachbarzellen erfolgen. Lediglich die Zellen, welche eine Kante mit der Basisfläche gemeinsam haben, gelten als Nachbarn. Die wird auch Von-Neumann-Nachbarschaft genannt.

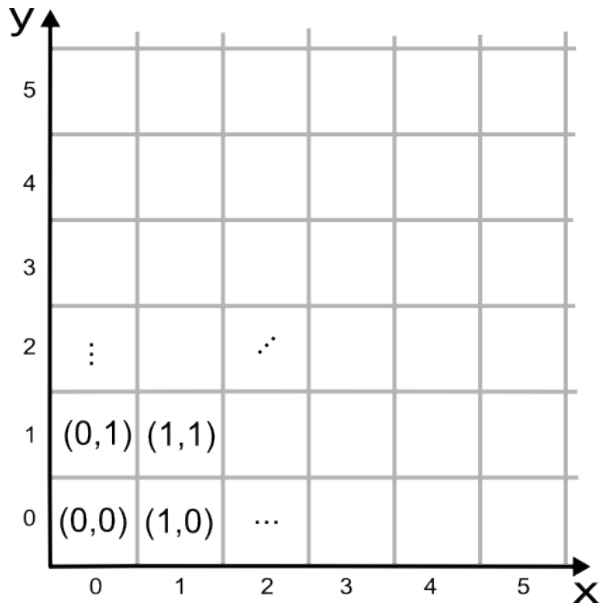


Abbildung 7: Diskretisierung der möglichen Positionen auf dem pazifischen Meeresboden durch quadratische Zellen. Jede Zelle ist eindeutig durch kartesische Koordinaten bestimmt. Das Gitter erstreckt sich unendlich weit in alle Richtungen.

Als wie gut sich diese Entscheidung schließlich herausgestellt hat, wird im Abschnitt HIER[1] evaluiert.

[1] HIER2

5 Theory Crafting

Der folgende Abschnitt beschreibt die zugrundeliegende Theorie hinter der Implementierung.

5.1 Platzierung

Die Platzierung der Roboter R kann wie folgt beschrieben werden: Platziere n Punkte so, dass folgende Bedingung gilt:

$$\forall r \in R \; \exists p \in R : p \neq r \wedge d(p, r) \leq 200 \tag{1}$$

$d(a, b)$ ist hier die euklidische Entfernung zwischen a und b.

Die hier gewählte Lösung ist iterativ. Beginnend mit einem ersten Punkt (gegeben durch `generateFirstPoint`), werden nach und nach Punkte durch `generatePointBoxed` erzeugt und überprüft, ob ein Punkt (1) einhält. Falls ja, wird der Punkt hinzugefügt. `generatePointBoxed` erzeugt einen zufälligen Punkt, der in einem Rechteck $\overline{PQRS}$ liegt, welches als Ecke links unten den Punkt P und als obere rechte Ecke den Punkt R hat. Es gilt:

$$P = (min_x - 200, min_y - 200)$$
$$R = (max_x + 200, max_y + 200)$$

$\texttt{min}_\texttt{x}$, $\texttt{min}_\texttt{y}$, $\texttt{max}_\texttt{x}$, $\texttt{max}_\texttt{y}$ sind hier die minimalen bzw. maximalen x- und y- Werte in R (**nicht** minimaler oder maximaler Punkt).

Dies schränkt die Menge an Punkten ein, die erzeugt werden kann und ist eine grobe Annäherung für (1). Wenn es nicht eingeschränkt würde, gäbe es zu viele Kandidaten zum testen, die eigentlich von vorneherein ausgeschlossen werden könnten.

Es wurde nicht darauf geachtet, wie die Wahrscheinlichkeitsverteilung der Platzierung aussieht, da keine Einschränkungen gegeben wurden, was zufällig genau bedeutet (uniform, normalverteilt, ...). Die zufällig erzeugten Elemente sind grundsätzlich gleichmäßig verteilt, aber die Trial-und-Error Methode könnte das Ergebnis eventuell verfälschen.

Algorithmus 1 : Platzierung von Robotern

Data : Die Anzahl von Robotern n, die platziert werden sollen
Result : Menge von Punkten R, sodass (1) erfüllt wird. Punkte werden zufällig erzeugt.

```
R ←{ generateFirstPoint() };
for i ← 1 to n do
    min_x, min_y, max_x, max_y ←getExtrema(R);
    repeat
        p ←generatePointBoxed(min_x, min_y, max_x, max_y);
        nearest ←nearestNeighbour(R, p)
    until d(p, nearest) ≤ 200;
    R ← R ∪ {p};
end
return R
```

Im Laufe der Zeit hat sich dieser Algorithmus als extrem langsam herausgestellt, und eine einfachere Lösung wurde gewählt (siehe “Platzierung von Robotern II”).

