Institut für Theoretische Informatik
Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Bachelorarbeit

Random-Walk-Algorithmen

Nico Ostendorf (10013970)

5. Dezember 2020

Erstprüfer:	PD Dr. rer. nat. habil. Arne Meier
Zweitgutachter:	Prof. Dr. rer. nat. Heribert Vollmer
Betreuer:	PD Dr. rer. nat. habil. Arne Meier

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, 07.12.2020

_____

(Nico Ostendorf)

Inhaltsverzeichnis

1. Einleitung

Wie wahrscheinlich ist es, in einer unbekannten Stadt, in welcher die Straßen im Schachbrettmuster angeordnet sind, irgendwann wieder zurück zum Ausgangspunkt zu gelangen, wenn man an jeder Kreuzung zufällig abbiegt, zurück oder geradeaus geht? Mit dieser oder einer ähnlichen Frage haben sich, besonders im Urlaub, die meisten Personen wahrscheinlich schon einmal beschäftigt. Eine ähnliche Frage hat sich auch Karl Pearson im Jahre 1905 gestellt und in der Zeitschrift *Nature* die folgende Frage veröffentlicht:

„A man starts from a point O and walks I yards in a straight line; he then turns through any angle whatever and walks another I yards in a second straight line. He repeats this process n times. I require the probability that after these n stretches he is at a distance between r and $r + \delta r$ from his starting point, O.“[Pea05a]

Nur fünf Wochen später bedankte sich Pearson bereits öffentlich bei Lord Rayleigh für dessen Antwort, die zur Lösung des Problems führte [Pea05b]. Durch Pearsons Frage und Lord Rayleighs Antwort wurde der Begriff des Random Walks geprägt.

Aktuell (November 2020) erzielt eine Suchanfrage in der *ACM digital libary* mit dem Schlagwort „Random Walk“ circa 7.700 Treffer. Allein im Jahr 2020 wurden 499 neue Publikationen zu diesem Schlagwort in der *ACM digital libary* veröffentlicht. Diese Zahlen verdeutlichen das wissenschaftliche Interesse an der Thematik des Random Walks.

Diese Bachelorarbeit beschäftigt sich ebenso mit dem Thema „Random Walk“, wobei der Fokus darauf liegt, gewonnene Erkenntnisse über verschiedene Random-Walk-Algorithmen, welche im Laufe der Zeit entstanden sind und teilweise auch noch Anwendung finden, zusammenzufassen und zu vergleichen. Dabei werden Algorithmen aus verschiedenen Themenbereichen vorgestellt, um zu verdeutlichen, wie vielfältig der Random Walk eingesetzt werden kann.

Im zweiten Kapitel werden alle benötigten Grundlagen für das Verständnis der später aufgeführten Random-Walk-Algorithmen erklärt. Dazu gehören sowohl mathematische Grundlagen, als auch solche der theoretischen Informatik. Des Weiteren wird die grundlegende Funktion eines Random Walks erklärt, auf der die Algorithmen, welche in den darauffolgenden Kapiteln aufgeführt sind, aufbauen.

In den Kapiteln drei bis sieben sind verschiedene Random-Walk-Algorithmen aus unterschiedlichen Bereichen aufgeführt. Jeder Algorithmus wird dabei anhand einer theoretischen Ausführung und eines Beispiels erklärt. Zusätzlich werden die Algorithmen in der Regel durch Pseudocode dargestellt.

Jedes der Kapitel endet mit einem Vergleich, in welchem die Algorithmen des Kapitels oder kapitelübergreifend auf Basis der Funktionsweise, der Laufzeit und des Anwendungsgebietes gegenüber gestellt werden.

Im Fazit wird ein Überblick von allen aufgeführten Algorithmen und ein zugehöriger Vergleich dargestellt. Darauf folgt ein Ausblick über weitere interessante Themen in dem Bereich der Random-Walk-Algorithmen.

Im Anhang befinden sich zu einigen der Algorithmen Beispiele in Form von Python-Code. Diese Beispiele sollen dem besseren Verständnis und der Veranschaulichung des jeweiligen Algorithmus dienen. Im zweiten Kapitel werden daher alle notwendigen Grundlagen zur Verwendung von Python erklärt. Sofern Probleme bei der Implementierung in Python aufgetreten sind, wurden die entsprechenden Python-Code-Beispiele im jeweiligen

Kapitel anstelle des Pseudocodes aufgeführt. Die aufgetretenen Probleme und die jeweiligen Lösungen wurden dabei ausführlich erläutert. Zusätzlich wurden alle Programme ausführlich kommentiert, um den Code nachvollziehbar zu machen.

2. Grundlagen

Im folgenden Kapitel werden die für das Verständnis einiger Algorithmen benötigten Grundlagen ausführlich erläutert und mit Definitionen sowie Beispielen gestützt.

2.1. Grundlagen der Analyse von Algorithmen

Zuerst wird erklärt, was ein Algorithmus ist und welche Eigenschaften dieser erfüllen muss.

Definition 1 (Algorithmus [Fec18])
Ein Algorithmus *ist eine Sequenz von Anweisungen zur Durchführung eines Arbeitsablaufs oder zur Lösung eines Problems.*

Nach dem Informatiker Donald Knuth muss ein Algorithmus die folgenden fünf Eigenschaften erfüllen [Knu68]:

Endlichkeit: Ein Algorithmus muss immer nach einer endlichen Anzahl von Schritten beendet werden.

Bestimmtheit: Jeder Schritt eines Algorithmus muss genau definiert sein. Die durchzuführenden Maßnahmen müssen für jeden Fall eindeutig festgelegt werden.

Eingabe: Eingaben sind Größen, die vor Beginn des Algorithmus oder dynamisch während der Ausführung des Algorithmus übergeben werden. Diese Eingaben stammen aus bestimmten Objektgruppen. Ein Algorithmus hat keine oder mehr Eingaben.

Ausgabe: Jeder Algorithmus hat eine oder mehrere bestimmte Ausgaben, welche eine festgelegte Beziehung zur Eingabe besitzen.

Effektivität: Es wird erwartet, dass ein Algorithmus in dem Sinne effektiv ist, dass seine Operationen alle so grundlegend sein müssen, dass sie im Prinzip von jemandem, der Bleistift und Papier verwendet, fehlerfrei und in endlicher Zeit ausgeführt werden können.

Algorithmen dienen der Problemlösung. Ein bekanntes Problem, welches in der vorliegenden Arbeit betrachtet wird, ist 3SAT. Dieses Problem ist eine Variante des Erfülbarkeitsproblems der Aussagenlogik (SAT), weshalb zuerst SAT noch einmal aufgegriffen wird und dann die Definition von 3SAT folgt.

Definition 2 (SAT [MV15])
$\text{SAT} = \{ \langle \varphi \rangle \mid \varphi \textit{ ist aussagenlogische, erfüllbare Formel} \}$

Definition 3 (3SAT [MV15])
$\text{3SAT} = \{ \langle \varphi \rangle \mid \varphi \textit{ ist eine erfüllbare aussagenlogische Formel in 3KNF} \}$

Komplexitätsklassen dienen der Einordnung solcher Probleme. Im Folgenden werden die für diese Arbeit benötigten Komplexitätsklassen vorgestellt.

Definition 4 (TIME ([MV15] Def. 6))
Sei $t\colon \mathbb{N} \to \mathbb{N}$. *Die Komplexitätsklasse* TIME(t) *besteht aus allen Sprachen A, für die es eine Mehrband-Turingmaschine gibt, die A entscheidet und in Zeit* $O(t)$ *arbeitet.*

Definition 5 (NTIME ([MV15] Def. 9))
Sei $t\colon \mathbb{N} \to \mathbb{N}$. *Die Komplexitätsklasse* NTIME(t) *besteht aus allen Sprachen A, für die es eine Mehrband-NTM gibt, die A entscheidet und in Zeit* $O(t)$ *arbeitet.*

Definition 6 (P [MV15])
Sei $n\colon \mathbb{N} \to \mathbb{N}$. *Die Komplexitätsklasse* P *besteht aus allen Sprachen A, für die gilt* $A \in \mathrm{TIME}(\mathrm{n}^{\mathrm{O}(1)})$.

P ist somit die Klasse der effizient lösbaren Probleme. Das bedeutet, es gibt einen Algorithmus, der dieses Problem in polynomieller Zeit lösen kann.

Definition 7 (NP [MV15])
Sei $n\colon \mathbb{N} \to \mathbb{N}$. *Die Komplexitätsklasse* NP *besteht aus allen Sprachen A, für die gilt* $A \in \mathrm{NTIME}(\mathrm{n}^{\mathrm{O}(1)})$.

NP ist somit die Klasse der effizient überprüfbaren Probleme. Sowohl SAT als auch 3SAT liegen in der Klasse NP [MV15]. Aufgrund dessen sind beide Probleme effizient überprüfbar und liegen in der Klasse NTIME($n^{O(1)}$). Dabei ist überprüfbar wie folgt definiert:

Definition 8 (Polynomielle U̇berprüfbarkeit ([MV15] Def. 13))
Eine Sprache A ist polynomiell-überprüfbar, falls es einen Algorithmus V (wie englisch Verifier - Überprüfer) gibt, sodass für alle Eingaben w gilt:

$$w \in A \text{ gdw. es gibt ein Wort } x, \text{ sodass } V \text{ auf } \langle w, x \rangle \text{ aktzeptiert.}$$

Die Laufzeit von V bei Eingabe $\langle w, x \rangle$ *ist dabei durch ein Polynom in* $\mid w \mid$ *beschränkt.*

Ein umstrittenes und ungelöstes Problem ist, ob P = NP oder P ≠ NP gilt. In einer Umfrage von William I. Gasarch aus dem Jahr 2018 [Gas19] wurden 126 Theoretiker zu diesem Thema befragt. 88% von ihnen sind der Meinung, dass P ≠ NP. Auf die Frage, wann man einen Beweis dafür erwartet, antworteten 0%, dass es in den nächsten 19 Jahren passieren wird. 34% der Befragten erwarten einen Beweis erst nach dem Jahr 2100 und 66% rechnen mit einem Beweis vor dem Jahre 2100. Daraus lässt sich schließen, dass diese Frage wahrscheinlich noch einige Zeit ungeklärt bleiben wird.

Bei der Analyse eines Algorithmus spielt dessen Laufzeit eine entscheidende Rolle. Die Laufzeit hängt jedoch von dem verwendeten Rechner ab. Ein leistungsstarker Rechner aus dem Jahre 2020 wird einen aktuellen Algorithmus deutlich schneller durchlaufen können als ein Rechner aus dem Jahre 2000. Daher werden für den Vergleich von Laufzeiten häufig maschinenunabhängige Verfahren verwendet. Um die Laufzeit eines Algorithmus maschinenunabhängig überprüfen zu können, wird ein formales Verfahren benötigt. Hierfür wurde die *O*-Notation entwickelt. Durch diese wird das Wachstumsverhalten der Laufzeit beziehungsweise die Effizienz eines Algorithmus bestimmt.

Definition 9 (*O*-Notation ([MV15] Def. 5))
Seien $f, g\colon \mathbb{N} \to \mathbb{N}$ *zwei Funktionen. Dann ist* $f \in O(g)$, *falls es* $c, n_0 \in \mathbb{N}$ *gibt, sodass für alle* $n \geq n_0$ *gilt* $f(n) \geq c \cdot g(n)$. *Wir schreiben dann auch:* $f(n) \in O(g(n))$. *Das heißt, f wächst höchstens so stark wie g.*

Konstanten sind bei unendlich großen Werten für n vernachlässigbar, das heißt $3n \in O(n)$ oder $10n^2 \in O(n^2)$. Das Wachstumsverhalten der Laufzeit lässt sich in verschiedene Klassen einordnen, dargestellt in Tabelle 2.1.

O-Notation	Klasse
$O(1)$	konstant
$O(log(n))$	logarithmisch
$O(n)$	linear
$O(n^2)$	quadratisch
$O(n^k), k > 2$	polynomiell
$O(c^n), c > 1$	exponentiell
$O(n!)$	faktoriell

Tabelle 2.1.: Klassen der O-Notation

2.2. Stochastische Grundlagen

Ein Random Walk ist ein stochastischer Prozess, weshalb einige Grundlagen in dem Bereich der Stochastik notwendig sind. Die entsprechende Grundlagen werden im Folgenden kurz aufgeführt und erläutert.

Definition 10 (Zufallsexperiment (vergl. [Hen17] S. 1))
Ein ideales Zufallsexperiment ist ein Experiment mit genau festgelegten Bedingungen, für welches alle möglichen Ergebnisse bekannt sind und welches beliebig oft unter gleichen Bedingungen wiederholbar ist.

Die möglichen Ausgänge eines solchen Zufallsexperimentes werden Ergebnisse genannt, die Menge aller dieser Ergebnisse ist die Ergebnismenge Ω.

Definition 11 ($\sigma - Algebra$ (vergl. [Hen17] S. 287))
Ω ist eine nichtleere Menge und $\mathcal{A}$ ein System von Teilmengen von Ω mit folgenden Eigenschaften:

- $\Omega \in \mathcal{A}$
- $A \in \mathcal{A} \Rightarrow A^C \in \mathcal{A}$
- $A_1, A_2, A_3, \cdots \in \mathcal{A} \Rightarrow \bigcup_{n=1}^{\infty} A_n \in \mathcal{A}$

Ein Mengensystem mit diesen Eigenschaften nennt man $\sigma-Algebra$. Die Elemente $A \in \mathcal{A}$ heißen Ereignisse.

Definition 12 (Wahrscheinlichkeitsmaß ([Hen17] Abschnitt 31.1))
Sei $\mathbb{P}$ eine auf $\mathcal{A}$ definierte reellwertige Funktion mit folgenden Eigenschaften:

a) $\mathbb{P}(A) \geq 0$ *für* $A \in \mathcal{A}$,

b) $\mathbb{P}(\Omega) = 1$,

c) $\mathbb{P}(\sum_{j=1}^{\infty} A_j) = \sum_{j=1}^{\infty} \mathbb{P}(A_j)$, *falls* $A_1, A_2, \cdots$ *disjunkte Mengen aus* $\mathcal{A}$ *sind.*

$\mathbb{P}$ heißt Wahrscheinlichkeitsmaß oder auch Wahrscheinlichkeitsverteilung auf $\mathcal{A}$.

Das Wahrscheinlichkeitsmaß $\mathbb{P}(x)$ für einen g-regulären ungerichteten Graphen $G = (V, E)$ ist durch $\mathbb{P}(x) = \frac{1}{|V|}, \forall x \in V$ definiert [LPW08].

Definition 13 (Zufallsvariable ([Hen17] Def. 3.1))
Ist Ω ein Grundraum, so heißt jede Abbildung $X : \Omega \to \mathbb{R}$ von Ω in die Menge $\mathbb{R}$ der reellen Zahlen eine Zufallsvariable (engl.: random variable) (auf Ω).

Ein Ereignis ω wird vom Zufall erzeugt. Die zugehörige Zufallsvariable X zu diesem Ereignis wird durch $X(\omega)$ dargestellt. Ein Beispiel hierfür wäre ein Wurf mit zwei Würfeln, welche jeweils 6 Seiten besitzen. Ω würde in diesem Fall alle möglichen Ergebnisse eines Wurfes darstellen, also $\Omega = \{1,2,3,4,5,6\} \times \{1,2,3,4,5,6\}$. $X\colon \Omega \to \{2,\cdots,12\}$ ordnet einem Wurf $\omega = (i,j)$ durch $X((i,j)) \coloneqq i+j$ eine Augensumme zu. In diesem Fall wäre $i+j$ die zum Ergebnis ω zugehörige Zufallsvariable. [Beh13]

Markov-Ketten und Markov-Prozesse

Markov-Prozesse verallgemeinern die Idee von Markov-Ketten. Ein Markov-Prozess ist ein stochastischer Prozess mit einem abzählbaren Zustandsraum. Die Grundidee dieser Prozesse besagt, dass das, was in der Zukunft passiert, nicht von der Vergangenheit abhängt, sondern lediglich von der Gegenwart. Anders ausgedrückt ist eine Zufallsvariable X ein Markov-Prozess, wenn die Übergangswahrscheinlichkeit vom Zustand U_t zum Zeitpunkt t in einen anderen Zustand U_{t+1} nur vom aktuellen Zustand U_i abhängt. [Yan10]

Eine Markov-Kette ist eine von einem Markov-Prozess erzeugte Folge von Zufallsvariablen $F_X = (X_1, X_2, ..., X_n)$, wobei X_n jeweils eine Zufallsvariable ist.

2.3. Algebraische Grundlagen

In einigen Random-Walk-Algorithmen sind Berechnungen aus dem Bereich der Linearen Algebra notwendig, weshalb in diesem Bereich die notwendigen Definitionen ebenfalls noch einmal aufgegriffen und wiederholt werden.

Definition 14 (Eigenwert (vergl. [Str03] S. 247))
Eine Zahl λ ist ein Eigenwert von A dann und nur dann, wenn $det(A - \lambda I) = 0$ gilt, wobei A eine Matrix ist und I die Einheitsmatrix

Definition 15 (Maximaler Eigenwert [LPW08])
$\lambda_* = max\{\mid \lambda \mid : \lambda$ *ist Eigenwert von Übergangsmatrix* P *und* $\lambda \neq 1\}$

Definition 16 ((Absolute) spektrale Lücke (vergl. [LPW08] S. 154))
Die spektrale Lücke γ ist durch $\gamma = 1 - \lambda_2$ definiert, wobei $\lambda_2 \leq \lambda_*$ der zweitgrößte Eigenwert ist. Die absolute spektrale Lücke ist durch $\gamma_* = 1 - \lambda_*$ definiert.

Definition 17 (relaxation time (vergl. [LPW08] S. 155))
Die relaxation time ist für eine Übergangsmatrix eines g-regulären ungerichteten Graphen mit absoluter spektraler Lücke γ_* wie folgt definiert $t_{rel} = \frac{1}{\gamma_*}$.

2.4. Einstieg in Python

Python ist aktuell die am stärksten wachsende Programmiersprache und ist gerade für Einsteiger sehr gut geeignet [Sri17]. Python-Programme laufen auf fast allen gängigen Betriebssystemen für Computer und sogar teilweise auf Android [IW18]. Dies ist auch der Grund, weshalb in dieser Bachelorarbeit für die Verdeutlichung von Schwierigkeiten bei der Implementierung der Algorithmen Python verwendet wird. Im Folgenden werden einige Grundlagen und verwendete Erweiterungen erklärt. Die folgenden Beispiele sind auf Grundlage von [pyt] erstellt worden.