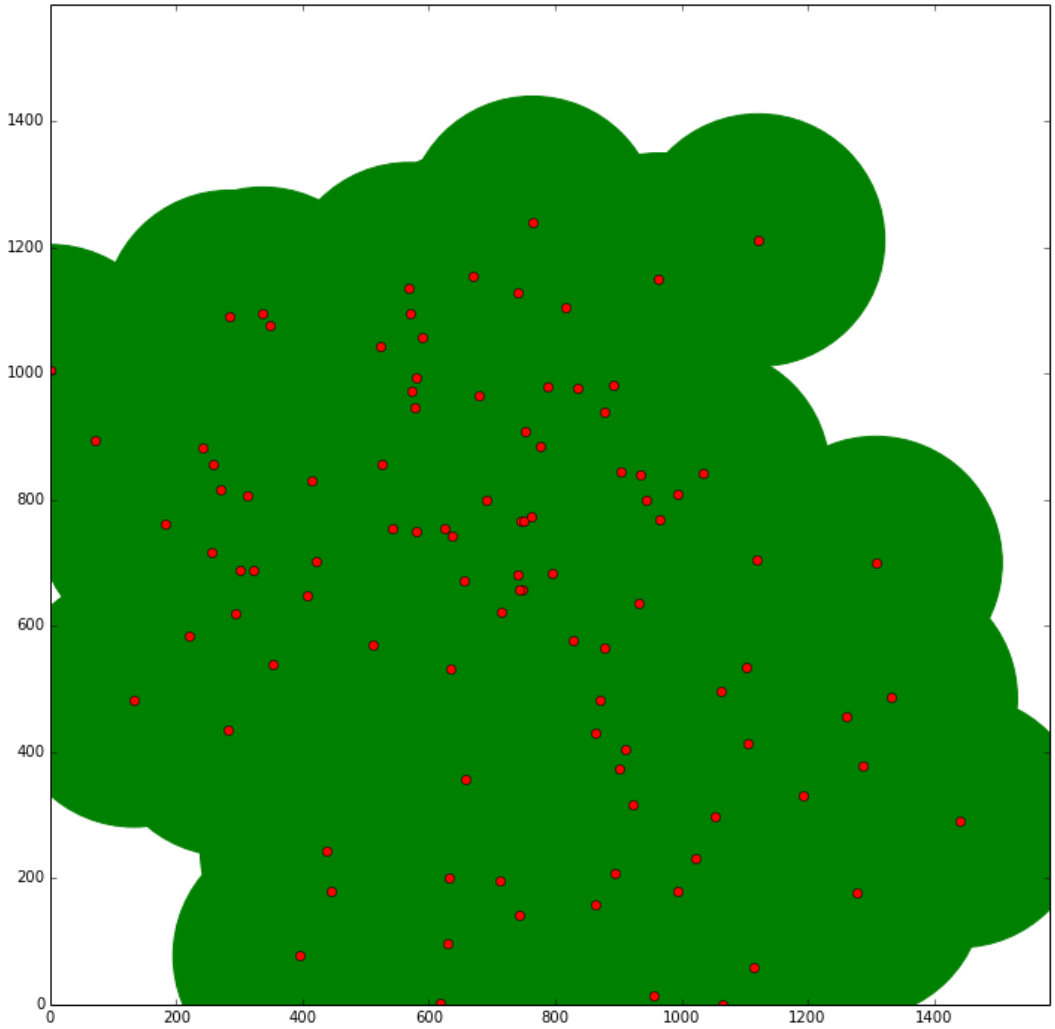


Abbildung 8: Platzierung von 100 Punkten. Die grüne Fläche entsteht aus Kreisen mit Radius 200 um jeden Punkt und bezeichnet die Positionen, für die (1) erfüllt ist. Da die Fläche verbunden ist und alle Punkte enthält, ist diese Anordnung valide.

Algorithmus 2 : Platzierung von Robotern II

Data : Die Anzahl von Robotern n, die platziert werden sollen
Result : Menge von Punkten R, sodass (1) erfüllt wird. Punkte werden zufällig erzeugt.

```
R ←{ generateFirstPoint() };
for i ← 1 to n do
    min_x, min_y, max_x, max_y ←getExtrema(R);
    pivot ←getRandomElement(data);
    repeat
        p ←generatePoint(min_x, min_y, max_x, max_y);
    until d(p, pivot) ≤ 200;
    R ← R ∪ {p};
end
return R
```

5.2 Missionsdauer

Die Missionsdauer ist definiert als die minimal benötigte Zeit für ein Treffen aller Roboter an einem gemeinsamen Ort auf dem Meeresboden. Es kann auch als Sonderfall des 1-center problem angesehen werden, welches sagt: Gegeben sei eine Menge von Punkten R, platziere einen Punkt M so, dass die maximale Entfernung eines Punktes $p \in R$ zu M minimiert wird.

Dieses Problemes wird auf das Problem des kleinsten umschliessender Kreises reduziert. Dieser ist der Kreis, welcher alle Punkte enthält und dabei einen minimalen Radius hat. Dabei ist der Radius dessen die maximale Entfernung oder auch Missionsdauer. Der Mittelpunkt des Kreises ist ein potentieller Treffpunkt der Roboter.

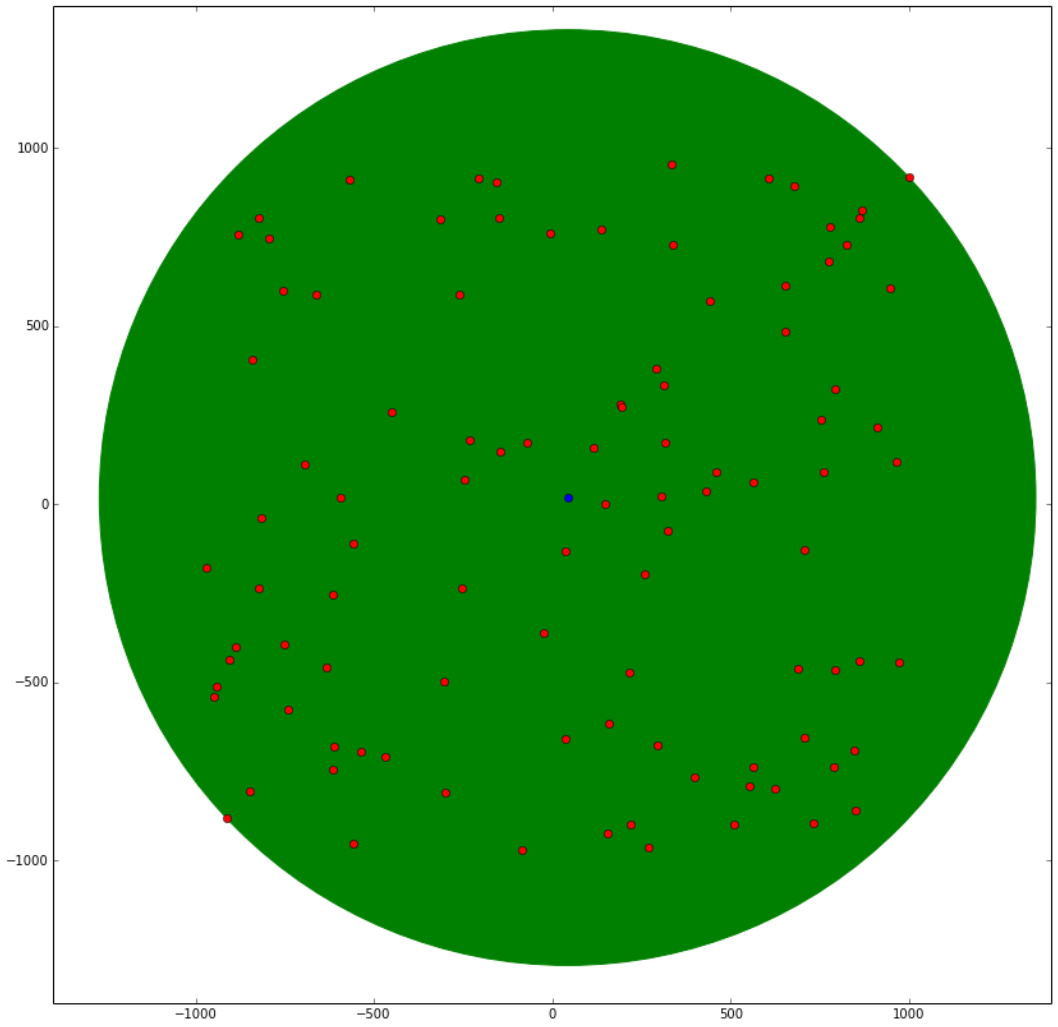


Abbildung 9: Kleinster umschliessender Kreis um eine Menge aus 100 zufällig generierten Punkten

Mit der Zeit sind viele verschiedene Algorithmen entwickelt worden. Der hier verwendete, im weiteren `minidisk` genannt, zeichnet sich dadurch aus, dass er äußerst einfach implementiert werden kann. Außerdem hat er eine Komplexität von erwarteten $\mathcal{O}(n)$. Interessant ist dies, da das Problem offensichtlich $\Omega(n)$ ist (jeder Punkt muss mindestens einmal gesichtet werden, da er prinzipiell auf der Kreislinie

liegt und somit den Radius vergrößern kann).

Vorgestellt wurde er in [Wel91]. Es wird im Folgenden kurz beschrieben, wie er funktioniert. Für Einzelheiten, z.B. warum es funktioniert, sei auf dieses Paper verwiesen. Die Implementierung von `minidisk` ist wie folgt:

Algorithmus 3 : Kleinster umschließender Kreis

Data : Punktmenge P

Result : Radius und Mittelpunkt des Kreises, der alle Punkte p $\in P$ enthält und einen minimalen Radius hat

```
function minidisk(points)
    function bMinidisk(P, R)
        if P is ∅ then
            return empty disk;
        else if |R| <= 3 then
            return disk(R);
        else
            return error;
        end
    end
    function bMinidisk(P, R)
        if P is ∅ or |R| is 3 then
            D ← bMd(∅, R);
        else
            choose random p ∈ D;
            D ← bMinidisk(P \ p, R);
            if p ∉ P then
                D ← bMinidisk(P \ p, R ∪ p)
            end
        end
        return D
    end
    return bMinidisk(points, ∅);
end
```

Zu beachten ist, dass `disk(R)` die kleinste Scheibe berechnet, die alle Punkte in R enthält. Der Algorithmus funkioniert wie folgt: Es gibt zwei Mengen P, R. P sind die Punkte, die noch verarbeitet werden müssen, R die Punkte, die auf dem Rand der Scheibe liegen. In jedem Schritt überprüft, ob es schon eine eindeutige Lösung berechnet werden kann. Dies ist der Fall, wenn P leer ist (kein nächster Schritt möglich) oder R genau drei Elemente enthält (ein Kreis ist eindeutig durch drei Punkte definiert).

Falls keine dieser beiden Bedingungen zutrifft, wird ein Punkt p aus P zufällig ausgewählt und ohne ihn versucht, eine `minidisk` zu berechnen. Falls p dann nicht in der neuen Scheibe liegt, muss er zwangsläufig im nächsten Schritt auf dem Rand liegen.

5.3 Finden der Missionsdaten

Die einfache Missionszeit ist der Radius der `minidisk`, der Treffpunkt ist dessen Mittelpunkt. In Fig. 9 ist dargestellt, wie 100 Roboter (rot) positioniert sind und sich auf einen Treffpunkt (blau) geeinigt

haben. Die Missionszeit hier ist minimal, da der Algorithmus versucht, die maximale Entfernung zu einem Punkt zu minimieren. Da nur mit Ganzzahlen gearbeitet wird, wird der Treffpunkt gerundet. Daher muss auch die Missionszeit angepasst werden, sie wird aufgerundet.

5.4 Manhattan-Metrik

Nachdem in 7 gezeigt wurde, wie genau die Umgebung modelliert wurde, fällt auf, dass die Distanz zwischen zwei Zellen nicht der euklidischen Entfernung entspricht. Dies ist in Fig. 10 gezeigt. Daher muss eine neue Entfernungsfunktion gefunden werden, bei der der Weg nur aus horizontalen und Vertikalen Wegstücken besteht.