Typdeklaration in Python

Im Gegensatz zu den meisten gängigen Programmiersprachen ist in Python keine Typdeklaration notwendig, sondern dies wird automatisch von dem Compiler erledigt. Die Zuweisung von Werten auf Variablen sieht daher wie folgt aus:

```
1  i = 42
2  s = 'Hallo'
```

Die automatische Typdeklaration kann schnell zu Fehlern führen, da ständig im Hinterkopf behalten werden muss, wie genau die Variable durch den Compiler deklariert wird. Es ist allerdings auch sehr praktisch für den Programmierer, der sich nicht mehr mit der Deklaration beschäftigen muss.

Ausgabe in Python

Die Ausgabe in Python erfolgt durch die print()-Funktion, welche seit Python 3 mit Klammern genutzt werden muss. Eine Ausgabe könnte wie folgt erzeugt werden:

```
1  hello = 42
2  print("hello")
3  print(hello)
```

Dieser Code würde auf dem Terminal folgende Ausgabe erzeugen: *hello* 42. Bei der Verwendung von Anführungszeichen wird der übergebene Parameter als String interpretiert und anschließend ausgegeben (Zeile 2). Sollten keine Anführungszeichen verwendet werden, so wird der übergebene Parameter als Variable interpretiert und die print()-Funktion gibt den Wert dieser Variable aus (Zeile 3).

Schleifen in Python

Durch eine Schleife wird ein bestimmter Teil eines Programmes solange ausgeführt, bis die Schleife gestoppt wird. Die Anweisung ist gerade in solchen Situationen hilfreich, in denen ein bestimmter Teil so lange wiederholt werden muss, bis ein bestimmtes Ereignis eintritt.

In Python nutzt man while- oder for-Schleifen. While-Schleifen werden so lange durchlaufen, bis eine angegebene Bedingung nicht mehr gilt. Die for-Schleife kann verwendet werden, um über Werte zu iterieren. Zur Hilfe kann zusätzlich die range-Funktion genutzt werden. Ein Beispiel wäre:

```
1  for i in range(1,4):
2      print(i)
```

Es ensteht die Ausgabe: 1 2 3. Die Schleife startet bei $i = 1$ und erhöht nach jedem print den Wert von i um 1. Sobald $i = 4$ gilt, bricht die Schleife ab.

if und else in Python

Der Code nach einer if-Anweisung wird nur ausgeführt, falls die Bedingung, die dem if beiliegt, erfüllt ist. Sollte die Bedingung nicht erfüllt sein, so wird der Code, der auf else folgt, ausgeführt. Da in einigen Situationen if und else nicht ausreichen, existiert noch elif. Elif folgt auf ein if und prüft erneut eine Bedingung, falls die Bedingung des if nicht erfüllt wurde. Ein Beispiel hierfür wäre:

```
1 if awake:
2     eat()
3 elif sleep:
4     dream()
5 else:
6     repeat()
```

Bibliotheken in Python

Bibliotheken werden in Python verwendet, um Code auszulagern. Dies macht die Nutzung deutlich übersichtlicher. Bibliotheken werden auch Erweiterungen genannt, da viele vorhandene Bibliotheken genutzt werden können, um verschiedene Funktionen und Klassen zu verwenden, welche andere Nutzer bereits implementiert haben. In dieser Arbeit werden die folgenden Bibliotheken verwendet:

Bibliothek	Erklärung
random	Pseudo-Zufallszahlen Generator
numpy	effiziente Operationen mit Arrays von numerischen Daten
math	verschiedene mathematische Funktionen
scipy.linalg	für zusätzliche Funktionen aus dem Bereich der linearen Algebra
time	stellt Funktionen bezüglich der Zeit zur Verfügung, zum Beispiel zum Warten für eine bestimmte Zeit
threading	zur Parallelisierung von verschiedenen Prozessen
queue	modulübergreifende Speicherwarteschlange, zum Beispiel für die Kommunikation zwischen Threads
pylab	für die übersichtliche Darstellung von Graphen
sys	bietet Zugriff auf Variablen, welche vom Interpreter verwendet oder entgegengenommen werden

Tabelle 2.2.: Verwendete Python-Bibliotheken

Bibliotheken werden am Anfang des Codes durch *import libaryname* eingebunden und können daraufhin vollständig verwendet werden.

Aufbau eines Python-Programmes

Ein Python-Programm benötigt nicht zwingend eine Main-Funktion. Bei größeren Programmen ist die Main-Funktion aufgrund der Übersichtlichkeit dennoch empfohlen. Die Definition von Klassen und Funktionen ist ähnlich wie in anderen Programmiersprachen möglich. In Python ist die Einrückung der Codezeilen ausschlaggebend für die Zugehörigkeit einer Codezeile. Zur Veranschaulichung dient das folgende Beispiel:

```
1 class TwoValues:
2     def __init__(self, first, second):
3         self.first = first
4         self.second = second
5     def add(self):
6         return self.first + self.second
7
```

```
 8  if __name__ == "__main__":
 9      x = TwoValues(1,2)
10      print(x.add())
```

Das Programm würde eine Klasse TwoValues erstellen, in der zwei beliebige Werte gespeichert werden können. Bei dem Aufruf TwoValues.add() werden die zwei Werte dann addiert und zurückgegeben. Im vorliegenden Fall werden 1 und 2 als TwoValues in x gespeichert. Danach wird die Addition der beiden Werte (3) ausgegeben.

Queue in Python

Queues, auch Warteschlangen genannt, sind Speicherketten. In Queues können durch die Funktion put() Werte geschrieben werden. Außerdem können durch die Funktion get() diese Werte auch wieder aus der Queue gelesen werden. Wenn ein Wert aus der Queue gelesen wird, so wird dieser entfernt. Der zuerst hinein geschriebene Wert ist auch der, der zuerst gelesen wird (First-in-First-out).

Threads in Python

Durch Threads können Prozesse pseudo-parallel ausgeführt werden, was notwendig ist, sobald mehrere verschiedene Programmteile zeitgleich ausgeführt werden sollen. Threads können außerdem über eine Queue miteinander kommunizieren.

2.5. Random Walk in $\mathbb{R}$

Ein Random Walk in $\mathbb{R}$ (RWiR) ist ein stochastischer Prozess, um eine Bewegung zu simulieren. Bei dieser Bewegung erfolgen die Schritte in eine zufällige Richtung [SS02]. Ein Random Walk wird in der deutschen Sprache auch häufig Irrfahrt genannt [Hen18].

Sei $X = (x_1, x_2, ..., x_n)$ mit $X \in \mathbb{R}^d$ eine Sequenz von Zufallsvariablen. Dann heißt der definierte stochastische Prozess

$$S_n = S_0 + \sum_{k=1}^{n} x_k$$

Random Walk beziehungsweise Irrfahrt mit den Schritten $x_1, x_2, ..., x_n$ in $\mathbb{R}^d$, wobei S_0 der Startpunkt ist. [Ber97]

2.5.1. 1D Random Walk

Ein Random Walk in der ersten Dimension (1D-RW) wird auch einfache Irrfahrt genannt. Der 1D-RW dient der Einführung in die Random-Walk-Algorithmen. Dieser kann beispielsweise für die Simulation von kleinen Spielen, wie etwa einem wiederholten Münzwurf, genutzt werden. Hierbei stellt Zahl einen Schritt in die negative Richtung und Kopf einen Schritt in die positive Richtung dar [MS09, Yan10]. Endet der 1D-RW auf einem negativen Feld, so hat der Spieler gewonnen, der Zahl gewählt hat und bei einem positiven Feld hat der Spieler gewonnen, der Kopf gewählt hat. Zusätzlich dient der 1D-RW als Standardbeispiel für stochastische Prozesse, da er einfach zu behandeln ist und viele typische Phänomene eines stochastischen Prozesses aufweist.

Funktionsweise

Die Funktionsweise wird anhand eines Beispiels erläutert. Im vorliegenden Beispiel ist $X = (x_1, x_2, ..., x_n)$ mit $X \in \mathbb{R}^1$. Die Schrittlänge s wird hierbei auf $s = 1$ festgelegt. Somit muss $x_i \in [-1, 1]$ gelten, $i \in [1, \cdots, n]$. Die Wahrscheinlichkeiten sind gleich verteilt, da die einfache symmetrische Irrfahrt betrachtet wird. Daher gilt $\mathbb{P}(x_n = -1) = 1/2 = \mathbb{P}(x_n = 1)$. [Hen18] Für den Startpunkt gilt $S_0 = 0$. Ein möglicher Ablauf für einen solchen Random Walk mit $n = 5$ Schritten ist in Abbildung 2.1 dargestellt. Es sind auch noch viele weitere Ergebnisse möglich. Um genau zu sein gibt es 2^n verschiedene Lösungspfade mit $n + 1$ verschiedenen Endpunkten (Abbildung 2.2). Folglich gibt es in diesem Beispiel $2^5 = 32$ verschiedene Lösungspfade mit $5 + 1 = 6$ verschiedenen Endpunkten.

In der Abbildung 2.1 wurde der Ausgangspunkt (0) mehrmals besucht. Die Wahrscheinlichkeit, dass der Walker beim 1D-RW nach genau $2n$ Schritten zum Ausgangspunkt zurückkehrt, lässt sich durch $\mathbb{P}(W = 2n) = \frac{\binom{2\cdot(n-1)}{n-1}}{2^{2\cdot(n-1)}} \cdot \frac{1}{2n}$ bestimmen, wobei $n \geq 1$. Die $2n$ Schritte sind darauf zurückzuführen, dass ein Walker nur nach einer geraden Anzahl an Schritten zurück zum Ausgangspunkt gelangen kann. Die Wahrscheinlichkeit, dass er innerhalb von $2n$ Schritten mindestens ein Mal zurückgekehrt ist, beträgt $\mathbb{P}(W \leq 2n) = \sum_{i=1}^{n} \mathbb{P}(W = 2i)$. Diese Summe konvergiert für $n \to \infty$ gegen 1. Daher liegt die Wahrscheinlichkeit, dass ein Walker irgendwann den Ausgangspunkt erneut besucht, bei 100% [Hen18]. Bereits bei dem genannten Beispiel mit $2n = 5$ Schritten beträgt diese Wahrscheinlichkeit $\mathbb{P}(W \leq 2 \cdot 2.5) = \frac{5}{8}$, also 62.5%.

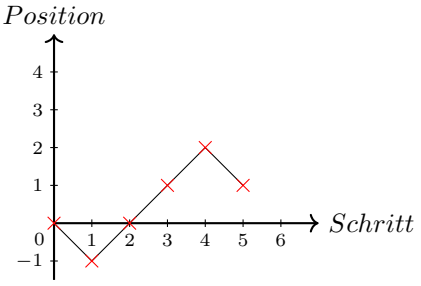


Abbildung 2.1.: Einfache symmetrische Irrfahrt

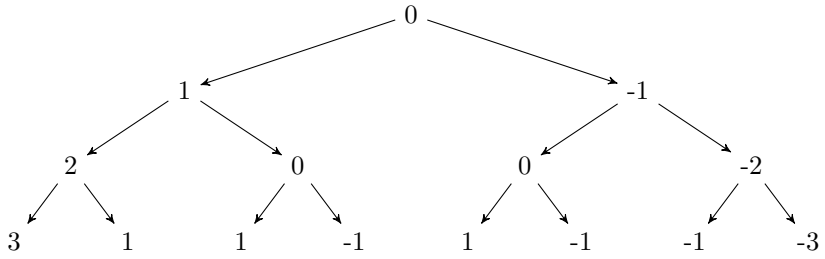


Abbildung 2.2.: Baumdiagramm einfache symmetrische Irrfahrt

Um eine Irrfahrt übersichtlich darstellen zu können, benutzt man häufig Diagramme, bei denen auf der x-Achse die Schritte zu finden sind und auf der y-Achse die aktuelle Position der Irrfahrt, wie zum Beispiel in Abbildung 2.1 oder in Abbildung 2.3.

Pseudocode

Der folgende Pseudocode beschreibt die einfache symmetrische Irrfahrt in $\mathbb{R}$. Die Abbildung 2.3 ist eine mögliche Ausgabe RW des Algorithmus für $n = 100$.

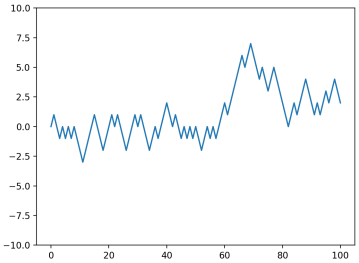


Abbildung 2.3.: Ausgabe des 1D-RW Algorithmus

Algorithmus 1 Einfache symmetrische Irrfahrt in $\mathbb{R}$

Eingabe: Startpunkt $s \in \mathbb{R}$, Schrittanzahl $n \geq 0$
Ausgabe: Random Walk RW

1: RW ist eine Liste der Länge $n + 1$
2: $RW[0] \leftarrow s$
3: $i \leftarrow 0$
4: **while** $i < n$ **do**
5: $\quad s \leftarrow randomNumber[-1, 1]$
6: $\quad RW[i + 1] \leftarrow RW[i] + s$
7: $\quad i \leftarrow i + 1$
8: **end while**
9: **return** RW

2.5.2. 2D Random Walk

Ein Random Walk in der zweiten Dimension (2D-RW) findet in der Realität schon deutlich häufiger Anwendung. Wenn er leicht modifiziert wird, kann dieser zur Simulation von Polymer-Ketten eingesetzt werden [DGKG65].

Funktionsweise

Die Funktionsweise wird anhand eines Beispiels erläutert. In diesem Beispiel ist $X = (x_1, x_2, ..., x_n) \in \mathbb{R}^2$. Die Schrittlänge s ist auf $s = 1$ festgelegt. Somit muss $x_i \in [(0, 1), (1, 0), (0, -1), (-1, 0)]$ gelten. Die Wahrscheinlichkeiten sind gleich verteilt, also gilt $\mathbb{P}(x_n = (0, 1)) = \mathbb{P}(x_n = (1, 0)) = \mathbb{P}(x_n = (0, -1)) = \mathbb{P}(x_n = (-1, 0)) = 1/4$ [Hen18]. Für den Startpunkt gilt $S_0 = (0, 0)$. Ein mögliches Ergebnis für einen solchen Random Walk mit $n = 10$ Schritten ist in Abbildung 2.5 dargestellt. Es sind auch noch viele weitere Lösungen möglich. Um genau zu sein gibt es $(2 \cdot 2)^n = 4^n$ verschiedene Lösungspfade mit $(n + 1)^2$ verschiedenen Endpunkten (Abbildung 2.4). Folglich sind in dem vorliegenden Beispiel $4^{10} = 1048576$ verschiedene Lösungspfade mit $(10 + 1)^2 = 121$ verschiedenen Endpunkten möglich.

In Abbildung 2.5 wurde der Ausgangspunkt (0,0) nicht erneut besucht. Anders verhält es sich in Abbildung 2.6, wo der Ausgangspunkt (0,0) mehrmals besucht wurde. Die Wahrscheinlichkeit, dass ein Walker nach $2n$ Schritten mindestens einmal zum Ausgangspunkt zurückkehrt, ist durch $\mathbb{P}(W \leq 2n) = 1 - \frac{1}{\sum_{i=0}^{n} \frac{1}{4^{2i}} \cdot \binom{2i}{i}^2}$ gegeben. Diese Summe konvergiert für $n \to \infty$ gegen 1, was bedeutet, dass der 2D-RW bei einer großen Anzahl an Schritten zu circa 100% den Ausgangspunkt erneut besucht. [Joh11]

Für das oben aufgeführte Beispiel mit $10 = 2n$ Schritten liegt die Rückkehrwahrscheinlichkeit bei $\mathbb{P}(W \leq 2 \cdot 5) = 0.3841$, also 38.41%.

Bezüglich der Frage aus der Einleitung, wie wahrscheinlich es ist, in einer unbekannten Stadt, in welcher die Straßen im Schachbrettmuster angeordnet sind, irgendwann wieder zurück zum Ausgangspunkt zu gelangen, wenn man an jeder Kreuzung zufällig abbiegt, zurück oder geradeaus geht, lässt sich nun sagen, dass diese Wahrscheinlichkeit, sofern eine sehr hohe Anzahl an Schritten zurückgelegt wurde, fast 100% beträgt. Bereits nach 40 Schritten liegt die Wahrscheinlichkeit bei circa 50%.

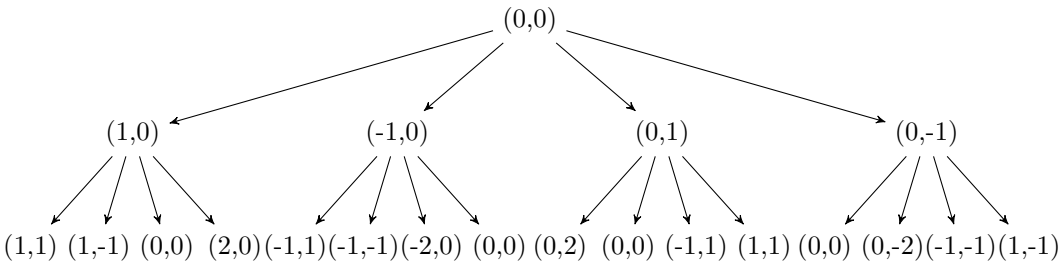


Abbildung 2.4.: Baumdiagramm 2D Random Walk

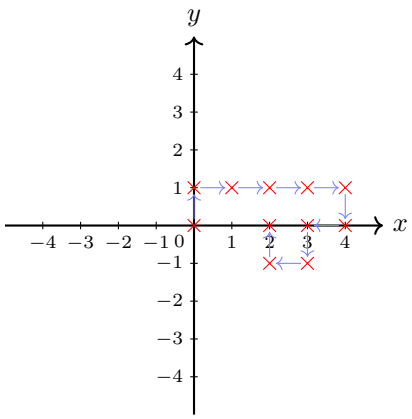


Abbildung 2.5.: 2D Random Walk mit 10 Schritten

Pseudocode

Der Pseudocode des 2D-RW entspricht ungefähr dem des 1D-RW aus Abschnitt 2.5.1, jedoch mit der kleinen Abänderung, dass in der Zeile 5 statt $s \leftarrow randomNumber[-1, 1]$

nun $s \leftarrow randomPoint[(0,1),(0,-1),(1,0),(-1,0)]$ steht. Daher wurde davon abgesehen, diesen Pseudocode aufzuführen. In Abbildung 2.6 ist eine mögliche Ausgabe eines 2D-RW mit 10.000 Schritten dargestellt.