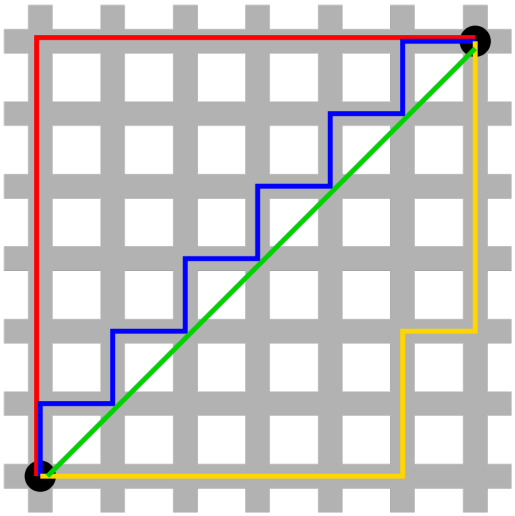


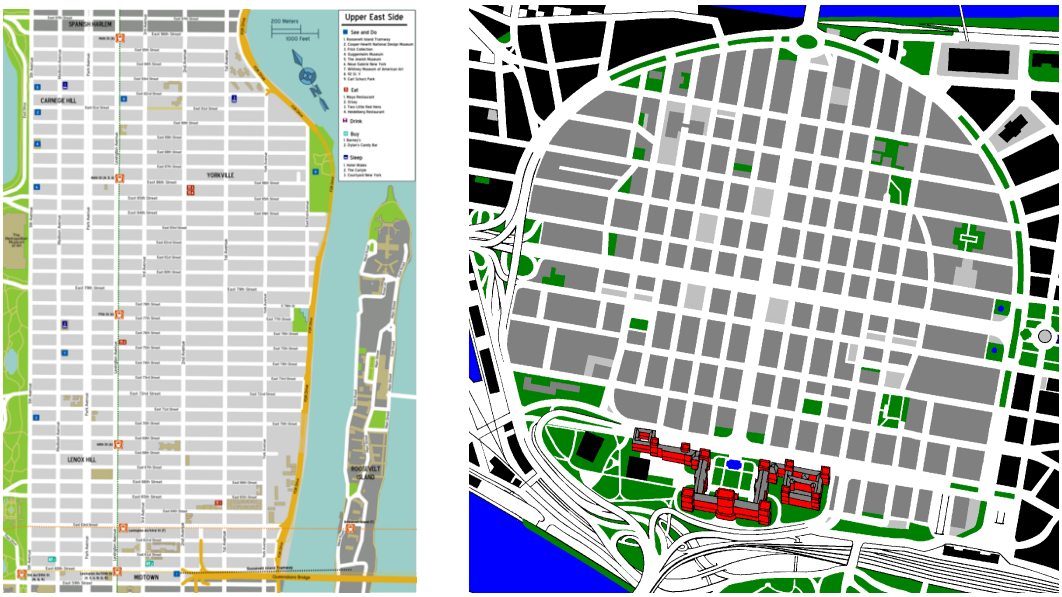
Abbildung 10: Verschiedene, gleichgroße Taxicab-Distanzen zwischen zwei Punkten (je 12 Einheiten lang). Zum Vergleich ist die grüne Linie die euklidische Distanz (etwa 8,5 Einheiten) eingetragen. [Wik]

Glücklicherweise wurde eine solche Geometrie, in der die Entfernung so definiert ist, bereits erforscht. Die Manhattan-Geometrie (oder Taxicab-Geometrie, Cityblock-Geometrie), ist eine Geometrie, in der die übliche, euklidische Entfernungsfunktion durch eine neue Metrik ersetzt ist. Eine Metrik ist hier eine Funktion, die zwei Elemente aus der Geometrie einen reele, nichtnegativen Abstand zuweist.

Ihr Name entstammt der Analogie mit dem Straßennetz von Manhattan (oder Mannheim, wo sich die Universität der beiden Autoren befindet). Es ist gitterförmig angelegt. Um ein Ziel zu erreichen, wird die Entfernung durch Aneinanderreihung von vertikalen und Horizontalen Stücken zurückgelegt.

Ein Taxifahrer, welcher eine Route durch solch eine System plant, legt immer die gleiche Strecke zurück, wenn er Wege benutzt, die ihn näher zum Ziel bringen, egal welche Wahl er dabei an Abzweigungen trifft.

Da diese Geometrie der Modellierung des Meeresbodens durch Quadrate entspricht, werden im Folgenden notwendige und wichtige Eigenschaften derer beschrieben.



(a) New York (b) Mannheim

Abbildung 11: Auszüge aus Stadplänen von New York und Mannheim.
Auffallend ist die gitterförmige Anordnung der Straßenzüge.

5.4.1 Entfernung

Die Entfernung zwischen zwei Punkten $a, b \in \mathbb{R}^n$ in einer Manhattan-Geometrie ist die Summe der absoluten Differenzen derer kartesischer Einzelkoordinaten:

$$d(a,b) = \sum_{i=1}^{n} |a_i - b_i|$$

Für den hier relevanten Spezialfall von $\mathbb{R}^2$ gilt somit:

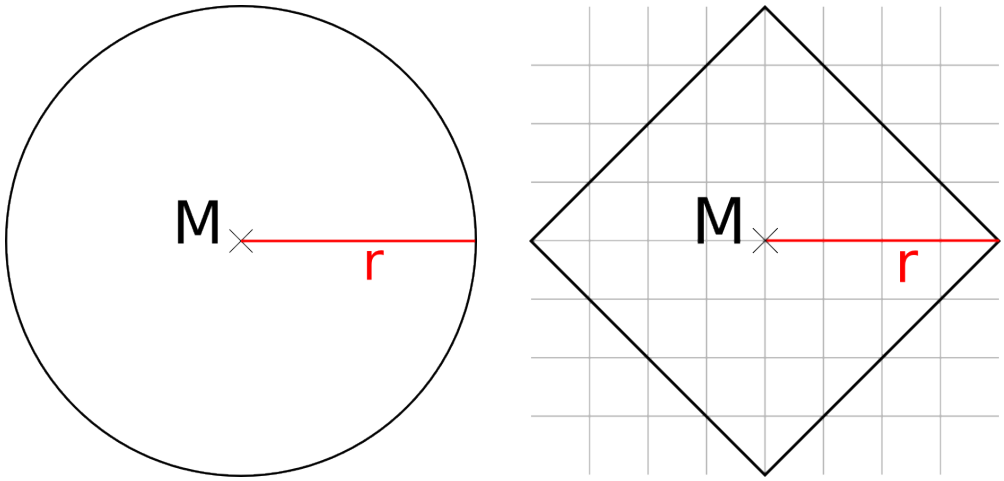
$$d(a,b) = |a_1 - b_1| + |a_2 - b_2|$$

5.4.2 Kreise

Da die Berechnung der Missionszeit auf der Berechnung von Kreisen basiert, müssen diese auch in der verwendeten Taxicab-Geometrie betrachtet werden. Bekannt ist, dass ein Kreis die Menge aller Punkte in einer Ebene ist, welche einen konstanten Abstand, dem Radius, zu einem bestimmten Punkt, dem Mittelpunkt, haben. Die gewählte Metrik, die in dem Wort *Abstand* versteckt ist, hat unmittelbar Einfluss auf die Form des Kreises.

Kreise in einer Taxicab-Geometrie haben die Form von Quadraten, die 45 ° um deren Mittelpunkt gedreht sind. Fig. 12 gibt eine Idee, wieso es so ist. Eine ausführliche Behandlung ist in [Jan07] zu finden.

Die Berechnung von Mittelpunkt und Radius eines Taxicab-Kreises ist schwieriger als in euklidischer Geometrie. Es wird hier keine Formel benutzt, sondern ein Algorithmus. Interessant hier ist das Berech-



(a) Euklidischer Kreis (b) Taxicab-Kreis

Abbildung 12: Alle Punkte auf der Kreislinie haben den gleichen Abstand zum Mittelpunkt. Auffallend ist, das der Kreis in Taxicab-Metrik nicht rund, sondern quadratisch ist.

nen von einem Kreis in Taxicab-Geometrie von zwei oder drei Punkten. Das Problem lässt sich auch wie folgt beschreiben: Finde das kleinste Quadrat, welches um 45 ° gedreht ist, welches zwei (drei) Punkte enthält. Da es nicht eindeutig definiert wird, reicht eine einzige Lösung für dieses Problem.

Algorithmus 4 : Kreis von zwei Punkten in Taxicab-Geometrie

Data : Zwei Punkte $X, Y \in \mathbb{R}^n$

Result : Radius und Mittelpunkt eines Quadrates, das X und Y enthält und eine minimale Seitenlänge hat. Radius hier ist die halbe Länge der Diagonale.

1. Um die Lösung des Problems zu vereinfachen, werden die Punkte je um 45 ° um den Ursprung herum gedreht, dann das Quadrat berechnet, die so enstandende Lösung schließlich zurückgedreht.
2. Rotiere X, Y 45 ° um den Ursprung und nenne die so entstehenden Punkte P, Q.
3. Offensichtlich ist nun die Seitenlänge s des minimalen Quadrates gegeben durch

$$s = \max\{|P_x - Q_x|, |P_y - Q_y|\}$$

4. Um eine Lösung zu finden, wähle die unterste linke Ecke als Ausgangspunkt und nenne sie A.

$$\begin{cases} A_x = \min\{P_x, Q_x\} \\ A_y = \min\{P_y, Q_y\} \end{cases}$$

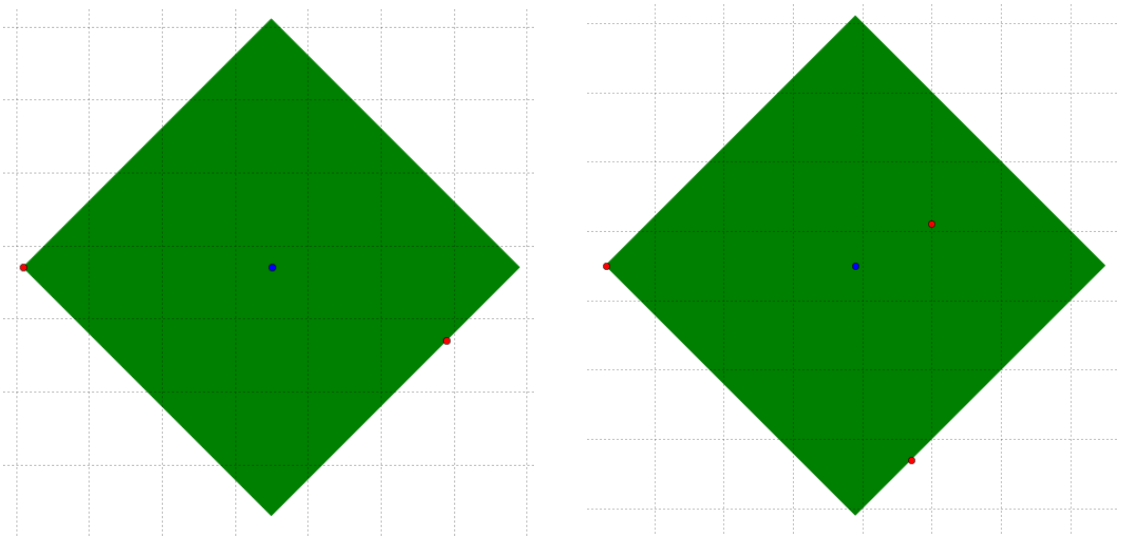
5. Der rotierte Mittelpunkt M' ist somit gegeben durch

$$\begin{cases} M'_x = A_x + \frac{s}{2} \\ M'_y = A_y + \frac{s}{2} \end{cases}$$

6. Rotiere M' um -45 ° um den Ursprung und nenne den so entstehenden Punkt M.
7. Der Radius r ist gegeben durch

$$r = \sqrt{A_x - M'_x)^2 + (A_y - M'_y)^2}$$

return r, M



(a) Zwei Punkte (b) Drei Punkte

Abbildung 13: Kreis in Taxicab-Geometrie von zwei respektiv drei Punkten. Auffallend ist, dass mindestens zwei Punkte auf der Kreislinie liegen.

Interessanterweise liefert dieser Algorithmus durch geringfügige Anpassungen auch die Lösung für folgendes Problem: Finde das kleinste Quadrat, welches um 45 ° gedreht ist, welches **drei** Punkte enthält:

Algorithmus 5 : Kreis von drei Punkten in Taxicab-Geometrie

Data : Drei Punkte $X, Y, Z \in \mathbb{R}^n$

Result : Radius und Mittelpunkt eines Quadrates, das X, Y, Z enthält und eine minimale Seitenlänge hat. Radius hier ist die halbe Länge der Diagonale.