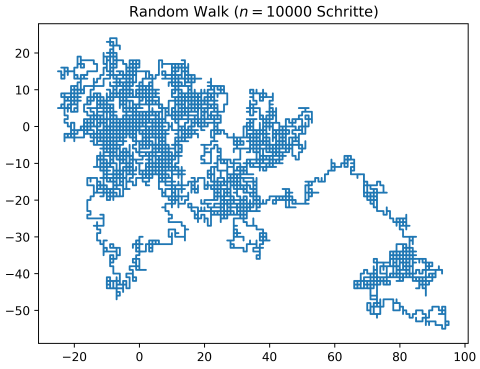


Abbildung 2.6.: Ausgabe des 2D-RW Algorithmus

2.5.3. Vergleich zwischen 1D und 2D Random Walk

In diesem Vergleich werden die Unterschiede und Gemeinsamkeiten des 1D-RW und des 2D-RW in Bezug auf die Funktionsweise, die Einsatzgebiete und die Laufzeiten herausgearbeitet.

Funktionsweise

Wird die Funktionsweise der beiden Algorithmen verglichen, so lässt sich feststellen, dass die Unterschiede sehr gering sind. Der größte Unterschied ist hier die Dimension, in der der jeweilige Algorithmus arbeitet. Daher ist ein weiterer Unterschied auch die Darstellung des Ergebnisses. Beim 1D-RW steht die x-Achse für den jeweiligen Schritt und die y-Achse für die Position des Walkers. Anders ist es beim 2D-RW. Dort wird die Position durch die Betrachtung beider Achsen bestimmt.

Weitere Unterschiede sind die Lösungsmenge und die Anzahl unterschiedlicher Endpunkte nach n Schritten. Beim 1D-RW gibt es 2^n verschiedene Lösungswege und $n+1$ verschiedene Endpunkte. Der 2D-RW hingegen hat quadratisch viele verschiedene Lösungswege mehr, nämlich $(2^n)^2 = 2^{2n} = 4^n$. Außerdem gibt es quadratisch viele verschiedene Endpunkte mehr, $(n+1)^2$.

Dies könnte die Vermutung aufkommen lassen, dass die Anzahl der verschiedenen Lösungswege sowie die Anzahl der verschiedenen Endpunkte wie folgt im Zusammenhang zur Dimension d stehen: $(2^n)^d$ verschiedene Lösungswege und $(n+1)^d$ verschiedene Endpunkte.

Diese Vermutung lässt sich jedoch mit einem Gegenbeweis widerlegen. Dazu wird der Random Walk der dritten Dimension (3D-RW) genutzt, der in der Funktionsweise ebenfalls sehr ähnlich ist und lediglich wieder zwei weitere Richtungen hinzufügt, nämlich $\pm s$ auf der z-Achse. Werden nun die verschiedenen möglichen Lösungswege betrachtet, ergibt sich die Formel $6^n \neq (2^n)^3 = 2^{3n} = 8^n$. Hinsichtlich der verschiedenen Endpunkte gibt es beim 3D-RW $\frac{2\cdot(n+1)^3+(n+1)}{3} \neq (n+1)^3$ Möglichkeiten. Durch diesen Gegenbeweis wurde diese Vermutung widerlegt.

Tatsächlich lässt sich die Anzahl der verschiedenen Lösungswege mit Hilfe der folgenden Definitionen und des folgenden Theorems herleiten.

Definition 18 (*k*-ary Baum [CLRS01])
Ein k-ary Baum ist ein Baum, bei dem jeder Knoten maximal k Kinder besitzt.

Definition 19 (Vollständiger *k*-ary Baum [CLRS01])
Ein vollständiger k-ary Baum ist ein k-ary Baum, bei dem jeder Knoten, der kein Blatt ist, genau k Kinder besitzt.

Theorem 1 (*k*-ary Baum Anzahl der Blätter [CLRS01])
Ein vollständiger k-ary Baum hat in der Ebene h genau k^h Blätter.

Ein Random Walk in $\mathbb{R}^d$ kann als vollständiger k-ary Baum dargestellt werden, wobei $k = 2 \cdot d$. Zu jedem Endpunkt in einem k-ary Baum führt genau ein Weg. Daraus lässt sich schließen, dass ein Random Walk in $\mathbb{R}^d$ genau $(2 \cdot d)^h$ verschiedene Lösungswege besitzt, wobei h die Anzahl der Schritte widerspiegelt.

Für die Anzahl der verschiedenen Endpunkte lässt sich auch nach ausführlicher Recherche kein Zusammenhang in Bezug auf die Dimension ausfindig machen.

Die Gemeinsamkeiten der Algorithmen sind offensichtlich und daher schnell genannt. Eine Gemeinsamkeit ist die Zufälligkeit der Schrittrichtung und die feste Schrittlänge. Eine Weitere sind die Startpunkte der Algorithmen, denn beide starten im Normalfall im Nullpunkt ihrer Dimension.

Einsatzgebiet

Die Einsatzgebiete beider Algorithmen unterscheiden sich deutlich stärker. Während der 1D-RW hauptsächlich für die Einführung in die Random Walk Algorithmen und als Standardbeispiel für stochastische Prozesse dient, findet der 2D-RW häufig Anwendung für die Simulation/Modellierung der Realität. Ein Beispiel hierfür wurde bereits in Abschnitt 2.5.2 genannt.

Laufzeit

Für den Pseudocode des 1D-RW (Algorithmus 2.3) ergibt sich die folgende Laufzeit:

$$O(1) + O(1) + O(n) \cdot (O(1) + O(1) + O(1))$$
$$= O(n)$$

Wobei $O(n)$ die Schleife widerspiegelt und $O(1)$ jeweils eine Zuweisung. Algorithmus 1 hat also eine Laufzeit von $O(n)$.

Für den 2D-RW ergibt sich ebenfalls eine Laufzeit von $O(n)$, da lediglich eine Zeile, welche einer Zuweisung entspricht ($O(1)$), angepasst werden muss. Daher ändert sich die Laufzeit nicht.

Folglich unterscheiden sich die beiden Algorithmen im worst-case in ihrer Laufzeit nicht.

2.5.4. Random Walk in $\mathbb{R}$ als Markov Kette

Der Übergang eines RWiR ist ein Markov-Prozess, da der Übergang von einem Punkt zum nächsten Punkt nur vom aktuellen Punkt abhängt. Es ist irrelevant, welche Punkte der Walker vorher besucht hat. Die Wahrscheinlichkeiten dafür, welcher Punkt als nächstes besucht wird, sind nur davon abhängig, an welchem Punkt sich der Walker jetzt gerade befindet.

Wenn nun der gesamte RWiR betrachtet wird, ist dieser eine Markov-Kette, da ein Übergang des RWiR ein Markov-Prozess ist und der gesamte RWiR eine Folge von Übergängen darstellt. [Yan10]

3. Self-Avoiding Walk

Die in diesem Kapitel folgenden Ausführungen des Self-Avoiding Walks beziehen sich auf [SM13]. Ein Self-Avoiding Walk (SAW) ist ein Random Walk, bei dem kein Punkt doppelt besucht werden darf. Um einen bereits besuchten Knoten kenntlich zu machen, wird dieser markiert. Sobald alle Punkte markiert wurden oder der aktuelle Punkt nur von markierten Punkten umgeben ist, ist kein Schritt mehr möglich. Der SAW terminiert, sobald kein Schritt mehr möglich ist oder sobald er die gewünschte Anzahl an Schritten zurückgelegt hat. SAWs können auf (begrenzten) Gittern in allen möglichen Dimension oder auf Graphen modelliert werden. Sie sind ein sehr bekanntes Mittel zur Modellierung von linearen Polymer-Ketten.

Definition 20 (Self-Avoiding Walk (vergl. [SM13]))
Sei $X = (x_1, x_2, \ldots, x_n)$ *mit* $X \in \mathbb{Z}^d$ *eine Sequenz von Zufallsvariablen, mit* $S_0 + \sum_{k=1}^{i} x_k \neq S_0 + \sum_{k=1}^{j} x_k$, *wobei* $i, j \in [0, \ldots, n]$, $i \neq j$ *und* S_0 *der Startpunkt. Dann ist* X *ein SAW in der Dimension* d.

3.1. Funktionsweise

Um die Funktionsweise deutlich zu machen, wird ein erklärendes Beispiel in der zweiten Dimension mit der Schrittlänge $s = 1$ verwendet. $X = (x_1, \ldots, x_n) \in \mathbb{Z}^2$ und somit ist $x_i \in [(0,1), (1,0), (0,-1), (-1,0)]$. Die Wahrscheinlichkeiten für die einzelnen Richtungen sind jeweils gleich verteilt. Daher gilt $\mathbb{P}(x_n = (0,1)) = \mathbb{P}(x_n = (1,0)) = \mathbb{P}(x_n = (0,-1)) = \mathbb{P}(x_n = (-1,0)) = 1/4$. Sollte aber beispielsweise $x_n = (0,1)$ zu einem bereits besuchten Punkt führen, so würde $\mathbb{P}(x_n = (0,1)) = 0$ und $\mathbb{P}(x_n = (1,0)) = \mathbb{P}(x_n = (0,-1)) = \mathbb{P}(x_n = (-1,0)) = 1/3$ werden. Dies bedeutet allgemein, dass jede Richtung x, die zu einem unbesuchten Knoten führt, die Wahrscheinlichkeit $\mathbb{P}(x) = \frac{1}{4-|besuchteNachbarn|}$ hat. Sollten alle Wahrscheinlichkeiten $\mathbb{P}(x_n \in [(1,0), (-1,0), (0,1), (0,-1)]) = 0$ sein, so terminiert der Algorithmus. Der Startpunkt S_0 ist in diesem Beispiel $S_0 = (0,0)$. Als begrenztes Gitter wird ein Gitter der Größe $3x3$ gewählt. Ein solcher SAW mit fünf Schritten ist in Abbildung 3.1 dargestellt. Die Anzahl der möglichen verschiedenen SAWs lässt sich nur für kleine Gitter und für eine kleine Anzahl von Schritten n sicher berechnen.

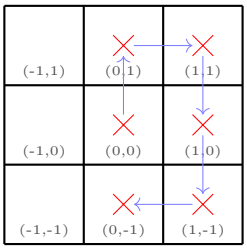


Abbildung 3.1.: SAW auf 3x3 Gitter mit 5 Schritten

Ein SAW in einer höheren Dimension $d > 2$ funktioniert analog zu diesem Beispiel. Für den Walker ändert sich lediglich die Anzahl an möglichen Richtungen. Pro Dimension

werden zwei Richtungen hinzugefügt, beziehungsweise es gibt immer $2 \cdot d$ mögliche Richtungen. Dementsprechend ändert sich auch die zugehörige Wahrscheinlichkeit $\mathbb{P}$ für die jeweiligen Richtungen, $\mathbb{P}(x) = \frac{1}{2d}$. Sofern bereits besuchte Nachbarn existieren, ist $\mathbb{P}(x) = \frac{1}{2d - |besuchteNachbarn|}$.

3.2. Pseudocode

Dieser Pseudocode beschreibt einen Self-Avoiding Walk der zweiten Dimension (2D-SAW). Die Abbildung 3.2 ist eine mögliche Ausgabe RW dieses Algorithmus für die Schrittzahl $n = 1000$.

Algorithmus 2 Self-Avoiding Walk auf $\mathbb{Z}^2$

Eingabe: Startpunkt $s \in \mathbb{Z}^2$, Schrittanzahl $n >= 0$
Ausgabe: Random Walk RW

1: RW ist eine Liste der Länge $n + 1$
2: $RW[0] \leftarrow s$
3: markiere s
4: $i \leftarrow 0$
5: **while** $i < n$ **do**
6: $possiblePoints \leftarrow$ jede Richtung $\in [(0,1),(0,-1),(1,0),(-1,0)]$ die zu einem unmarkierten Knoten führt
7: **if** $possiblePoint = \varnothing$ **then**
8: **return** RW
9: **end if**
10: $s \leftarrow randomPoint(possiblePoints)$
11: $RW[i+1] \leftarrow RW[i] + s$
12: markiere s
13: $i \leftarrow i + 1$
14: **end while**
15: **return** RW

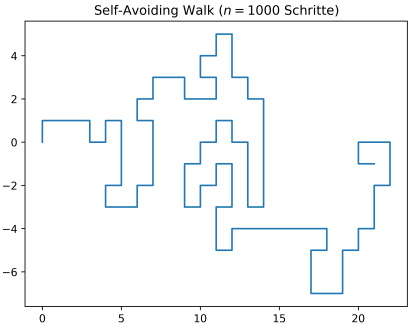


Abbildung 3.2.: Ausgabe des 2D-SAW Algorithmus

3.3. Spezialfall: Erste Dimension

Ein Self-Avoiding Walk der ersten Dimension (1D-SAW) kann als Spezialfall betrachtet werden. Der 1D-SAW kann im Gegensatz zu den SAWs in höheren Dimensionen nicht vorzeitig terminieren, da immer ein möglicher Weg existiert. Bei dem 1D-SAW ist nur der erste Schritt zufällig. Ein kleines Beispiel zeigt den Grund dafür:

Beispiel 1
Der Startpunkt S_0 befindet sich am Nullpunkt, also $S_0 = 0$. Die möglichen Richtungen, welche der Walker wählen kann, sind $(+1, -1)$. Zuerst wird zufällig eine Richtung gewählt, z.B. -1. Nun befindet sich der Walker am Punkt (-1). Jetzt sind nur noch die Richtungen möglich, welche zu unmarkierten Knoten führen. Punkt (0) ist bereits markiert, weshalb die Richtung $(+1)$ nicht mehr möglich ist. Der Punkt (-2) ist unmarkiert, sodass die Richtung (-1) möglich ist. Dementsprechend ist $\mathbb{P}(-1) = \frac{1}{2-1} = 1$. Dies ändert sich im Laufe des Walks auch nicht mehr, da der Punkt in Richtung $(+1)$ immer im vorherigen Schritt markiert wurde.

Hat sich der Walker bei dem 1D-SAW erst einmal für eine Richtung entschieden, so kann er nur noch in diese Richtung r gehen. Ab diesem Moment sind die Schritte nicht mehr zufällig, da für $\mathbb{P}(r) = 1$ gilt. Dies hat zur Folge, dass der 1D-SAW in der Praxis keine Anwendung findet.

3.4. Modellierung von Polymer-Ketten

Am häufigsten findet der SAW Anwendung bei der Modellierung von linearen Polymer-Ketten. Im Folgenden wird grundlegend erklärt, was Polymer-Ketten sind und warum sich der SAW der dritten Dimension zur Modellierung eignet.

Ein Polymer ist ein Molekül, welches aus vielen Gruppen von Atomen besteht, auch genannt Monomere. Monomere sind durch eine chemische Bindung miteinander verbunden und haben eine Funktionalität, welche beschreibt, wie viele Verbindungen sie zu anderen Gruppen von Atomen eingehen werden. Beträgt innerhalb einer Verkettung bei jedem Monomer M die Funktionalität $f = 2$, so spricht man von einem linearen Polymer. Ausnahmen hierbei sind das erste und das letzte Monomer, für welche jeweils $f = 1$ gilt. Dies könnte dann wie folgt aussehen:

$$M - \cdots - M - M - M - M - M - M - M - \cdots - M$$

Solche Polymere können sehr lang werden. Um genau zu sein, kann ein Polymer mehr als 10^5 Monomere beinhalten. Ein Polymer breitet sich zufällig in der dritten Dimension aus. Daher bietet sich zur Simulation ein Random Walk in $\mathbb{R}^3$ an. Dieses Modell ist als Ideale Polymer-Kette bekannt. Jedoch spiegelt dieses Modell nicht die Realität wider, da es bei Polymeren den ausgeschlossenen Volumeneffekt gibt. Dieser besagt, dass zwei unterschiedliche Monomere nicht dieselbe Position im Raum einnehmen können. Beim Random Walk in $\mathbb{R}^3$ wird dieser Effekt nicht berücksichtigt. Anders hingegen ist es beim SAW. Dieser vermeidet eine solche Überschneidung.

Der Self-Avoiding Walk in $\mathbb{R}^3$ ist daher einer der besten Algorithmen für die Simulation von linearen Polymerketten.

3.5. Vergleich zwischen Self-Avoiding Walk und Random Walk in $\mathbb{R}$

Funktionsweise

In Anbetracht der Funktionsweise gibt es mehrere, deutliche Unterschiede. Einer dieser ist, dass der SAW keinen Punkt mehrfach besucht, wohingegen die RWiR jeden Punkt beliebig oft besuchen können. Dies hat zur Folge, dass der SAW vor Ablauf der angegebenen Schrittzahl n terminieren kann, weil kein Schritt mehr möglich ist. Die RWiR terminieren hingegen immer genau nach Ablauf der Schrittanzahl n und geben daher auch immer einen Walk der Länge $n+1$ zurück. Bei dem SAW hat der Walk keine feste Länge, sondern kann je nach Dimension in seiner Mindestlänge variieren, während seine maximale Länge aber weiterhin $n + 1$ ist.

Die Anzahl der verschiedenen Walks in Abhängigkeit von n können bei den RWiR außerdem in allen Dimensionen exakt bestimmt werden. Bei dem SAW ist dies nur in der ersten Dimension für alle $n \in \mathbb{N}$ möglich. In der zweiten Dimension ist es bisher nur für $n \leq 34$ möglich, diese Anzahl exakt zu bestimmen. [SM13]

Bei den Gemeinsamkeiten dieser Algorithmen fällt auf, dass sowohl der SAW als auch die RWiR eine maximale Schrittlänge und einen Startpunkt benötigen. Die Startpunkte liegen im Normalfall im Nullpunkt der jeweiligen Dimension. Hinsichtlich der Schrittwahl gibt es ebenfalls eine Gemeinsamkeit: Die Wahrscheinlichkeiten bei der Auswahl der erlaubten Schritte sind bei dem SAW und den RWiR jeweils gleich verteilt.

Einsatzgebiet

Sowohl der 3D-RW als auch der Self-Avoiding Walk in $\mathbb{R}^3$ (3D-SAW) können zur Modellierung von linearen Polymer-Ketten eingesetzt werden. Allerdings ist durch den 3D-RW nur eine ideale Modellierung möglich. Der 3D-SAW ist daher deutlich besser geeignet, wie bereits in Abschnitt 3.4 erläutert wurde.

Laufzeit

Die Laufzeiten der RWIR sind bereits in Abschnitt 2.5.3 aufgeführt. In allen Fällen beträgt die Laufzeit $O(n)$. Bezüglich des 1D-SAW ergibt sich ebenfalls eine Laufzeit von $O(n)$, da bei diesem Algorithmus n Mal in die zuerst gewählte Richtung gelaufen wird und diese dem Walk hinzugefügt wird. Dieses Vorgehen ist dem 1D-RW sehr ähnlich.

Beim 2D-SAW, welcher in Abschnitt 3.2 aufgeführt wurde, ergibt sich die folgende Laufzeit:

$$\begin{aligned} &O(1) + O(1) + O(n) \cdot (O(1) + O(1) + O(1) + O(1) + O(1)) \\ &= O(n) \end{aligned}$$

Wobei $O(n)$ die Schleife und $O(1)$ jeweils eine Zuweisung, ein Statement oder eine Funktion widerspiegelt.

Hinsichtlich der Laufzeit lässt sich somit kein Unterschied im Bereich der O-Notation feststellen.

4. Schönings Random Walk für 3SAT

3SAT ist ein NP-vollständiges Problem, weshalb davon ausgegangen wird, dass 3SAT lediglich effizient überprüfbar ist und es aus diesem Grund keinen Polynomialzeit-Algorithmus zur Lösung dieses Problems gibt. Bewiesen ist dies allerdings nicht, wie bereits in Abschnitt 2.1 erläutert. Aufgrund dessen wird seit längerer Zeit daran geforscht, wie schnell man dieses Problem mit Hilfe eines Algorithmus lösen kann. Eine der einfachsten und schnellsten Lösungen für das 3SAT Problem hat Uwe Schöning entwickelt. Er hat einen randomisierten Algorithmus geschrieben, welcher das Problem in $O(1.333^n)$ Zeit lösen kann [Sch99]. Die folgende Ausführung bezieht sich auf [Sch99].

4.1. Funktionsweise

Gegeben ist eine 3SAT Formel $F = c_1 \wedge c_2 \wedge \cdots \wedge c_m$, mit $c_k \neq c_l$, $l \neq k, 0 < k, l \leq m$. Dabei sind $c_1, \cdots, c_m$ Disjunktionen von Variablen oder negierten Variablen in 3KNF. Zuerst wird eine zufällige Zuordnung $a = [0,1]^n$ bestimmt, welche eine Belegung für die Variablen darstellt. Sollte dies eine erfüllende Zuordnung für F sein, gibt der Algorithmus *wahr* zurück. Sollte es sich allerdings um keine erfüllende Zuordnung handeln, so wird eine zufällige Klausel c von F, welche nicht durch a erfüllt ist, ausgewählt. Aus c wird dann eine zufällige Variable bestimmt. Daraufhin wird diese Variable in der Belegung geflippt und die neue Zuordnung geprüft. Dieser Vorgang wird maximal $3n$ Male durchlaufen oder solange bis eine erfüllende Belegung gefunden wurde, wobei $n = |Variablen|$. Jede Wiederholung spiegelt dabei einen Schritt wider. Die $3n$ Schritte lassen sich wie folgt herleiten:

Die Anzahl der Variablen, die geflippt werden müssen, um eine erfüllende Belegung zu erhalten, nennt man j, wobei $j \leq n$. Der Algorithmus könnte sich $i \leq j$ Schritte von der erfüllenden Zuordnung entfernen. Danach bräuchte er $i + j$ Schritte, um wieder zu der erfüllenden Zuordnung zu gelangen, also insgesamt $j + 2i$ Schritte. Aus diesen $j + 2i$ Schritten kann man die 3n Wiederholungen dann erschließen, da $j + 2i \leq n + 2n = 3n$.

Beispiel 2
$F = (x_1 \vee x_2 \vee x_4) \wedge (\overline{x_1} \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_3} \vee \overline{x_1} \vee \overline{x_4})$, $a = (1,0,0,1)$, $n = 4$, $j = 2$

Schritt 1: *a nicht erfüllend, wähle zufällig* $c = \overline{x_1} \vee x_3 \vee \overline{x_4}$*, wähle zufällig und flippe* x_3
$\Rightarrow a = (1,0,1,1)$, $j = 1$

Schritt 2: *a nicht erfüllend, wähle zufällig* $c = \overline{x_3} \vee \overline{x_1} \vee \overline{x_4}$*, wähle zufällig und flippe* x_4
$\Rightarrow a = (1,0,1,0)$, $j = 0$

Schritt 3: *a erfüllend, gebe wahr zurück.*

Um mit einer gewissen Wahrscheinlichkeit sagen zu können, dass es eine bzw. keine erfüllende Zuordnung für eine Formel in 3KNF gibt, muss der Algorithmus häufiger wiederholt werden. Die Wahrscheinlichkeit, dass der Algorithmus in einem Durchlauf eine erfüllende Zuordnung findet, liegt bei $\mathbb{P} = (\frac{3}{4})^n$. Das heißt, im Durchschnitt werden $\frac{1}{\mathbb{P}} = (\frac{4}{3})^n$ Durchläufe des Algorithmus benötigt, bis eine erfüllende Zuordnung gefunden wird. Um dann noch eine akzeptable Fehlerrate von e^{-20} zu erreichen, benötigt es t Wiederholungen mit $t = 20 \cdot (\frac{4}{3})^n$.

4.2. Pseudocode

Algorithmus 3 3SAT Random Walk Schöning [Sch99]

Eingabe: Formel in 3 KNF mit n Variablen
Ausgabe: Boolescher Wert

```
1: rate eine Zuweisung a ∈ {0,1}^n nach dem Zufallsprinzip
2: for 3n Male do
3:    if a ist eine erfüllende Zuweisung für f then
4:       return wahr
5:    end if
6:    c ← zufällige Klausel von F, die nicht durch a erfüllt wurde
7:    wähle eines der 3 Literale von c zufällig
8:    flippe das Literal in der aktuellen Zuweisung
9: end for
10: return falsch
```

Den oben stehenden Algorithmus lässt man exponentiell viele Male mit zufälligen Zuweisungen a ausführen. Sollte der Algorithmus dabei einmal ‚yes' zurückgeben, so gibt es eine Zuweisung, die die Formel erfüllt. Somit ist diese Formel dann in 3SAT.

4.3. Warum ist dieser Algorithmus ein Random Walk

Auf den ersten Blick ist kein Random Walk in diesem Algorithmus zu erkennen. Es scheint, als würde der Algorithmus lediglich dem Zufall zugrunde liegen. Auf den zweiten Blick jedoch ist der Random Walk, durch die folgende Betrachtungsweise, auch in diesem Algorithmus wieder zu finden:

Der Algorithmus startet in dem Zustand $z \in [0, \cdots, n]$. Der Zustand spiegelt die Hamming Distanz zwischen der aktuellen Zuweisung a und einer erfüllenden Zuweisung $a*$ wieder. Die Übergangwahrscheinlichkeit vom Start zu einem Zustand j beträgt hierbei $\binom{n}{j} \cdot 2^{-n}$. Sollte sich das System in dem Zustand $z = 0$ befinden, so wurde eine erfüllende Zuweisung gefunden. In einigen Fällen kann es auch dazu kommen, dass in einem anderen Zustand eine erfüllende Zuweisung gefunden wird. Dies geschieht, falls es mehrere verschiedene erfüllende Zuweisungen für diese Formel gibt. Die Übergangswahrscheinlichkeit von einem Zustand j zu einem Zustand $j + 1$ beträgt im Falle von kSAT $\frac{k-1}{k}$, also in unserem Fall $\frac{2}{3}$. Von einem Zustand j zu einem Zustand $j - 1$ hingegen entspricht die Wahrscheinlichkeit $\frac{1}{k}$, also $\frac{1}{3}$. $j + 1$ bewegt sich von der gewünschten Belegung weg und $j - 1$ nähert sich dieser an.

Wird der Algorithmus also aus dieser Perspektive betrachtet, so kann eindeutig ein Random Walk erkannt werden. Dieser Random Walk bewegt sich, wie der 1D-RW aus Abschnitt 2.5.1, in der ersten Dimension. In diesem Fall handelt es sich allerdings nicht um einen symmetrischen Random Walk, da keine Gleichverteilung der Übergangswahrscheinlichkeiten vorliegt.

4.4. Vergleich zwischen 3SAT Random Walk von Schöning und Random Walk in $\mathbb{R}$

Für diesen Vergleich wird der 3SAT Random Walk von Schöning (3RWS) aus der Sichtweise des Abschnitt 4.3 betrachtet.

Funktionsweise

Der RWiR und der 3RWS haben in ihrer Funktionsweise einige Gemeinsamkeiten, aber auch Unterschiede. Einer der größten Unterschiede ist die Verteilung der Wahrscheinlichkeiten. Während diese bei dem RWiR gleich verteilt sind, sind sie es bei dem 3RWS nicht. Dies hat zur Folge, dass bei dem 3RWS eine Richtung indirekt bevorzugt wird, was beim RWiR nicht der Fall. ist. Ein weiterer Unterschied ist die Länge des Walks. Bei dem RWiR gibt es eine von dem Nutzer festgelegte Länge n. Bei dem 3RWS beträgt die Länge mindestens 0 Schritte und maximal $3 \cdot |Variablen|$ Schritte. Die unterschiedlichen Längen sind zurückzuführen auf die verschiedenen Abbruchbedingungen der Algorithmen. Der RWiR bricht erst ab, sobald auch alle n Schritte vollzogen wurden. Der 3RWS hingegen bricht ab, sobald Zustand 0 erreicht, ein anderer akzeptierender Zustand gefunden oder $3 \cdot |Variablen|$ Schritte vollzogen wurden. Außerdem wird der 3RWS je nach Ausgabe exponentiell viel Male wiederholt, während der RWiR genau einmal durchlaufen wird, wenn nicht explizit etwas anderes angegeben wurde.

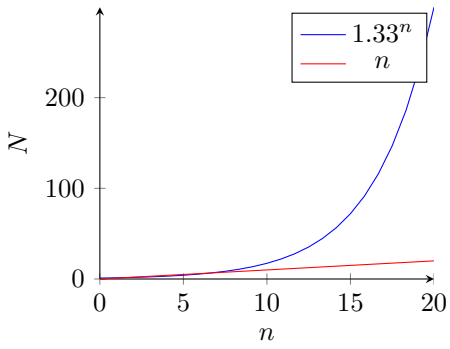
Eine Gemeinsamkeit der Algorithmen ist, dass bereits besuchte Punkte erneut besucht werden können und sich diese auch nicht gemerkt werden. Des Weiteren können beide als Markov-Kette gesehen werden, da der zukünftige Zustand/Punkt nicht von den vorherigen Zuständen/Punkten abhängig ist, sondern lediglich von dem Aktuellen.

Einsatzgebiet

Die Einsatzgebiete könnten unterschiedlicher kaum sein. Während der 3RWS in der Problematik der Entscheidungsprobleme, also hauptsächlich in der theoretischen Informatik, Anwendung findet, findet der RWiR bei der Modellierung von Polymer-Ketten Anwendung.

Laufzeit

Hinsichtlich der Laufzeit dieser Algorithmen gibt es einen erheblichen Unterschied. Der RWiR hat eine Laufzeit von $O(n)$, also eine lineare Laufzeit. Der 3RWS hat laut [Sch99] eine Laufzeit von $O(1.333^n)$, also eine exponentielle Laufzeit. Schaut man sich diese Laufzeiten für verschiedene n an, so wird der Unterschied sehr deutlich.



Die Laufzeit des 3RWS ist damit zu begründen, dass das Problem, mit welchem dieser Algorithmus sich beschäftigt, in NP liegt und somit nach bisherigen Forschungsstand lediglich effizient überprüfbar und nicht effizient lösbar ist.

5. Random Walk in Peer-to-Peer Netzwerken

5.1. Peer-to-Peer Netzwerk

Ein Peer-to-Peer Netzwerk (P2P Netzwerk) ist „ein System mit vollständig dezentraler Selbstorganisation und Ressourcennutzung“[SW04], in dem jeder Peer die gleichen Berechtigungen besitzt. P2P Netzwerke machen mittlerweile über 60% des Gesamtvolumens des Verkehrsaufkommen im Internet aus [MGS13].

Ein Peer besitzt in einem solchem Netzwerk sowohl die Fähigkeiten eines Clients als auch die eines Servers. Peers sind außerdem über das gesamte Internet miteinander verbunden und können weltweit verteilt sein. Somit lässt sich ein P2P Netzwerk als ungerichteter Graph $G = (V, E)$ darstellen, wie in Abbildung 5.1 zu erkennen ist. Peers haben allerdings keine feste Adresse, sondern sie werden meist durch unstrukturierte Identifikatoren adressiert. Dies hat zur Folge, dass Peers nur noch durch ihren Inhalt und eben nicht mehr über eine bekannte Adresse identifiziert werden können. In einem P2P Netzwerk ist also der genaue Ort von gesuchten Daten unbekannt. Daher müssen, bei einer Suche nach diesen Daten, an jedem besuchten Peer dessen Inhalte mit den gesuchten Inhalten verglichen werden, um festzustellen, ob der aktuelle Peer der Gesuchte ist. [SW04] Dies ist auf verschiedene Arten möglich. Im Folgenden werden zwei Methoden, die auf dem Random Walk basieren, betrachtet.

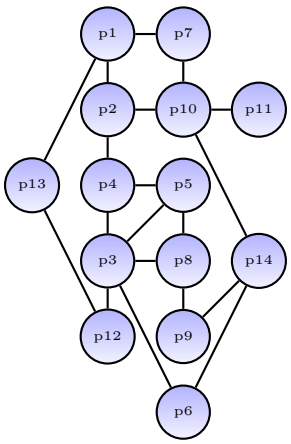


Abbildung 5.1.: Kleines Peer-to-Peer Netzwerk

5.2. Random Walk im P2P Netzwerk

Eine Möglichkeit der Suche des gewünschten Peers ist der Random Walk auf einem ungerichteten Graphen $G = (V, E)$, hier speziell der Random Walk im P2P Netzwerk (RIN).

Dieser Random Walk ähnelt dem in Abschnitt 3 vorgestellten SAW stark. Um die mögliche Anzahl verschiedener Richtungen zu ermitteln, wird in diesem Fall nicht die Dimension betrachtet, sondern der Grad des jeweiligen Peers $v \in V$. Ist also $Grad(v) = n$, so gibt es n verschiedene Möglichkeiten für den nächsten zufälligen Schritt. Die Wahrscheinlichkeiten sind am jeweiligen Peer gleich verteilt. Aufgrund dessen hat jede Kante e des Peers die Wahrscheinlichkeit $\mathbb{P}(e) = \frac{1}{n}$, um für den Schritt genutzt zu werden. An jedem Peer, der von dem Walker besucht wird, wird geprüft, ob dieser der Gesuchte ist, indem die Inhalte geprüft werden. Wenn dies der Fall ist, terminiert der Algorithmus, da er sein Ziel erreicht hat. Zu diesem Zeitpunkt ist dann die Position des gesuchten Peers bekannt. Falls der aktuelle Peer jedoch nicht dem gesuchten Peer entsprechen sollte, so wird der Random Walk fortgesetzt. Bereits besuchte Peers werden, so fern es möglich ist, gemieden. Besitzt ein Peer nur Nachbarn, die bereits besucht wurden, so wird einer dieser jedoch erneut besucht. Sollten alle Peers aus V besucht worden sein, so terminiert der Algorithmus. Der gesuchte Peer ist in diesem Fall entweder vom Startpunkt nicht erreichbar, existiert nicht mehr oder ist aktuell nicht online. [DLP07, MGS13, LCC+02]

Das Problem bei diesem Algorithmus ist der große Delay, der bei der Suche des richtigen Peers auftritt. Der Delay fällt bei kleinen Netzwerken zwar kaum auf, bei großen Netzwerken ist er jedoch deutlich erkennbar. [LCC+02]

Pseudocode

Algorithmus 4 Random Walk in einem P2P Netzwerk $G = (V, E)$

Eingabe: Graphen $G = (V, E)$, Startknoten $s \in V$, Gesuchter Knoten x
Ausgabe: Random Walk RW

1: RW ist eine Liste von unbekannter Länge
2: $RW[0] \leftarrow s$
3: $i \leftarrow 0$
4: $aktuellerKnoten \leftarrow s$
5: **while** $s \neq x$ und $\exists v \in V, v \notin RW$ **do**
6: $aktuellerKnoten \leftarrow randomNeighbour(s)$
7: **while** $aktuellerKnoten \notin RW$ und $\exists v \in neighbour(s), v \notin RW$ **do**
8: $aktuellerKnoten \leftarrow randomNeighbour(s)$
9: **end while**
10: $s \leftarrow aktuellerKnoten$
11: $RW[i+1] \leftarrow aktuellerKnoten$
12: **if** $\forall v \in V$ gilt $v \in RW$ **then**
13: **return** $x \notin V$
14: **end if**
15: $i = i + 1$
16: **end while**
17: **return** RW

Beispiel 3
Bei der Eingabe: Graph aus Abbildung 5.1, Startknoten: P2, Gesuchter Knoten: P12, könnte eine mögliche Ausgabe die Folgende sein:

$$P2 \rightarrow P4 \rightarrow P3 \rightarrow P8 \rightarrow P9 \rightarrow P14 \rightarrow P10 \rightarrow P7 \rightarrow P1 \rightarrow P13 \rightarrow P12$$

5.3. k-Random Walk

Der k-Random Walk (k-RW) ist eine Erweiterung des oben aufgeführten RIN. Bei dem k-RW werden k unabhängige Random Walks auf ungerichteten Graphen gestartet. $X = \{(x0_0, ..., x0_n), \cdots, (xk_0, ..., xk_n)\}$ ist ein k-RW, wobei $(x0_0, ..., x0_n)$ ein Random Walk im P2P Netzwerk und n die Anzahl der Schritte ist.

Die unabhängigen Random Walks laufen parallel ab und prüfen an jedem besuchten Peer, ob es sich um den gesuchten Peer handelt. Sollte dies der Fall sein, so wird eine Abbruchnachricht an alle übrigen Walker gesendet. Solange der gesuchte Peer nicht gefunden wurde und noch kein Walker alle Peers besucht hat, laufen alle Walker weiter. [MGS13, DLP07, LCC+02]

Die Größe von k kann zufällig bestimmt werden. Allerdings sollte immer beachtet werden, dass k nicht zu groß gewählt wird, um keinen großen Nachrichten-Overhead zu erzeugen, bei welchem sich einige Nachrichten sogar doppeln können. Dies würde für eine Überlastung des Netzwerkes sorgen und die Vorteile des k-RW würden erloschen sein.