1. Um die Lösung des Problems zu vereinfachen, werden die Punkte je um 45 ° um den Ursprung herum gedreht, dann das Quadrat berechnet, die so enstandende Lösung schließlich zurückgedreht.
2. Rotiere X, Y, Z 45 ° um den Ursprung und nenne die so entstehenden Punkte P, Q, R.
3. Offensichtlich ist nun die Seitenlänge s des minimalen Quadrates gegeben durch

$$s = \max\{|P_x - Q_x|, |P_x - R_x|, |Q_x - R_x|, |P_y - Q_y|, |P_y - R_y|, |Q_y - R_y|\}$$

4. Um eine Lösung zu finden, wähle die unterste linke Ecke als Ausgangspunkt und nenne sie A.

$$\begin{cases} A_x = \min\{P_x, Q_x, R_x\} \\ A_y = \min\{P_y, Q_y, R_y\} \end{cases}$$

5. Der rotierte Mittelpunkt M' ist somit gegeben durch

$$\begin{cases} M'_x = A_x + \frac{s}{2} \\ M'_y = A_y + \frac{s}{2} \end{cases}$$

6. Rotiere M' um -45 ° um den Ursprung und nenne den so entstehenden Punkt M.
7. Der Radius r ist gegeben durch

$$r = \sqrt{A_x - M'_x)^2 + (A_y - M'_y)^2}$$

return r, M

Im Rahmen dieser Arbeit wurden diese Algorithmen auch mittels GeoGebra visualisiert. Eine interaktive Visualisierung für einen Kreis in Taxicab-Geometrie, der zwei (drei) Punkte enthält, ist unter folgenden Links zu sehen:

`http://www.geogebratube.org/student/m65241`
`http://www.geogebratube.org/student/m68655`

Interessanterweise funktioniert `minidisk` auch mit Taxicab-Kreisen, wenn man die `disk`-Funktion anpasst. Dies wurde hier nicht mathematisch bewiesen, aber die Ergebnisse sprechen für sich.

5.5 Zeitbeschränkung

Das Einhalten Zeitbeschränkung ist nun trivial. Da es einen festen Treffpunkt gibt, sind nur Bewegungen in Zellen erlaubt, die einen Abstand von gleich oder weniger als dem Wert der verbliebenen Schritten haben. Mit jedem neuen Schritt wird dieser Kreis um den Treffpunkt kleiner, und die Roboter treffen sich zwangsläufig in einem Punkt. Ein Schritt hier ist vollzogen, wenn alle Roboter ihren Zug gemacht haben (Bewegung in benachbarte Zelle oder NOP).

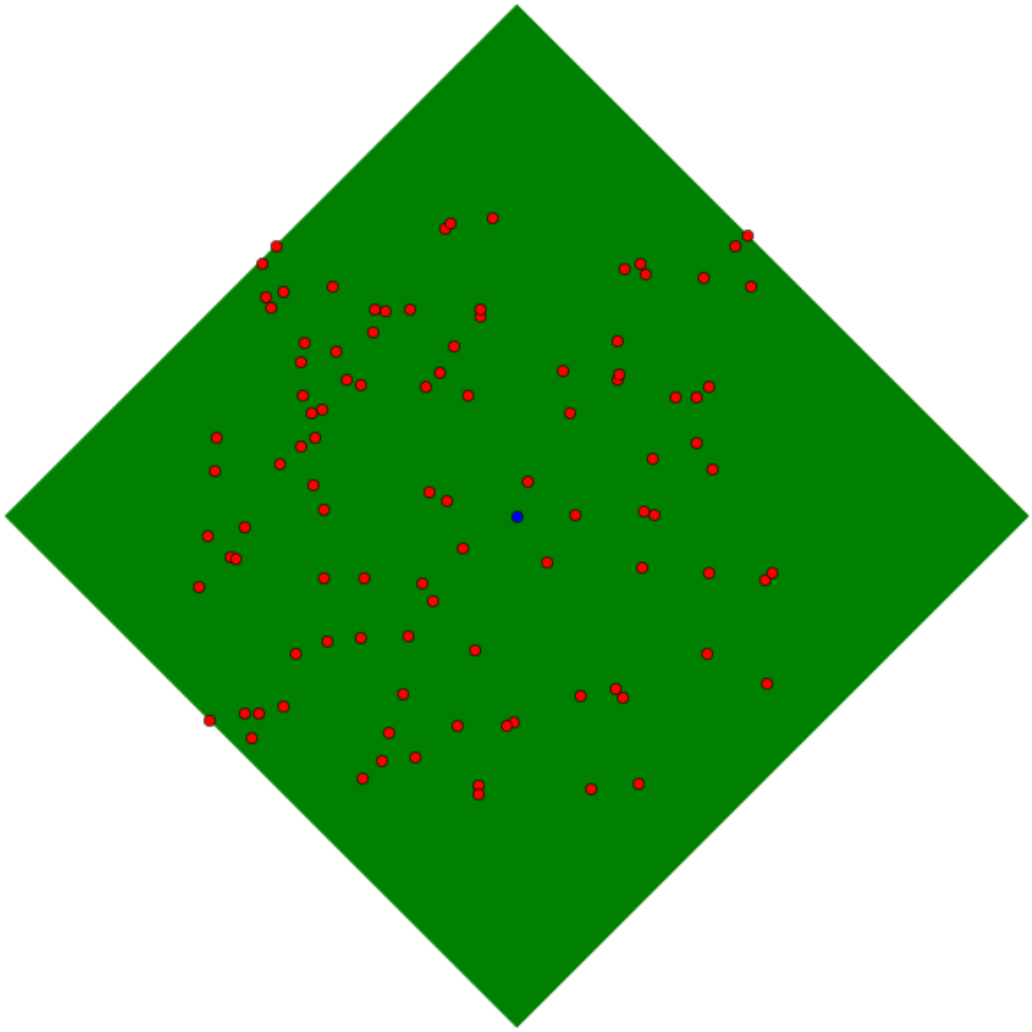


Abbildung 14: Platzierung von 100 Punkten und deren kleinster umschließender Kreis in Taxicab-Geometrie.