Der Nachrichten-Overhead beschreibt die Anzahl der gesendeten Anfragen an die besuchten Peers.

Die Vorteile des k-RW liegen darin, dass bereits nach T Schritten dieselbe Anzahl an Knoten besucht wurde wie bei dem RIN nach $k \cdot T$ Schritten. Dies hat zur Folge, dass der Delay deutlich geringer ist wie noch bei dem RIN. Zudem hat der k-RW einen geringen Nachrichten-Overhead. [LCC+02]

Code Beispiel

Bei der realen Implementierung eines k-RW gibt es einige Probleme, welche beim Pseudocode nicht auftreten. Aufgrund dessen wird hier, anders als bei den anderen Algorithmen, kein Pseudocode, sondern Python Code verwendet. Zum besseren Verständnis wurde der Code ausführlich kommentiert. Kommentare sind in Python durch ein # gekennzeichnet.

Listing 5.1: Random Walk auf einen ungerichteten Graphen

```
1  from queue import Queue
2  from threading import Thread
3  import random
4  import numpy
5  import math
6  import time
7  import sys
8
9  #Prueft die Anzahl uebergebener Parameter
10 if len(sys.argv) != 4:
11     print('Es muessen genau drei Argumente uebergeben werden!')
12     exit()
13
14 #Das k im k-Random Walk
15 walker = int(sys.argv[1])
16 #Startknoten
17 start = str(sys.argv[2])
18 #Gesuchter Knoten
19 searched = str(sys.argv[3])
20 processes = []
21
22 #Prueft, ob die Elemente aus list1 in list2 enthalten sind
23 def isIn(list1, list2):
24   for v in list1:
25     if v not in list2:
26       return False
27   return True
```

```
28
29 #Startet verschiedene Threads mit der Function fun
30 def runInParallel (fun):
31   q = Queue()
32   global processes
33   for i in range(walker):
34     t = Thread(target = fun, args =(q,  ))
35     processes.append(t)
36   for p in processes:
37     p.start()
38
39 class Graph(object):
40
41     def __init__(self, graph_dict={}):
42         self.graph = graph_dict
43
44   #Gibt alle vertices des Graphen self wieder
45     def vertices(self):
46         return list(self.graph.keys())
47
48   #Gibt alle edges des Graphen self wieder
49     def edges(self):
50         edges = []
51         for vertex in self.graph:
52             for neighbour in self.graph[vertex]:
53                 edges.append([vertex, neighbour])
54         return edges
55
56   #Gibt alle Nachbarn des Knoten vertex im Graphen self wieder
57     def getNeighbours(self, vertex):
58         neighbours = []
59         for v in self.graph:
60             if vertex == v:
61                 for neighbour in self.graph[v]:
62                     neighbours.append(neighbour)
63         return neighbours
64
65   #Gibt einen zufaelligen Nachbarn des Knoten vertex
66   #im Graphen self wieder
67     def randomNeighbour(self, vertex):
68       step = random.choice(self.getNeighbours(vertex))
69       return step
70
71   #Random Walk auf einem Graphen durchfuehren, wobei self
72   #der Graph ist, q die Queue zur Kommunikation
73     def randomSearch(self, q):
74         global start, searched, processes
75         actual = start
76         walk = []
77         vert = self.vertices()
78         walk.append(actual)
79         n = 0
80         while actual != searched and not isIn(vert, walk) :
81             old = actual
82             neigh = self.getNeighbours(old)
83             actual = self.randomNeighbour(old)
84             while actual in walk and (not isIn(neigh, walk)):
85               actual = self.randomNeighbour(old)
86             walk.append(actual)
87             n += 1
88             time.sleep(0.001)
89             if not q.empty():
90               if q.get() == "done":
91                 return
```

```
92          for i in range(walker):
93            q.put("done")
94          if walk[-1]!=searched:
95            print ("Peer nicht gefunden")
96            return
97          print(walk)
98
99
100 if __name__ == "__main__":
101
102   #Graph wird erstellt
103   g = { "p1" : ["p2","p7", "p13"],
104           "p2" : ["p1", "p4", "p10"],
105           "p3" : ["p4", "p5", "p6", "p8", "p12"],
106           "p4" : ["p2", "p3", "p5"],
107           "p5" : ["p3", "p4", "p8"],
108           "p6" : ["p3","p14"],
109           "p7" : ["p1", "p10"],
110           "p8" : ["p3", "p9", "p5"],
111           "p9" : ["p8", "p14"],
112           "p10" : ["p7", "p14", "p11", "p2"],
113           "p11" : ["p10"],
114           "p12" : ["p13", "p3"],
115           "p13" : ["p12", "p1"],
116           "p14" : ["p10", "p9", "p6"]
117         }
118   graph = Graph(g)
119
120   #Prueft, ob der Startknoten existiert
121   if not start in graph.vertices():
122       print("Unbekannter Start Knoten")
123       exit()
124
125   #Start des k-Random Walks
126   runInParallel(graph.randomSearch)
```

Das erste und größte Problem, welches auftritt, ist die Implementierung der echten Parallelität. Diese ist auf aktuellen Computern bei großen Mengen parallel ablaufender Prozesse kaum umsetzbar. Folglich muss Pseudo-Parallelität verwendet werden. Diese kann mithilfe von Threads und einem minimalen Wartefenster (0,001 Sekunden) umgesetzt werden. Durch das Wartefenster, welches sich nach jedem Schritt eines Walker befindet, wird dem nächsten Walker die Rechenleistung zugeordnet und dieser kann daraufhin seinen Schritt ausführen. Dies führt zu einer Pseudo-Parallelität.

Das zweite Problem ist die Abbruchnachricht, welche hier durch ein Queue realisiert wurde. Über eine Queue kommunizieren die verschiedenen Threads miteinander, um so den aktuellen Status der anderen Threads überprüfen zu können. Ist ein Thread am Endknoten angekommen, so schreibt er k mal „done“ in die Queue und signalisiert so den anderen Threads, dass sie abbrechen können. Der Thread muss k mal „done“ in die Queue schreiben, denn sobald ein Thread auf eine Nachricht zugreift, verschwindet diese aus der Queue [Het20]. Um also alle Threads zu erreichen, braucht es k Abbruchnachrichten.

Eine mögliche Ausgabe des Algorithmus ist die Folgende:

$$P2 \to P1 \to P13 \to P12$$

5.4. Vergleich zwischen Random Walk im P2P Netzwerk und k-Random Walk

Funktionsweise

Die Funktionsweisen des RIN und des k-RW bauen aufeinander auf. Daher weisen sie einige Gemeinsamkeiten, aber auch Unterschiede auf. Ein Unterschied liegt darin, dass der k-RW den RIN mehrfach parallel ausführt. Des Weiteren besucht der RIN in t Schritten genau t Knoten. Der k-RW hingegen besucht in t Schritten $k \cdot t$ Knoten. Letzterer arbeitet also in weniger Schritten eine höhere Anzahl an Knoten ab und findet daher auch in kürzerer Zeit den Zielknoten, was sich besonders in großen Netzwerken bemerkbar macht. In der Tabelle 5.4 ist ein Vergleich des k-RW mit $k \in [2, 3, 4]$ und dem RIN zu sehen. Beide Algorithmen suchen im Netzwerk aus Abbildung 5.1 ausgehend von Peer $P2$ den Peer $P12$. Der Prozentwert bildet sich auf Basis von je 100 Versuchen.

k	**k-RW kürzer als RIN (in %)**
2	63
3	69
4	73

Tabelle 5.1.: k-RW vs. RIN

Bereits bei diesem kleinen Netzwerk fällt schnell auf, dass der k-RW häufig die kürzere Lösung findet und somit in diesen Fällen weniger Schritte benötigt als der RIN.

Außerdem unterscheiden sich die Algorithmen in der Anzahl der gesendeten Nachrichten/Anfragen an die Peers. Der RIN sendet nur eine Nachricht pro Schritt an einen Peer und sorgt so auf keinen Fall für eine Überlastung eines Netzwerkes. Anders kann es bei dem k-RW aussehen. Dieser sendet pro Schritt an k Peers eine Nachricht. Sollte k also zu groß gewählt werden, so kann es durchaus zur Folge haben, dass das Netzwerk überlastet wird. Wird k allerdings dem Netzwerk entsprechend klein gewählt ($k < max(Grad(v \in V))$, so besteht auch hier nicht die Gefahr der Überlastung.

Die Gemeinsamkeiten dieser Algorithmen sind sehr eindeutig, da der k-RW den RIN beinhaltet. Jeder Walker des k-RW spiegelt hierbei einen RIN wieder, weshalb auf die Gemeinsamkeiten nicht weiter eingegangen wird.

Einsatzgebiet

Bezüglich des Einsatzgebietes sind keine Unterschiede festzustellen. Beide Algorithmen werden zur Suche in P2P Netzwerken eingesetzt. Allerdings ist der k-RW die häufiger genutzte Methode zur Suche in einem solchem Netzwerk, da er die Stärken des RIN beinhaltet und zusätzlich einige seiner Schwächen verbessert.

Laufzeit

Bei dem RIN (Algorithmus 4) ergibt sich die folgende O-Notation:

$$O(1) + O(1) + O(1) + O(|V|) \cdot (O(1) + O(|V|) \cdot O(1) + O(1) + O(1) + O(|V|) + O(1))$$
$$= O(|V|^2)$$

Wobei $O(|V|)$ die zwei Schleifen und die IF-Anweisung widerspiegelt und O(1) jeweils eine Zuweisung, ein Statement oder eine Funktion.

Bei dem k-RW ergibt sich ebenfalls eine Laufzeit von $O(|V|^2)$, da der Ablauf sehr ähnlich zum RIN ist und die Parallelität keine negativen Auswirkungen auf die Laufzeit hat. Somit lässt sich bei der Laufzeit im worst-case kein Unterschied feststellen.

6. Alzheimer Random Walk

Der Alzheimer Random Walk (ARW) funktioniert ähnlich wie der SAW aus Abschnitt 3, jedoch mit dem Unterschied, dass er den besuchten Punkt nur mit einer Wahrscheinlichkeit von $0 \leq p_m \leq 1$ markiert. Ein Punkt kann somit doppelt besucht werden, wenn er nicht markiert wurde. Sollte $p_m = 1$ gelten, dann funktioniert dieser Algorithmus exakt wie der SAW, da jeder besuchte Punkt markiert wird. Sollte $p_m = 0$ gelten, dann funktioniert er wie der RWiR aus Abschnitt 2.5, da die besuchten Punkte in keinem Fall markiert werden. Der ARW kann in allen Dimensionen d in $\mathbb{R}^d$ ausgeführt werden. Bisher gibt es für diesen Algorithmus zwar noch keine konkrete Anwendung, jedoch sind die Entwickler des ARW der Meinung, dass er in vielen Bereichen wirtschaftlicher und assoziativer Gedächtnisprobleme eingesetzt werden könnte. [OK17]

Der Algorithmus wurde erst Ende 2017 entwickelt, was verdeutlicht, dass noch immer an der Effektivität und der Verwendung des Random Walks geforscht wird. Diese Ausführung des ARW beruht auf [OK17].

6.1. Funktionsweise

Die Funktionsweise des ARW wird anhand des folgenden Beispiels erläutert:

$X = (x_1, \ldots, x_n)$ ist ein ARW auf $\mathbb{R}^2$. Die Schrittlänge s beträgt $s = 1$. Somit ist $x_i \in [(0,1),(1,0),(0,-1),(-1,0)]$. Die Wahrscheinlichkeit p_m, dass ein bereits besuchter Punkt markiert wird, beträgt $p_m = 0,5$. Die Wahrscheinlichkeiten für die einzelnen Wege sind jeweils gleich verteilt. Daher gilt $\mathbb{P}(x_n = (0,1)) = \mathbb{P}(x_n = (1,0)) = \mathbb{P}(x_n = (0,-1)) = \mathbb{P}(x_n = (-1,0)) = 1/4$. Sollte aber beispielsweise $x_n = (-1,0)$ zu einem markierten Punkt führen, so würde $\mathbb{P}(x_n = (-1,0)) = 0$ und $\mathbb{P}(x_n = (1,0)) = \mathbb{P}(x_n = (0,-1)) = \mathbb{P}(x_n = (-1,0)) = 1/3$ werden. Sollte jeder benachbarte Knoten bereits markiert sein, so terminiert der Algorithmus. Gestartet wird der Algorithmus im Punkt $S_0 = (0,0)$. Das Ergebnis dieses Beispiels mit $n = 10$ Schritten ist in Abbildung 6.1 dargestellt. Die roten Punkte spiegeln in der Abbildung einen markierten Punkt wider, während die schwarzen Punkte einen besuchten, aber nicht markierten Punkt widerspiegeln. Dies ist nicht die einzige mögliche Lösung. Wie viele Lösungen es genau gibt, lässt sich nicht berechnen.

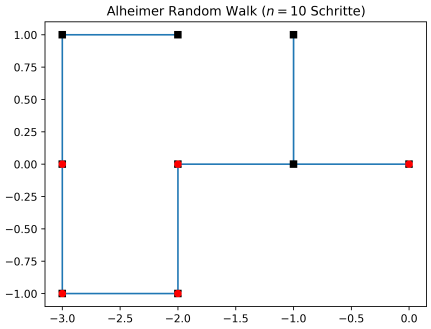


Abbildung 6.1.: Alzheimer Random Walk

In anderen Dimensionen $d \neq 2$ funktioniert der ARW analog zu diesem Beispiel. Lediglich die Anzahl an möglichen Richtungen ändert sich. Für jede weitere Dimension werden zwei Richtungen hinzugefügt. Es gibt also $2 \cdot d$ verschiedene Richtungen für den jeweiligen Walk. Außerdem ändert sich die Wahrscheinlichkeit $\mathbb{P}$ für die jeweiligen Richtungen x_n. Allgemein ist diese Wahrscheinlichkeit $\mathbb{P}(x_n) = \frac{1}{2 \cdot d}$. Sollte der aktuelle Knoten bereits n verschiedene markierte Nachbarn haben, so ändert sich die Wahrscheinlichkeit für die verbliebenen Richtungen in $\mathbb{P}(x_n) = \frac{1}{2 \cdot d - n}$. Anders als beim SAW ist der ARW der ersten Dimension kein Sonderfall.

6.2. Pseudocode

Der folgende Pseudocode beschreibt einen Alzheimer Random Walk in $\mathbb{R}^2$. Die Abbildung 6.1 ist eine mögliche Ausgabe RW dieses Algorithmus.

Algorithmus 5 Alzheimer Random Walk in $\mathbb{Z}^2$

Eingabe: Startpunkt $s \in \mathbb{Z}^2$, Schrittanzahl $n >= 0$, Merkwahrscheinlichkeit $0 \leq p_m \leq 1$
Ausgabe: Random Walk RW

1: RW ist eine Liste der Länge $n + 1$
2: $RW[0] \leftarrow s$
3: markiere s mit der Wahrscheinlichkeit p_m
4: $i \leftarrow 0$
5: **while** $i < n$ **do**
6: $possiblePoints \leftarrow$ jede Richtung $\in [(0,1),(0,-1),(1,0),(-1,0)]$, die zu einem unmarkierten Knoten führt
7: **if** $possiblePoint = \varnothing$ **then**
8: **return** RW
9: **end if**
10: $s \leftarrow randomPoint(possiblePoints)$
11: $RW[i+1] \leftarrow RW[i] + s$
12: $i \leftarrow i + 1$
13: markiere s mit der Wahrscheinlichkeit p_m
14: **end while**
15: **return** RW

6.3. Vergleich zwischen Alzheimer Random Walk und Self Avoiding Walk

Funktionsweise

Die Funktionsweise des SAW und des ARW ähnelt sich, je nach Wahl der Wahrscheinlichkeit p_m des ARW, mehr oder weniger stark. Sollte $p_m = 1$ gelten, so gibt es keinen Unterschied zwischen diesen Algorithmen, da der ARW dann genau wie der SAW jeden besuchten Knoten markiert und diesen meidet.

Sollte $p_m = 0$ gelten, so funktioniert der ARW wie ein Random Walk in $\mathbb{R}^d$. Der Vergleich zwischen SAW und Random Walk in $\mathbb{R}^d$ wird bereits in Abschnitt 3.5 thematisiert.

Der folgende Vergleich bezieht sich daher auf $0 < p_m < 1$.

Für den Fall, dass $0 < p_m < 1$, gibt es in der Funktionsweise einen großen Unterschied, aber auch einige Gemeinsamkeiten. Der Unterschied besteht darin, dass der ARW einen

besuchten Knoten nur mit einer bestimmten Wahrscheinlichkeit markiert und es aus diesem Grund möglich ist, dass er einige Knoten, falls diese nicht markiert wurden, mehrfach besuchen kann. Beim SAW hingegen ist es komplett ausgeschlossen, dass ein Knoten mehrfach besucht wird. Jeder besuchte Knoten wird markiert und in den folgenden Schritten gemieden. Markierte Knoten können dazu führen, dass der Algorithmus terminiert, falls der aktuelle Knoten ausschließlich von bereits markierten Knoten umgeben ist. Dies haben beide Algorithmen gemeinsam. Eine weitere Gemeinsamkeit ist die Gleichverteilung der Wahrscheinlichkeiten bei der Richtungsauswahl und die Abänderungen der Wahrscheinlichkeiten, falls sich markierte Knoten in der Nachbarschaft befinden.

Eine letzte wichtige Gemeinsamkeit ist, dass aktuell nicht für jeden Walk genau berechnet werden kann, wie viele mögliche verschiedene Ergebnisse bei einer Ausführung erreicht werden können.

Einsatzgebiet

Bei dem Einsatzgebiet gibt es einen entscheidenden Unterschied. Während der SAW in der Realität Anwendung bei der Modellierung von Polymer-Ketten findet, gibt es für den ARW keine konkreten Anwendungsgebiete. Der ARW hat ausschließlich ein Themengebiet, für das er eingesetzt werden könnte und zwar die wirtschaftlichen und assoziativen Gedächtnislücken. Das Themengebiet, in dem der SAW arbeitet, ist ein ganz anderes, nämlich das der Chemie.

Laufzeit

Für den SAW aus Algorithmus 2 wurde in Abschnitt 3.5 bereits gezeigt, dass er eine Laufzeit von $O(n)$ besitzt. Bei dem ARW aus Algorithmus 5 ergibt sich die folgende Laufzeit:

$$\begin{aligned} &O(1) + O(1) + O(1) + O(n) \cdot (O(1) + O(1) + O(1) + O(1) + O(1) + O(1)) \\ &= O(n) \end{aligned}$$

Wobei $O(n)$ die Schleife und $O(1)$ jeweils eine Zuweisung, ein Statement oder eine Funktion widerspiegelt.

Im Bezug auf die Laufzeit lassen sich daher im worst-case keine Unterschiede feststellen. Dies lässt sich unter anderem darauf zurückführen, dass der einzige Unterschied dieser Algorithmen darin besteht, dass die Makierung beim SAW immer geschieht und beim ARW nur mit der Wahrscheinlichkeit p_m. Daher ist die Implementierung dieser Algorithmen sehr ähnlich.

6.4. Anwendungsvorschlag

Idee

Die Idee besteht darin, den ARW in leicht abgewandelter Form für die Suche von Dateien in einem P2P Netzwerk einzusetzen. Die vorgesehene Form wäre ein k-Alzheimer Random Walk (k-ARW), welcher dem k-RW aus Abschnitt 5.3 ähnlich ist. Während ein k-RW bereits besuchte Knoten meidet, soll der k-ARW bereits besuchte Knoten mit einer gewissen Wahrscheinlichkeit nicht markieren und somit nicht meiden. Durch die Anwendung des k-ARW soll verhindert werden, dass durch die Meidung eines bereits besuchten Peers, welcher möglicherweise der einzige Nachbarpeer des gesuchten Knotens ist, letzterer ziemlich lange gemieden wird. Eine solche Situation wird im folgenden Beispiel dargestellt.

Beispiel 4
Betrachtet wird das Netzwerk aus Abbildung 5.1. Wenn nun ausgehend von dem Peer P7 der Peer P11 gesucht wird, könnte Folgendes passieren:

Von dem Peer P7 aus wird Peer P10 besucht und es wird geprüft, ob dieser der Gesuchte ist. Da dies nicht der Fall ist, wird zufällig ein Nachbar von P10 besucht. Sollte jetzt P14 besucht werden und nicht P11, so dauert es mindestens sieben Schritte, bevor P10 erneut besucht werden kann:

$$P7 \rightarrow P10 \rightarrow P14 \rightarrow P6 \rightarrow P3 \rightarrow P8 \rightarrow P9 \rightarrow P14 \rightarrow P10 \rightarrow P11$$

P14 und P10 dürfen in diesem Fall doppelt besucht werden, da P9 und P14 keine unmarkierten Nachbarn besitzen, aber noch nicht alle Peers besuchten wurden (siehe Abschnitt 5.3).

Damit der im Beispiel beschriebene Fall nicht eintritt, könnte ein k-Random Walk verwendet werden, welcher nicht selbst meidend ist. Dies führt aber zu dem Problem, dass einzelne Peers häufig besucht werden, während andere nicht einmal einen besuchten Nachbarpeer besitzen.

Um die Probleme des sich nicht selbstmeidenden k-RW und des k-RW zu lösen, kam mir die Idee, den ARW mit verschiedenen Markierwahrscheinlichkeiten p_m als k-ARW auf einem P2P Netzwerk anzuwenden.

Funktionsweise eines k-Alzheimer Random Walk

Bei dem k-ARW werden k unabhängige ARW in einem P2P-Netzwerk gestartet. $X = \{(x0_0, ..., x0_n), \cdots, (xk_0, ..., xk_n)\}$ ist ein k-ARW, wobei $(x0_0, ..., x0_n)$ ein ARW und n die Anzahl der Schritte ist.

Bei einem ARW in einem P2P-Netzwerk $G = (V, E)$ ergibt sich die Anzahl an möglichen Richtungen durch den *Grad* des aktuellen Peer $v \in V$. Die Wahrscheinlichkeiten für die jeweiligen Richtungen sind gleichverteilt.

Jeder ARW in dem k-ARW markiert einen bereits besuchten Peer mit der Wahrscheinlichkeit $0 < p_m < 1$. Die Markierung gilt nur für den jeweiligen ARW, das heißt, ein ARW berücksichtigt nur seine eigenen Markierungen. Ein bereits markierter Peer wird gemieden. Sollte der aktuelle Peer allerdings nur von bereits markierten Peers umgeben sein und noch nicht alle Peers besucht worden sein, dann besucht der ARW zufällig einen dieser Peers erneut.

Sobald der gesuchte Peer in einem der ARWs gefunden wurde oder ein ARW alle Peers besucht hat, wird eine Abbruchnachricht versendet. Der k-ARW terminiert und gibt den Ablauf des Walks dieses ARW zurück.

Hypothese und Prüfung

Hypothese 1
Der k-ARW findet für alle $k \in \mathbb{R}$, mit $0 < p < 1$, auf Netzwerken jeder Größe häufiger eine gleichlange oder kürzere Lösung als der k-RW.

Zur Prüfung dieser Hypothese wurden der k-RW und der k-ARW auf dem Netzwerk aus Abbildung 5.1 gestartet. Dabei wurden verschiedene Werte für p_m und k sowie verschiedene Startpeers und gesuchte Peers gewählt. Für jede Kombination von p_m und k wurden 100 Durchläufe gemacht und es wurde notiert, wie oft der k-ARW die kürzere Lösung gefunden hat. Die Ergebnisse dieser Durchläufe sind in Tabelle 6.1 dargestellt.

p_m	k	**k-ARW kürzere Lösung (in %)**
0.3	2	30
	3	37
	4	48
0.5	2	41
	3	42
	4	46
0.7	2	43
	3	44
	4	46
0.8	2	55
	3	59
	4	51
0.85	2	45
	3	55
	4	50
0.9	2	47
	3	55
	4	50

Tabelle 6.1.: Vergleich k-ARW und k-RW in einem kleinen Netzwerk

Für $p_m = 0.8$ hat sich die Hypothese für das Netzwerk aus Abbildung 5.1 bestätigen können. Für alle anderen Werte von p_m konnte sich die Hypothese in diesem Netzwerk nicht bestätigen. Um die Hypothese weitergehend zu prüfen, wird nun ein größeres Netzwerk betrachtet, nämlich das aus Abbildung 6.2. Auch hier werden die Ergebnisse des k-ARW mit denen des k-RW unter den Voraussetzungen der ersten Prüfung verglichen.

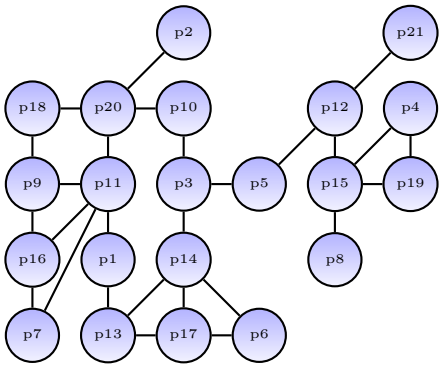


Abbildung 6.2.: ARW Peer-to-Peer Netzwerk

p_m	k	**k-ARW kürzere Lösung (in %)**
0.6	2	26
	3	30
	4	34
0.8	2	37
	3	40
	4	41
0.9	2	45
	3	53
	4	41

Tabelle 6.2.: Vergleich k-ARW und k-RW in einem großen Netzwerk

Nach Betrachtung der Ergebnisse aus Tabelle 6.2 wird deutlich, dass der k-ARW erheblich schlechter abschneidet als zuvor bei dem kleineren Netzwerk (Tabelle 6.1). Auch nach weiteren Tests mit verschiedenen Netzwerken, zeichnete sich immer mehr ab, dass bei großen Netzwerken nur gute Ergebnisse des k-ARW erzielt werden, falls auch p_m immer größer gewählt wird.

Da die meisten P2P Netzwerke sehr viel größer sind als die hier Aufgeführten, konvergiert p_m auf Dauer gegen 1. In diesem Fall wäre der k-ARW äquivalent zu dem k-RW. Gerade bei großen Netzwerken findet der k-RW häufiger die kürzere Lösung. Die Hypothese wurde daher eindeutig widerlegt.

Aufgrund dieser Tatsachen und der aufwendigen Auswahl des perfekten p_m im Verhältnis zum vorliegenden Netzwerk, ist der k-RW dem k-ARW vorzuziehen.

7. Graph Sampling

Graph Sampling beschäftigt sich mit dem Problem, wie man aus einem riesigen Graphen mit Milliarden von Knoten eine repräsentative Stichprobe entnehmen kann, um diesen Graphen mit Hilfe dieser Stichprobe effizient analysieren zu können. Graph Sampling wird besonders häufig in Sozialen Netzwerken oder zur Analyse des Webgraphen des World Wide Web eingesetzt, da es eine der kostengünstigsten Methoden ist, um diese effizient analysieren zu können [WCZ+11]. Im Folgenden wird ein idealer Graph Sampling Algorithmus, auch maximum-degree Random-Walk genannt, vorgestellt, welcher auf dem Random Walk basiert. Außerdem wird der WebWalker vorgestellt, welcher wiederum auf den Ideen des maximum-degree Random-Walk beruht. Random Walk basierte Graph Sampling Verfahren wurden als eine grundlegende Technik anerkannt, um gleichmäßig verteilte Proben aus einem Graphen zu entnehmen. [LYQ+15]

7.1. Struktur des Web

Das „Web“ kann durch einen Graphen dargestellt werden, wie in Abbildung 7.1. Ein solcher Graph ist in die folgenden vier Abschnitte unterteilt [BBC+00]:

(1) Eine große und stark verbundene Komponente in der jede Webseite (Knoten) jede anderen Webseite durch folgen der Links (Kanten) erreichen kann.

(2) Eine Komponente, in der jede Webseite durch (1) erreicht werden kann, jedoch können diese nicht zu (1) zurückkehren.

(3) Eine Komponente, in der jede Webseite zu einer Webseite in (1) gelangen kann, aber die Webseiten in (3) können nicht durch (1) erreicht werden.

(4) Eine Komponente, welche alle restlichen Webseiten enthält, welche zum Teil durch (3) erreicht werden können, zu Webseiten aus (2) gelangen können oder nicht durch Links erreicht werden können.

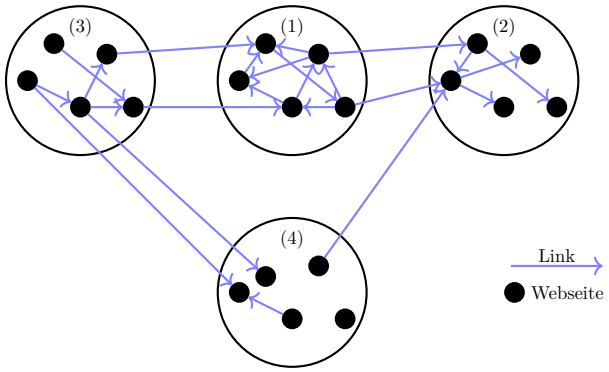


Abbildung 7.1.: Web Struktur

Die wichtigsten und am meisten durch User durchsuchten Bereiche sind hierbei Bereich (1) und (2). Diese Bereiche werden indexierbares Web genannt. [BBC+00]

7.2. maximum-degree Random-Walk-Algorithmus

Der maximum-degree Random-Walk-Algorithmus (MDRW), auch Ideal Random Walk on the Web genannt, stellt eine Markov-Kette auf einem ungerichteten regulären Graphen $G = (V, E)$ dar. Durch diese Eigenschaften kann bewiesen werden, dass durch den MDRW eine gleichmäßige Verteilung über die Knoten des Graphen erzeugt werden kann. Allerdings ist kaum ein Graph $G = (V, E)$, welcher in der Realität für Graph Sampling in Frage kommt, regulär und ungerichtet. Aufgrund dessen wird ein regulärer und ungerichteter Graph $G' = (V', E')$ auf Basis von G erstellt, wobei $V' = V$. Im ersten Schritt werden alle gerichteten Kanten $e \in E$ als ungerichtete Kanten zu G' hinzugefügt. Dies macht den Graphen allerdings noch nicht regulär. Um dies zu erreichen, werden jedem Knoten $v' \in V'$ so viele Selbstverweise hinzugefügt, dass $Grad(v')$ dem maximalen Grad eines Knotens in G' entspricht. Durch diese zwei Maßnahmen wird sichergestellt, dass ein ungerichteter regulärer Graph vorliegt. Des Weiteren erstellt man die Übergangsmatrix P_G für den Graphen G'. Diese wird für spätere Berechnungen benötigt. [BBC+00]

Um nun eine akzeptable Verteilung durch den MDRW, welche nahe einer Gleichverteilung ist, zu erhalten, muss der MDRW eine gewisse Anzahl an Schritten zurücklegen. Ein bekanntes Theorem für diese Anzahl der Schritte beziehungsweise für die Mischzeit t_{mix} mit Fehler $0 < \epsilon < 1$ ist:

Theorem 2 (mixing time [LPW08])
$t_{mix}(\epsilon) \leq log(\frac{1}{\epsilon \cdot \mathbb{P}_{min}}) \cdot t_{rel}$

Der eigentliche Walk auf dem Graphen G' funktioniert dann ähnlich zu dem Random Walk im P2P Netzwerk aus dem Abschnitt 5.2. Es gibt an jedem Knoten $v \in V'$ genau $Grad(v) = n$ Möglichkeiten für den nächsten zufälligen Schritt, wobei jeder Schritt die Wahrscheinlichkeit $\frac{1}{n}$ besitzt. Der Walk hat eine Länge von t_{mix} Schritten und jeder besuchte Knoten $v_i \in V$ mit $i \leq t_{mix}$ wird in eine Liste geschrieben. Die daraus resultierende Liste spiegelt nach Ablauf des MDRW dann die aktzeptable Verteilung, nahe der Gleichverteilung, über die Knoten des Graphen G' wider. [BBC+00]

Beispiel 5
Dieses Beispiel beruht auf den Wertangaben aus [BBC+00], welche sich auf einen indizierten Crawl des Web durch Alexa [ale] von 1996 beziehen. Dieser Crawl wird durch einen gerichteten Graphen $G = (V, E)$ dargestellt, wobei die Knoten die Webseiten und die Kanten die Links zwischen den Webseiten widerspiegeln. Der maximale Grad eines Knoten $v \in V$ in G beträgt 300.000. *Der durchschnittliche Grad eines Knoten $v \in V$ beträgt* 10 *und die Anzahl der Webseiten ist durch $\mid V \mid = 10^9$ dargestellt. Auf Basis von G wird nun ein g-regulärer ungerichteter Graph $G' = (V', E')$ erstellt, wobei $g =$* 300.000. *Dabei wird aus jeder gerichteten Kante $e \in E$ eine ungerichtete $e' \in E'$. Außerdem werden jedem Knoten $v' \in V'$ noch so viele Kanten e' auf sich selbst hinzugefügt, dass $Grad(v') =$* 300.000.

*Um nun die benötigte Anzahl an Schritten zu berechnen, die der MDRW zurücklegen muss, damit eine Verteilung nahe der Gleichverteilung erreicht wird, werden noch weitere Größen benötigt. Die spektrale Lücke von G' beträgt $\gamma = 10^{-5}$. Es wird davon ausgegangen, dass $\lambda_* = \lambda_2 = 0,99999$, woraus folgt, dass $\gamma = \gamma_* = 10^{-5}$. Für diesen Fall beträgt die relaxation time $t_{rel} = \frac{1}{\gamma *} =$ 100.000 und das minimale Wahrscheinlichkeitsmaß $\mathbb{P}_{min} = \frac{1}{|V|} = \frac{1}{10^9}$. Die Mischzeit mit Fehler 10^{-4} beträgt dann $t_{mix}(10^{-4}) \leq log(\frac{1}{\mathbb{P}_{min} \cdot 10^{-4}}) \cdot t_{rel} = log(\frac{1}{\frac{1}{10^9} \cdot 10^{-4}}) \cdot 100000 =$ 2.993.360 *Schritte. Zur Vereinfachung wird auf* 3.000.000 *Schritte aufgerundet.*

Im ersten Moment erscheint der Wert von 3.000.000 *Schritten sehr groß, bei genauerer Betrachtung sind es allerdings sehr wenige Schritte, welche tatsächliche Zugriffe auf neue Webseiten und damit Webzugriffe darstellen. Im Durchschnitt ist nur jeder* 30.000 *Schritt des Walks kein Selbstverweis, da im Schnitt nur* 10 *von* 300.000 *ausgehenden Links auf neue Webseiten führen. Außerdem muss die Webseite bei einem Selbstverweis nicht neu geladen werden. In diesem Beispiel sind also nur etwa* 100 *Webzugriffe notwendig, um eine Verteilung nahe der Gleichverteilung von diesem sehr großen Crawl zu erhalten.*

Pseudocode

Algorithmus 6 maximum-degree Random Walk

Eingabe: Graph $G = (V, E)$, Startknoten $s \in V$
Ausgabe: Sample S

1: $d \leftarrow max\{Grad(v), \forall v \in V\}$
2: d-regulärer ungerichteter Graph $G' = (V', E')$, $V' \leftarrow V$
3: füge jede Kante $e \in E$ als ungerichtete Kante E' hinzu
4: füge jedem Knoten $v' \in V'$ genau $d - Grad(v')$ Selbstverweise hinzu
5: $t_{mix} \leftarrow$ calculateTmix(G')
6: S ist eine leere Liste
7: $p \leftarrow s$
8: füge p zu S hinzu
9: **for** t_{mix} Schritte **do**
10: folge zufällig einer Kante von p
11: $p \leftarrow$ resultierende Seite
12: **if** $p \neq s$ **then**
13: füge p zu S hinzu
14: $s \leftarrow p$
15: **end if**
16: **end for**
17: **return** S

7.3. WebWalker

Der WebWalker ist ein Algorithmus, welcher auf den Ideen des MDRW basiert und im indexierbaren Web eingesetzt werden kann, um von diesem eine repräsentative Stichprobe zu erhalten. Die folgenden Ausführungen zu dem WebWalker beruhen auf [BBC+00]. Aufgrund der zugrunde gelegten Struktur des Web (Abbildung 7.1), kann der MDRW nicht ohne Anpassungen verwendet werden. Denn um einen idealen Random Walk durchführen zu können, müssen alle eingehenden und ausgehenden Kanten bekannt sein. Da aber auf Webseiten nicht vermerkt ist, welche Webseiten auf diese verlinken und auch nicht der gesamte Graph des aktuellen, indexierbaren Web bekannt ist, sind die eingehenden Kanten unbekannt. Um nun möglichst alle Kanten des Graphen zu erhalten, gibt es verschiedene Methoden.