6 Implementierung

Die Implementierung beruht auf der Modellierung der Umgebung als Gitter von quadratischen Zellen. Zunächst wird, wie zuvor beschrieben, Missionsdauer und Ziel bestimmt. Falls mit der doppelten Missionsdauer gerechnet werden soll, muss `steps` einfach verdoppelt werden. In jedem Schritt werden die Roboter nacheinander bewegt. Wichtig ist hier, das in-place gearbeitet wird, damit für jeden Roboter die Invariante aus (1) gilt. Ein Roboter macht einen Schritt und guckt auf die Positionen der anderen Roboter und macht dann seinen Zug. Der nächste Roboter betrachtet dann die neue Position des zuvor berechneten Roboters. Da bei einem Zug beachtet wird, dass er legal sein muss (kann Ziel erreichen und Roboter in der Nähe), bleibt diese Invariante auch invariant.

Die eigentliche Logik ist in der `agent`-Funktion. Diese aktualisiert den Wert des gesammelten Mangans und der gelaufenen Strecke. Viel wichtiger ist aber, dass dort entschieden wird, ob und in welche Zelle sich ein Roboter bewegt.

Algorithmus 6 : Mangan Harvest

Data : Menge von Robotern R
Result :

```
circle ←minidisk(R);
steps ←radius(circle);
goal ←center(circle);
for t ← 1 to steps do
    timeleft ←steps - t; for n ← 1 to |R| do
        agent(n, timeleft);
    end
end
return R
```

6.1 Logik

Jeder Roboter soll eigens mit seinen Sensoren seine Umgebung erfassen und daraus eine rationale Entscheidung treffen. Dabei kann er andere Roboter sowie geerntete und ungeerntere Felder in einem Umkreis von 200 Metern erfassen. Eine Entscheidung beinhaltet den nächsten Zug in eine der benachbarten quadratischen Zellen, sodass jeder Roboter bis zu 4 Auswahlmöglichkeiten pro Zug besitzt.

6.1.1 Heuristic Agent

Der verfolgte Lösungsansatz eines Heuristic Agent lässt sich in drei Heuristiken gliedern. Eine Heuristik hier ist etwas wie eine Daumenregel, eine Entscheidungshilfe, von der man ausgeht, dass sie gute Ergebnisse liefert.

1. Von einen Set aller benachbarten Zellen, die besucht werden können, filtere jene Zellen heraus, die noch nicht geerntet wurden. Falls alle Zellen bereits geerntet sind, behalte alle Zellen im Set.
2. Falls mehr als eine Zelle zur Auswahl steht, wähle jene Zellen, bei der die Richtung in die sich bewegt wird eine höhere Dichte and ungeernteten Zellen aufweist. Dazu wird ein quadratisches Feld von definierbarer größe wie in Abbildung [ref] vor dem Roboter auf gebaut und berechnet wie viele ungeerntete Zellen in diesem Feld vorhanden sind. Ebenfalls wirkt sich ein anderer Roboter, der sich in diesem Feld befindet negativ auf die Gewichtung aus. Schließlich werden

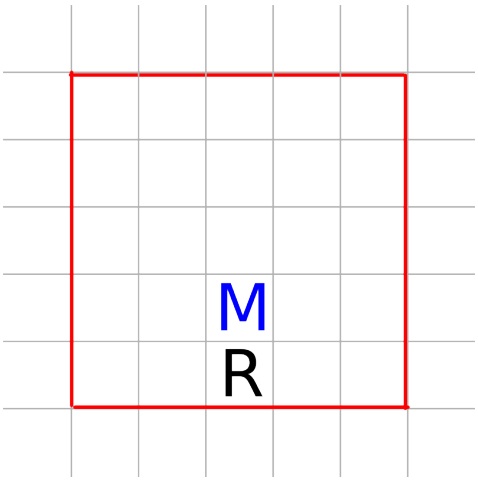


Abbildung 15: Schritt 2 des Heuristic Agent
R - Roboter M - Zug, der auf Dichte untersucht wird
Roter Bereich - Quadrat der Größe 5, dessen Zellen in die Dichteberechnung einfließen

ein oder mehrere Felder mit der höchsten Dichte an ungeernteten Feldern ausgewählt und die dazugehörigen Zellen in die Entscheidungsfindung weiter aufgenommen.

3. Falls mehr als eine Zelle zur Auswahl steht, wähle jene Zelle, die die Distanz zu den umstehenden Robotern maximiert. Dazu werden die Zellen gefiltert, welche die Distanz zum naheligensten Roboter maximieren. Falls es mehr als eine Zelle ist, wird die Distanz zum zweitnähesten Roboter maximiert usw.

4. Falls mehr als eine Zelle zur Auswahl steht (zu diesem Zeitpunkt eher unwahrscheinlich), eine beliebige Zellen aus den Entscheidungsmöglichkeiten.

6.2 Collected

Da auf Genauigkeit Wert gelegt wird, wird nicht nur berechnet, wie viele Zellen ein Roboter besucht hat, um auf die gesammelten Manganmenge zu schließen. Wenn ein Roboter eine Zelle verlässt und in eine Benachbarte wechselt, so überstreicht sein Arbeitsbereich auch die Ecken. Dieser Vorgang ist in Fig. 16 dargestellt.

Die gesammelte Menge an Mangan $M[kg]$ berechnet sich wie folgt:

$$M = |Z| \times \frac{\pi}{4} + |E| \times \frac{4 - \pi}{16}$$

wobei Z die Menge an besuchten Zellen ist, und E die Menge an überstrichenen Ecken (keine Duplikate). Hintergrund ist hier, dass der Innenkreis einen Anteil von $\frac{\pi}{4}$ an der Zelle hat, somit jede Ecke $\frac{1}{4} \times (1 - \frac{\pi}{4})$.