Für die ausgehenden Kanten wird das HTML-Dokument der Seite verwendet, in welchem alle von der Seite verlinkten Webseiten stehen. Bei den eingehenden Kanten ist dies jedoch aufwendiger, da diese nicht im HTML-Dokument vermerkt sind. Eingehende Kanten können auf zwei verschiedene Varianten ermittelt werden: Die erste Variante ist, dass die Kanten während des Walks ermittelt werden. Jedes mal, wenn der Walk zu einer Webseite gelangt, wird in eine Liste eingetragen, über welche Webseite er dorthin gelangt

ist. Das Problem dieser Variante ist, dass, falls ein Link von Webseite u auf Webseite v existiert und dieser nicht verwendet wird, bevor v besucht wird, dieser Link bei der zufälligen Auswahl eines Nachbarn nicht berücksichtigt wird. Eine weitere Möglichkeit ist die Zuhilfenahme von Suchmaschinen, wie zum Beispiel Google. Allerdings geben Suchmaschinen maximal 1000 Webseiten, welche auf die gesuchte Webseite verlinken, zurück. Viele Seiten haben jedoch deutlich über 1000 Verweise, sodass auch hier das Problem auftritt, dass einige Links nicht berücksichtigt werden. Diese Probleme versucht der WebWalker so gut es geht zu umgehen.

Funktionsweise

Der WebWalker führt einen ungerichteten Random Walk auf dem Web-Graphen aus und stellt dabei einen d-regulären Graphen auf, wobei d dem maximalen Grad des Web entspricht. Dieser Wert ist in den meisten Fällen nicht bekannt. Er kann aber beliebig groß gewählt werden, da diese Wahl keinen Einfluss auf den Walk nimmt, solange dadurch nur die Anzahl der Selbstverweise erhöht wird.

Der d-reguläre Graph $G' = (V', E')$ wird während des Walks erstellt, indem jedes mal, wenn ein Knoten v das erste mal besucht wird, dieser zu V' hinzufügt wird. Auf Basis des HTML-Dokuments von v werden ungerichtete Kanten zu noch nicht besuchten Knoten hinzufügt. Außerdem wird mit Hilfe von Suchmaschinen nach Knoten gesucht, welche auf den aktuellen Knoten verweisen. Für diese Verweise wird ebenfalls jeweils eine ungerichtete Kante hinzugefügt, sofern die verweisende Seite noch nicht besucht wurde. Allerdings werden nur r verschiedene Ergebnisse der Suchmaschine verwendet, um eine mögliche Tendenz zu gesponserten oder ähnlichen Seiten, welche von den Suchmaschinen erfasst wurden, zu verringern. Zuletzt werden zusätzlich $d - Grad(v)$ Selbstverweise hinzugefügt.

Sollte ein Knoten u bereits besucht worden sein, so wird seine Kantenmenge $N(v)$ nicht mehr verändert, auch wenn er über eine Kante besucht wird, welche nicht in dieser Menge enthalten ist. Dadurch soll gewährleistet werden, dass jedes mal, wenn u besucht wird, exakt dieselbe Knotenmenge vorhanden ist. Das jedes mal dieselbe Knotenmenge vorhanden ist, hat den Effekt, dass sich der WebWalker wie eine Markov-Kette auf dem erstellten Graphen verhält. Aus Abschnitt 7.2 ist bekannt, dass für einen regulären ungerichteten Graphen der WebWalker in diesem Fall zu einer gleichmäßigen Verteilung konvergiert.

Nachdem ein Knoten/eine Webseite v abgearbeitet wurde, wird dieser Knoten einer Liste hinzugefügt. Zusätzlich wird ein zufälliger Link aus $N(v)$ verfolgt und die Webseite, welche aus der Verfolgung des Links resultiert, wird behandelt wie bereits beschrieben. Die resultierende Liste spiegelt nach Ablauf des WebWalkers eine akzeptable Verteilung nahe der Gleichverteilung wider.

Der Startknoten für den WebWalker sollte ein Knoten aus dem Bereich (1) sein. Meist wird dafür die Webseite www.yahoo.com verwendet, da von dort aus alle Webseiten aus Bereich (1) und (2) erreicht werden können.

Sollte eine Seite nicht erreichbar sein, eine über 300 Zeichen lange URL besitzen oder keine statische HTML-Seite sein, so wird diese als ‚schlecht' bezeichnet und ein anderer zufälliger Link der vorherigen Seite wird verfolgt.

Ein Problem bei diesem Algorithmus ist allerdings, dass nicht genau gesagt werden kann, nach wie vielen Schritten eine Gleichverteilung erreicht wird, da über den Webgraphen und dem daraus resultierenden Teilgraphen nicht genügend Informationen bekannt sind, um die mixing time berechnen zu können. Außerdem kann es durch die Suchmaschine dazu kommen, dass auch Webseiten aus den Bereichen (3) und (4) aus Abbildung 7.1 besucht werden, welche in diesem Fall keine Relevanz haben und nicht zum indexierbaren

Web gehören. Aus diesem Grund ist der WebWalker zwar nicht perfekt, aber durchaus nutzbar für Analysen, welche von einer kleinen Abweichung nicht stark beeinflusst werden.

Pseudocode

Dieser Pseudocode beschreibt das Besuchsverfahren einer Seite und das Auswahlverfahren der nächsten Seite.

Algorithmus 7 WebWalker [BBC+00]

Eingabe: Webseite v, Durchläufe n, Durchlauf i

füge v einer globalen Liste gl hinzu
if $i \geq n$ **then**
  return
end if
if v wurde bereits besucht **then**
  springe zu SELF
end if
$I \leftarrow r$ zufällige eingehende Kanten von v, welche von den Suchmaschinen zurückgegeben wurden
$O \leftarrow$ alle ausgehenden Kanten von v
for each $u \in (I \cup O) \setminus N(v)$ **do**
  if u wurde noch nicht besucht **then**
    füge u zu $N(v)$ hinzu
    füge v zu $N(u)$ hinzu
  end if
end for
SELF:
$w \leftarrow d- \mid N(v) \mid$
berechne Anzahl der Selbstverweise durch $1 - \frac{w}{d}$
SELECT:
wähle ein zufälliges $u \in N(v)$
if u ist schlecht **then**
  kehre zurück zu SELECT
end if
$this(u, n, i + 1)$

Für die vollständige Funktion des WebWalker fehlt noch eine aufrufende Funktion, in welcher die genaue und benötigte Schrittzahl berechnet wird. Da es für diesen Algorithmus jedoch keine zuverlässige Berechnung für die Konvergenz gibt, wurde dies außen vorgelassen.

7.4. Vergleich zwischen maximum-degree Random-Walk und WebWalker

Der Vergleich zwischen dem MDRW und dem WebWalker wird aufgeführt, um zu zeigen, inwiefern der WebWalker auf dem MDRW aufbaut und in welchen Punkten der MDRW verändert wurde.

Funktionsweise

In der Funktionsweise gibt es einige Gemeinsamkeiten, aber auch große Unterschiede. Der größte Unterschied liegt in der Behandlung des übergebenen Graphen. Während dieser bei dem MDRW vor dem Start des eigentlichen Walks umgeformt wird, wird er bei dem WebWalker während des Walks umgeformt. Außerdem wird bei dem MDRW vorausgesetzt, dass alle Kanten bekannt sind und daher auch direkt umgeformt werden können. Der WebWalker hingegen geht davon aus, dass die Kanten des zu behandelnden Graphen nicht oder nur teilweise bekannt sind, weshalb er versucht, diese während des Walks zu ermitteln.

Ein weiterer, entscheidender Unterschied ist, dass beim MDRW genau bekannt ist, wie viele Schritte dieser benötigt, um eine Verteilung nahe der Gleichverteilung über die Knoten des Graphen zu erhalten (t_{mix}). Bei dem WebWalker ist keine genaue Schrittzahl bekannt, um diese Verteilung zu erhalten. Es kann lediglich nachgewiesen werden, dass sie zu einem unbekannten Zeitpunkt auf jeden Fall erreicht wird.

Ein letzter Unterschied liegt im Code der beiden Algorithmen. Während der WebWalker rekursiv ausgeführt wird, nutzt der MDRW eine iterative Vorgehensweise.

Eine Gemeinsamkeit der Algorithmen befindet sich in der Abwandlung der Graphen. Beide Algorithmen erstellen auf Basis des zu behandelnden Graphen G einen neuen Graphen G', welcher d-regulär und ungerichtet ist, wobei d der maximale Grad eines Knoten von G ist. Das Problem der Regularität lösen der MDRW und der WebWalker durch das Einfügen von Selbstverweisen.

Eine weitere Gemeinsamkeit liegt in der Auswahl der nächsten Webseite beziehungsweise der zu folgenden Kante. Eine Kante wird mit der Wahrscheinlichkeit $\frac{1}{d}$ ausgewählt und verfolgt.

Einsatzgebiet

Das Einsatzgebiet des MDRW ist eigentlich Graph Sampling im Web. Jedoch ist dieser Algorithmus nur verwendbar, wenn die gesamte Struktur des übergebenen Graphen bekannt ist. Aus diesem Grund wird der Algorithmus in der Realität nicht für das Web verwendet, sondern dient nur als ideales Beispiel für einen Graph Sampling Algorithmus im Web. Allerdings kann dieser Algorithmus auf allen vollständig bekannten Graphen eingesetzt werden, um eine gleichmäßig verteilte Probe aus diesen Graphen zu erhalten.

Der WebWalker hingegen kann auf einem Webgraphen, welcher nicht vollständig bekannt ist, eingesetzt werden. Es kann jedoch nicht sicher gesagt werden, ob die erhaltene Verteilung nahe einer Gleichverteilung ist. Daher kann dieser Algorithmus nur in Bereichen eingesetzt werden, in welchen eine Abweichung keine großen Folgen hat.

Anders als der MDRW ist der WebWalker ausschließlich auf einen Webgraphen anwendbar.

Laufzeit

Für den MDRW aus Algorithmus 6 ergibt sich die folgende Laufzeit:

$$O(\mid V \mid) + O(1) + O(\mid E \mid) + O(\mid V \mid) + O(1) + O(n) \cdot (O(1) + O(1) + O(1)) + O(1)$$

$$= O(\mid V \mid + \mid E \mid + n)$$

Abhängig von der Eingabelänge $\mid G \mid + \mid n \mid + \mid s \mid$ ist diese linear.

Für den WebWalker aus Algorithmus 7 ergibt sich die folgende Laufzeit:

$$n \cdot (O(1) + O(1) + O(1) + O(1) + O(1) + O(\mid V \mid) \cdot (O(1)) + O(1) + O(1) + O(1) + O(1))$$
$$= O(n \cdot \mid V \mid)$$

Abhängig von der Eingabelänge $\mid v \mid + \mid n \mid + \mid i \mid$ ist diese quadratisch.

Bei der Laufzeit gibt es einen großen Unterschied zwischen diesen beiden Algorithmen. Während der MDRW im worst-case eine lineare Laufzeit hat, hat der WebWalker im worst-case eine quadratische Laufzeit. Dies ist darauf zurückzuführen, dass der MDRW zuerst den regulären Graphen erstellt und dann den Random Walk auf diesem ausführt, während der WebWalker dies zeitgleich macht und daher in jedem Durchlauf i des Walks im worst-case $\mid V \mid -i$ Knoten durcharbeiten muss (Zeile 10).

8. Übersicht und Fazit

Zum Abschluss dieser Arbeit wird eine Übersicht über alle vorgestellten Algorithmen in Tabelle 8.1 gegeben und ein Fazit in Form eines Vergleichs aus dieser Übersicht gezogen.

Algorithmus	**Beschreibung**	**Einsatzgebiet**	**Laufzeit**	**Schrittanzahl**
Random Walk in $\mathbb{R}$	Ein Random Walk in $\mathbb{R}^d$ ist ein stochastischer Prozess, um eine Bewegung zu simulieren, bei der die Schritte in eine Richtung zufällig erfolgen.	Grundlage aller Random-Walk-Algorithmen	$O(n)$ - linear	n - vom Nutzer festgelegt
Self-Avoiding Walk	Ein Self-Avoiding Walk ist ein Random Walk, bei dem kein Punkt doppelt besucht werden darf.	Chemie - Modellierung von Polymerketten	$O(n)$ - linear	$\leq n$ - vom Nutzer festgelegt
Schönings Random Walk für 3SAT	Der Random Walk für 3SAT von Schöning bestimmt für eine Formel in 3KNF, ob sie in 3SAT liegt oder nicht.	Theoretische Informatik - Lösung des 3SAT Problems	$O(1.333^n)$ - exponentiell	$\leq 3 \cdot \mid Variablen \mid$
Random Walk im Peer to Peer Netzwerk	Ein Random Walk im Peer to Peer Netzwerk ist ein Random Walk auf einem Graphen, welcher einen gesuchten Peer sucht und lokalisiert.	Peer to Peer Netzwerke	$O(\mid V \mid^2)$ - quadratisch	$n < \infty$
k-Random Walk	Ein k-Random Walk ist ein Random Walk, bei dem k Walker parallel in einem Peer to Peer Netzwerk einen gesuchten Peer suchen und lokalisieren.	Peer to Peer Netzwerke	$O(\mid V \mid^2)$ - quadratisch	$n < \infty$
Alzheimer Random Walk	Der Alzheimer Random Walk ist ein Random Walk, bei dem ein Punkt mit einer gewissen Wahrscheinlichkeit nicht erneut besucht wird.	Voraussichtlich in Bereichen wirtschaftlicher und assoziativer Gedächtnisprobleme	$O(n)$ - linear	$\leq n$ - vom Nutzer festgelegt
maximum-degree Random Walk	Ein maximum-degree Random Walk ist ein Random Walk, bei dem eine repräsentative Stichprobe von Knoten eines ungerichteten Graphen erzeugt wird.	Graphen-Analyse	$O(\mid V \mid + \mid E \mid + n)$ - linear	$n \leq log(\frac{1}{\epsilon \cdot \mathbb{P}_{min}}) \cdot t_{rel}$
WebWalker	Der WebWalker ist ein Random Walk, bei dem eine repräsentative Stichprobe von Webseiten eines Webgraphen erzeugt wird.	Webgraphen-Analyse	$O(n \cdot \mid V \mid)$ - quadratisch	$n < \infty$

Tabelle 8.1.: Vergleich aller Algorithmen

Die vorgestellten Random-Walk-Algorithmen verdeutlichen die in der Einleitung er-

wähnte Vielfalt der Einsatzgebiete. Allein in dieser Arbeit wurden fünf verschiedene Einsatzgebiete für die verschiedenen Algortihmen aufgeführt. Die quasi grenzenlose Einsatzmöglichkeit von Random-Walk-Algorithmen führt zu einem großen wirtschaftlichen Interesse an ihnen.

Des Weiteren fällt auf, dass die Laufzeiten der Algorithmen verschieden und nicht automatisch linear sind, nur weil der ursprünglichen Random Walk in $\mathbb{R}$ eine lineare Laufzeit hat. Eine bessere als die lineare Laufzeit wird aber durch keinen der aufgeführten Algorithmen erreicht. Außerdem ist zu erkennen, dass die Laufzeit stark von dem Einsatzgebiet abhängig ist. Algorithmen, die ähnliche Probleme behandeln, haben in den meisten Fällen im worst-case die gleiche Laufzeit. Die einzigen Ausnahmen sind der WebWalker und der maximum-degree Random Walk, welche beide in einem ähnlichen Bereich eingesetzt werden, aber unterschiedliche Laufzeiten aufweisen. Der Random Walk für 3SAT von Schöning, welcher mit einer exponentiellen Laufzeit mit Abstand die längste Laufzeit aller aufgeführten Algorithmen hat, sticht besonders heraus. Die exponentielle Laufzeit hängt auch hier mit dem Einsatzgebiet zusammen.

Die Abhängigkeit von dem Einsatzgebiet spiegelt sich auch bei der Schrittanzahl der einzelnen Walks wider. Bei vielen der Walks (Random Walk in $\mathbb{R}$, Self-Avoiding Walk, Schönings Random Walk für 3SAT, Alzheimer Random Walk, maximum-degree Random Walk) ist die Länge des Walks bezüglich der maximalen Länge genau definiert. Die exakte Länge des Walks ist nur in einem Fall vorgegeben (Random Walk in $\mathbb{R}$). Bei dem Random Walk im Peer to Peer Netzwerk, dem k-Random Walk und dem WebWalker ist eine maximale oder minimale Länge nicht exakt vorgegeben.

9. Ausblick

Der Random Walk wurde in dem Jahr 1905 das erste Mal erwähnt [Hen18]. Bis heute wurden viele verschiedene Algorithmen auf Basis des Random Walks entwickelt. Die Summe dieser Algorithmen ist sehr groß, weshalb nicht alle interessanten Algorithmen in dieser Arbeit aufgeführt werden konnten.

Ein interessanter Themenbereich, welcher in dieser Arbeit nicht betrachtet wurde, ist der der Expander Graphen. Random-Walk-Algorithmen auf Expander Graphen finden Anwendung in theoretischen und praktischen Rechenproblemen [KM17]. Eine konkrete Anwendung für einen solchen Algorithmus ist zum Beispiel die Pfadauswahl in Expander Graphen, welche in der Realität in Kommunikationsnetzwerken Anwendung findet [BFU99]. Der Vorteil dieser Walks ist, dass sie schnell gegen ihre stationäre Verteilung konvergieren [KM17].

Ein weiterer Algorithmus, welcher auf den Algorithmen auf Expander Graphen aufbaut, ist der High Order Random Walk (HD-Walk) von Tali Kaufman und David Mass [KM17]. Der HD-Walk findet Anwendung auf hochdimensionalen Simplizialkomplexen, welche analog zu hochdimensionalen Graphen sowie Expandern sind. Der HD-Walk hat, obwohl er erst 2016 entwickelt wurde, bereits zu einigen Durchbrüchen in der theoretischen Informatik geführt [HKL20].
Der bekannteste Durchbruch ist Kuikui Liu und Shayan Oveis Gharan im Jahr 2019 gelungen. Sie beschäftigten sich mit einseitigen lokal-spektralen Expandern und konnten dadurch eine 30 Jahre alte Vermutung von Mihail und Vazirani bestätigen. Für das zugehörige Paper [ALGV19] gewannen Kuikui Liu und Shayan Oveis Gharan den Best Paper Award der Association for Computing Machinery's 51st annual Symposium on the Theory of Computing (STOC 2019).