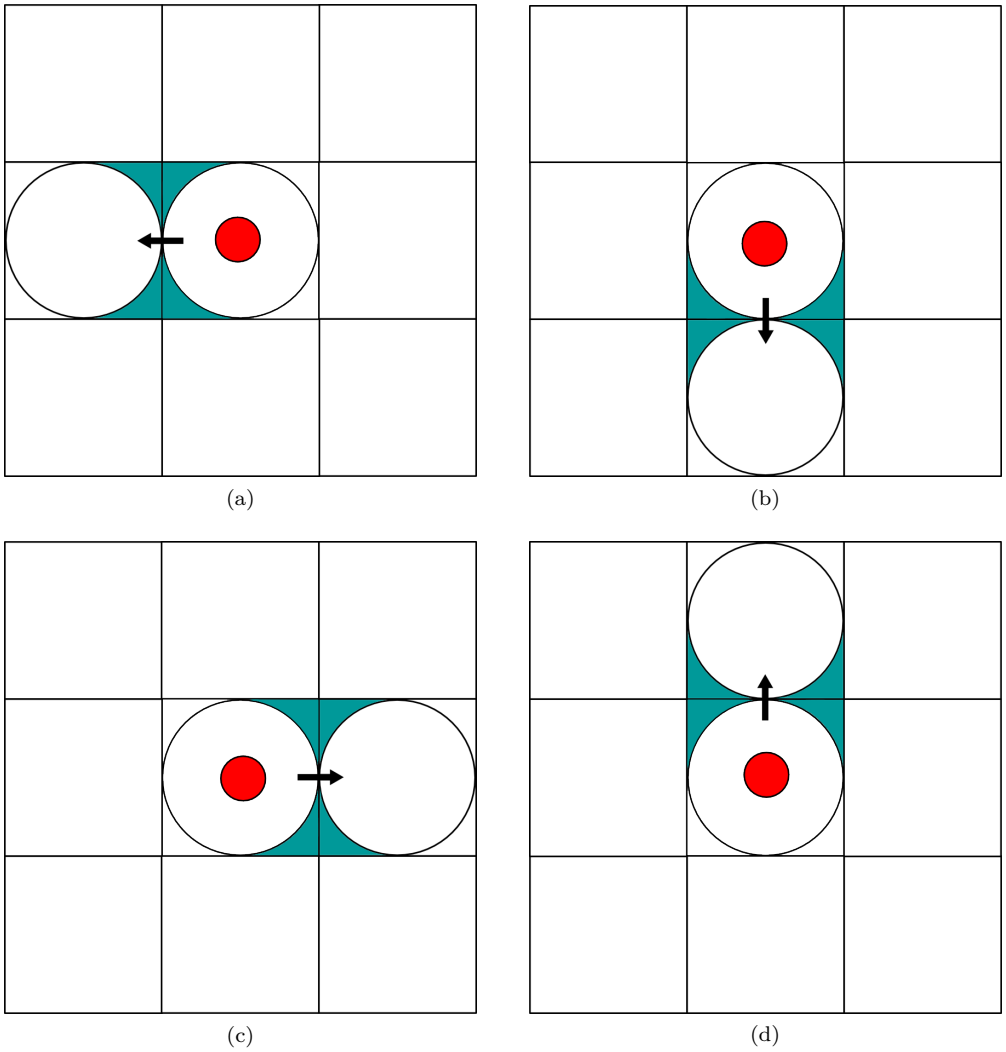


Abbildung 16: Überstreichen der Ecken beim Wechseln in eine benachbarte Zelle: Der Roboter (hier rot), kann nicht nur den ursprünglichen, kreisförmigen Teil in der Mitte einer Zelle abernten, sondern beim Wechseln auch die Ecken, da sein Arbeitsbereich diese in der Bewegung überschneidet.

7 Diskussion

7.1 Performance

Die Zeit für Berechnungen ist kaum zu merken; am Längsten dauert noch die Visualisierung. Die einzige Einschränkung, die es der Zahl zu simulierenden Robotern gibt, ist die Zeit, die man warten möchte.

7.2 Fehlerrechnung

7.2.1 Fehler durch Modellierung

In der Modellierung selbst gibt es keine Fehler, wie es zum Beispiel durch Annäherung der kreisförmigen Arbeitsbereiche durch Polygone der Fall wäre.

7.2.2 Abweichung von der Optimalen Lösung

Die maximale Menge an Mangan M_max, die gesammelt werden kann, lässt sich wie folgt berechnen:

$$M_max = |R| \times T$$

wobei T die Missionsdauer ist. Im Folgenden wird für verschiedene Szenarien jeweils die prozentuale Zielereichung aufgezeigt. Diese Dateien entsprechen denen, die abgegeben wurden.

n	Single	Double
4	99.61	99.68
10	94.84	97.24
50	78.91	91.08
100	88.74	91.77

Abbildung 17: Erreichung des Optimums in %
für verschiedene Anzahl von Robotern

Literatur

[HH02] **Christian Hartfeldt** und **Prof. Dr. Herbert Henning**. *Muster, Flächen, Parkettierungen — Anregungen fur einen kreativen Mathematikunterricht*. Techn. Ber. Otto-von-Guericke-Universität Magdeburg, 2002. URL: http://www.math.uni-magdeburg.de/reports/2002/parkett.pdf.

[Jan07] **Christina Janssen**. *Taxicab Geometry: Not the Shortest Ride Across Town*. Techn. Ber. Iowa State University, 2007. URL: http://www.math.iastate.edu/thesisarchive/MSM/JanssenMSMSS07.pdf.

[Kli08] **Peter Kling**. *Facility Location*. Techn. Ber. Universität Paderborn, 2008.

[Wel91] **Emo Welzl**. "Smallest Enclosing Disks (balls and Ellipsoids)". In: *Results and New Trends in Computer Science*. Springer-Verlag, 1991, S. 359–370. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.1450&rep=rep1&type=pdf.

[Wik] **Wikipedia**. *Manhattan-Metrik*. URL: https://de.wikipedia.org/wiki/Manhattan-Metrik.