Ein aktueller Algorithmus (16. November 2020) aus dem Themenbereich der Expander Graphen ist der HD-Walk auf zweiseitigen lokal-spektralen Expandern. Dieser ist der erste Polynomialzeit-Algorithmus für „affine unique games over the Johnson sheme“, welche ein Problem der Graphentheorie sind [HKL20].

Weitere Aspekte, die in einer weiterführenden Arbeit genauer behandelt werden könnten, sind einige ungeklärte Probleme bezüglich verschiedener Random Walk Algorithmen. Beispiele dafür sind der WebWalker, bei dem noch kein Theorem für die mixing time bekannt ist oder der Self-Avoiding Walk mit n Schritten, bei dem die genaue Anzahl der möglichen Wege noch nicht für alle Dimensionen und n bekannt ist.

Abschließend kann ich sagen, dass alle in dieser Arbeit dargestellten Aspekte und Forschungen den Random Walk zu einem interessanten Themenbereich machen, weshalb ich auch in Zukunft neue Errungenschaften und Paper im Bereich der Random-Walk-Algorithmen im Blick behalten werde und dies auch jedem anderen empfehlen kann.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ale] Alexa. `https://www.alexa.com`, `https://archive.org/details/alexacrawls`.

[ALGV19] Nima Anari, Kuikui Liu, Shayan Oveis Gharan, and Cynthia Vinzant. Log-concave polynomials II: high-dimensional walks and an FPRAS for counting bases of a matroid. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 1–12. ACM, 2019.

[BBC+00] Ziv Bar-Yossef, Alexander C. Berg, Steve Chien, Jittat Fakcharoenphol, and Dror Weitz. Approximating aggregate queries about web pages via random walks. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 535–544. Morgan Kaufmann, 2000.

[Beh13] Ehrhard Behrends. *Elementare Stochastik : ein Lernbuch - von Studierenden mitentwickelt*. Vieweg+Teubner Verlag, Wiesbaden, 2013.

[Ber97] Lawrence Gray Bert Fristedt. *A Modern Approach to Probability Theory*. Birkhäuser, Boston, MA, 1997.

[BFU99] Andrei Z. Broder, Alan M. Frieze, and Eli Upfal. Static and dynamic path selection on expander graphs: A random walk approach. *Random Struct. Algorithms*, 14(1):87–109, 1999.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[DGKG65] V. D. Gupta and A. K. Gupta. A two-dimensional random walk model for polymers. *Journal of the Physical Society of Japan*, 20(3):423–427, 1965.

[DLP07] Reza Dorrigiv, Alejandro López-Ortiz, and Pawel Pralat. Search algorithms for unstructured peer-to-peer networks. In *32nd Annual IEEE Conference on Local Computer Networks (LCN 2007), 15-18 October 2007, Clontarf Castle, Dublin, Ireland, Proceedings*, pages 343–352. IEEE Computer Society, 2007.

[Fec18] Uli Fechner. Algorithmus, algorithm. In *Roempp Lexikon Chemie vom Georg Thieme Verlag KG*. Georg Thieme Verlag KG, 2018.

[Gas19] William I. Gasarch. Guest column: The third p=?np poll. *SIGACT News*, 50(1):38–59, 2019.

[Hen17] Norbert Henze. *Stochastik für Einsteiger : Eine Einführung in die faszinierende Welt des Zufalls*. Springer Spektrum, Wiesbaden, 2017.

[Hen18] Norbert Henze. *Die einfache symmetrische Irrfahrt auf Z – gedächtnisloses Hüpfen auf den ganzen Zahlen*. Springer Fachmedien Wiesbaden, Wiesbaden, 2018.

[Het20] Raymond Hettinger. A synchronized queue class. `https://github.com/python/cpython/blob/master/Doc/library/queue.rst`, 27. April 2020.

[HKL20] Max Hopkins, Tali Kaufman, and Shachar Lovett. High dimensional expanders: Random walks, pseudorandomness, and unique games. *CoRR*, abs/2011.04658, 2020.

[IW18] Joshua Izaac and Jingbo Wang. Python. In *Undergraduate Lecture Notes in Physics, Computational Quantum Mechanics*, pages 83–162. Springer International Publishing, Cham, 2018.

[Joh11] Derek Johnston. An introduction to random walks. `https://www.math.uchicago.edu/~may/VIGRE/VIGRE2011/REUPapers/Johnston.pdf`, 5. August 2011.

[Kle] Bernd Klein. Graphen in python. `https://www.python-kurs.eu/graphen_python.php`. Accessed: 2020-11-22.

[KM17] Tali Kaufman and David Mass. High dimensional random walks and colorful expansion. In Christos H. Papadimitriou, editor, *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*, volume 67 of *LIPIcs*, pages 4:1–4:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[Knu68] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[LCC+02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In Richard R. Muntz, Margaret Martonosi, and Edmundo de Souza e Silva, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, June 15-19, 2002, Marina Del Rey, California, USA*, pages 258–259. ACM, 2002.

[LPW08] David Levin, Yuval Peres, and Elizabeth Wilmer. Markov chains and mixing times. with a chapter on "coupling from the past" by james g. propp and david b. wilson. 01 2008.

[LYQ+15] Rong-Hua Li, Jeffrey Xu Yu, Lu Qin, Rui Mao, and Tan Jin. On random walk based graph sampling. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 927–938. IEEE Computer Society, 2015.

[MGS13] Mohammad Reza Meybodi, Mahdi Ghorbani, and Ali Mohammad Saghiri. A new version of k-random walks algorithm in peer-to-peer networks utilizing learning automata. In *The 5th Conference on Information and Knowledge Technology*, pages 1–6. IEEE, 2013.

[MS09] Franklin Mendivil and Ronald W. Shonkwiler. *Explorations in Monte Carlo methods, Undergraduate texts in mathematics*. Springer, Dordrecht, 2009.

[MV15] Arne Meier and Heribert Vollmer. *Komplexität von Algorithmen, Informatik, Mathematik für Anwendungen*, volume 4. Lehmanns Media, Berlin, 2015.

[OK17] Takashi Odagaki and Keisuke Kasuya. Alzheimer random walk, the european physical journal b. *The European Physical Journal B*, 90(9):1–5, 2017.

[Pea05a] Karl Pearson. The problem of the random walk. *Nature*, 72(1865):294–294, Jul 1905.

[Pea05b] Karl Pearson. The problem of the random walk. *Nature*, 72(1867):342–342, Aug 1905.

[pyt] Python documentation. `https://docs.python.org/3/`.

[Sch99] Uwe Schöning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 410–414. IEEE Computer Society, 1999.

[SM13] Gordon Slade and Neal Madras. *The Self-Avoiding Walk, Modern Birkhäuser Classics*. Springer, New York, NY, 2013.

[Sri17] KR Srinath. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET)*, 4(12):354–357, 2017.

[SS02] Thomas Schickinger and Angelika Steger. *Stochastische Prozesse*, pages 165–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[Str03] Gilbert Strang. *Lineare Algebra, Introduction to linear algebra, Springer-Lehrbuch*. Springer, Berlin, 2003. Introduction to linear algebra.

[SW04] Ralf Steinmetz and Klaus Wehrle. Peer-to-peer-networking & -computing - aktuelles schlagwort. *Inform. Spektrum*, 27(1):51–54, 2004.

[WCZ+11] Tianyi Wang, Yang Chen, Zengbin Zhang, Tianyin Xu, Long Jin, Pan Hui, Beixing Deng, and Xing Li. Understanding graph sampling algorithms for social network analysis. In *31st IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2011 Workshops), 20-24 June 2011, Minneapolis, Minnesota, USA*, pages 123–128. IEEE Computer Society, 2011.

[Yan10] Xin-She Yang. Random walk and markov chain. In *Engineering Optimization, An Introduction with Metaheuristic Applications*, pages 153–170. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2010.

A. Python Code Beispiele

Einfache symmetrische Irrfahrt auf $\mathbb{Z}$

Listing A.1: Symmetrische Irrfahrt, 100 Schritte

```
1  import random
2  import numpy as np
3  import pylab
4  import sys
5
6  #Prueft die Anzahl uebergebener Parameter
7  if len(sys.argv) != 2:
8          print('Es muss genau ein Argument uebergeben werden!')
9          exit()
10
11 #Anzahl der Schritte
12 n= int(sys.argv[1])
13 #Startpunkt
14 x=0
15 positions = [x]
16 #Wiederholungen des zufaelligen Richtungswechsel
17 for i in range(n):
18     step = random.choice(['L', 'R'])
19     if step == 'L':
20         x=x-1
21     elif step == 'R':
22         x=x+1
23     positions.append(x)
24 #Graph anzeigen lassen
25 pylab.title("1D Random Walk")
26 pylab.plot(positions)
27 pylab.show()
```

Aufruf: python3 Einfache_symmetrische_Irrfahrt.py <Anzahl an Schritten>

2D Random Walk

Listing A.2: 2D Random Walk, 10000 Schritte

```
1  import numpy
2  import pylab
3  import random
4  import sys
5
6  #Prueft die Anzahl uebergebener Parameter
7  if len(sys.argv) != 2:
8      print('Es muss genau ein Argument uebergeben werden!')
9      exit()
10
11
12 #Anzahl der Schritte
13 n = int(sys.argv[1])
14
15 #Jeweils ein Array fuer die x und y Achse, in der Groesse der Schritte
16 x = numpy.zeros(n)
17 y = numpy.zeros(n)
18
19 #Wiederholungen des zufaelligen Richtungswechsels
20 for i in range(1, n):
21     step = random.choice(['Left', 'Right', 'Up', 'Down'])
22     if step == 'Right' :
23         x[i] = x[i - 1] + 1
24         y[i] = y[i - 1]
25     elif step == 'Left':
26         x[i] = x[i - 1] - 1
27         y[i] = y[i - 1]
28     elif step == 'Up':
29         x[i] = x[i - 1]
30         y[i] = y[i - 1] + 1
31     elif step == 'Down':
32         x[i] = x[i - 1]
33         y[i] = y[i - 1] - 1
34
35
36 #Graph anzeigen lassen:
37 pylab.title("Random Walk ($n = " + str(n) + "$ Schritte)")
38 pylab.plot(x, y)
39 pylab.show()
```

Aufruf: python3 2D_Random_Walk.py <Anzahl an Schritten>

Self-Avoiding Walk

Listing A.3: Self-Avoiding Walk, 1000 Schritte

```
1  import numpy
2  import pylab
3  import random
4  import sys
5
6  #Prueft die Anzahl uebergebener Parameter
7  if len(sys.argv) != 2:
8      print('Es muss genau ein Argument uebergeben werden!')
9      exit()
10
11
12 #Anzahl der Schritte
13 n = int(sys.argv[1])
14
15 #Zwei Arrays, fuer x und y Achse
16 #Beide Arrays haben die Groesse n
17 #und sind mit 0en gefuellt
18 x = numpy.zeros(n)
19 y = numpy.zeros(n)
20 possible = []
21 visited = [[0 for x in range(n)] for y in range(n)]
22 posx = n // 2
23 posy = posx
24 visited[posx][posy] = 1
25
26 #Wiederholungen des zufaelligen Richtungswechsels
27 for i in range(1, n):
28     #Welche Schritte sind ueberhaupt moeglich
29     possible = []
30     if not visited[posx + 1][posy]:
31         possible.append('Right')
32     if not visited[posx - 1][posy]:
33         possible.append('Left')
34     if not visited[posx][posy + 1]:
35         possible.append('Up')
36     if not visited[posx][posy - 1]:
37         possible.append('Down')
38     if len(possible) == 0:
39         print('Abbruch')
40         while i < n:
41             x[i] = x[i - 1]
42             y[i] = y[i - 1]
43             i+=1
44         break
45     #Zufaellige Auswahl aus moeglichen Schritten
46     step = random.choice(possible)
47     if step == 'Right' :
48         x[i] = x[i - 1] + 1
49         posx += 1
50         y[i] = y[i - 1]
51     elif step == 'Left':
52         x[i] = x[i - 1] - 1
53         posx -= 1
54         y[i] = y[i - 1]
55     elif step == 'Up':
56         x[i] = x[i - 1]
57         y[i] = y[i - 1] + 1
58         posy += 1
59     elif step == 'Down':
60         x[i] = x[i - 1]
```

```
61            y[i] = y[i - 1] - 1
62            posy -= 1
63        visited[posx][posy] = 1
64
65    #Graph anzeigen lassen
66    pylab.title("Self-Avoiding Walk ($n = " + str(n) + "$ Schritte)")
67    pylab.plot(x, y)
68    pylab.show()
```

Aufruf: python3 Self-Avoiding_Walk.py <Anzahl an Schritten>

Schönings Random Walk für 3SAT

Listing A.4: Schönings Random Walk für 3SAT

```
1  import random
2
3  #Spiegelt eine Variable aus einer Disjunktion wieder
4  class Var(object):
5      #Flag ist false, wenn die Variable negiert verwendet wird
6      def __init__(self, varname, flag):
7          self.varname = varname
8          self.value = 0
9          self.flag = flag
10
11
12 #Spiegelt eine einzelne Disjunktion wieder
13 class Disj(object):
14     def __init__(self, var1, var2, var3):
15         self.vars = [var1, var2, var3]
16
17 #Spiegelt eine Formel in 3KNF wieder und kann einige
18 #Operationen auf dieser durchfuehren
19 class Sat(object):
20
21     #Initialisiert aus einer Liste von Variablen eine
22     #Formel in 3KNF
23     def __init__(self, disjs):
24         if len(disjs) % 3 != 0:
25             print('no 3Sat')
26             return
27         SatList = []
28         i = 0
29         while i < len(disjs):
30             disjunktion = Disj(disjs[i], disjs[i+1], disjs[i+2])
31             SatList.append(disjunktion)
32             i += 3
33         self.disjunktions = SatList
34
35     #Gibt die verschiedenen Variablen zurueck
36     def varList(self):
37         vlist = []
38         for i in self.disjunktions:
39             for j in i.vars:
40                 if j.varname not in vlist:
41                     vlist.append(j.varname)
42         return vlist
43
44     #Prueft, ob die aktuelle Formel mit der Belegung erfuellt wird
45     def isTrue(self):
46         if len(self.getFalse()) == 0:
47             return True
48         else:
49             return False
50
51     #Erzeugt eine zufaellige Anfangsbelegung der Variablen
52     def setValues(self):
53         value = random.choice([0,1])
54         cnt = 0
55         varList = self.varList()
56         for i in varList:
57             for j in self.disjunktions:
58                 for k in j.vars:
59                     if k.varname == i:
60                         k.value = value
```

```
61              value = random.choice([0,1])
62              cnt +=1
63
64      #Gibt alle nicht erfuellten Disjunktionen zurueck
65      def getFalse(self):
66          listOfFalse = []
67          l = 0
68          for i in self.disjunktions:
69              flag = False
70              for j in i.vars:
71                  if j.flag == True:
72                      if j.value == 1:
73                          flag = True
74                          break
75                  else:
76                      if j.value == 0:
77                          flag = True
78                          break
79              if flag == False:
80                  listOfFalse.append(i)
81          return listOfFalse
82
83      #Waehlt zufaellig eine nicht erfuellte Disjunktion
84      #Dann zufaellig eine Variable dieser
85      #Flippt diese Variable dann
86      def flipOne(self):
87          falseList = self.getFalse()
88          disj = random.choice(falseList)
89          var = random.choice(disj.vars)
90          for j in self.disjunktions:
91              for i in j.vars:
92                  if var.varname == i.varname:
93                      if i.value == 0:
94                          i.value = 1
95                      else:
96                          i.value = 0
97
98
99
100 def satSolver(sat):
101     sat.setValues()
102     for i in range(0,3*len(sat.varList())):
103         if(sat.isTrue()):
104             return True
105         sat.flipOne()
106     return False
107
108 #Gibt eine erfuellende Zuweisung wieder,
109 #sofern eine existiert
110 def getAssignment(sat):
111     varList = sat.varList()
112     varList.sort()
113
114     #Anzahl der Wiederholungen von SatSolver
115     t = int(20 * ((4/3)** len(varList)))
116     flag = True
117     isSat = False
118     assignment = []
119     for i in range(0, t):
120         if(satSolver(sat)):
121             for i in varList:
122                 flag = True
123                 for j in sat.disjunktions:
124                     for k in j.vars:
```

```
125                             if k.varname == i:
126                                 assignment.append(k.value)
127                                 flag = False
128                                 break
129                         if not flag:
130                             break
131                 return assignment
132         return assignment
133
134
135 if __name__ == "__main__":
136
137     #Definition einer Formel in 3KNF, wobei immer 3 Var
138     #eine Disjunktion wiedergeben
139     var = [Var('a',True),Var('b',True),Var('c',True),
140     Var('a',False),Var('e',False),Var('d',False)]
141     sat = Sat(var)
142
143     #Erhalte eine erfuellende Zuweisung oder eine
144     #leere Liste, falls keine Zuweisung existiert
145     assignment = getAssignment(sat)
146     if len(assignment) > 0:
147         varList = sat.varList()
148         varList.sort()
149         print(varList)
150         print(assignment)
151         print('is SAT')
152     else:
153         print('no SAT')
```

python3 Schoenings_Random_Walk_fuer_3SAT.py

Random Walk im P2P Netzwerk

Listing A.5: Random Walk im P2P Netzwerk

```
1  import random
2  import numpy
3  import math
4  import sys
5
6  #Prueft die Anzahl uebergebener Parameter
7  if len(sys.argv) != 3:
8      print('Es muessen genau zwei Argumente uebergeben werden!')
9      exit()
10
11 #Startknoten
12 start = str(sys.argv[1])
13 #Gesuchter Knoten
14 searched = str(sys.argv[2])
15
16 #Prueft, ob die Elemente aus list1 in list2 enthalten sind
17 def isIn(list1, list2):
18     for v in list1:
19         if v not in list2:
20             return False
21     return True
22
23 class Graph(object):
24
25     def __init__(self, graph_dict={}):
26         self.graph = graph_dict
27
28     #Gibt alle vertices des Graphen self wieder
29     def vertices(self):
30         return list(self.graph.keys())
31
32     #Gibt alle edges des Graphen self wieder
33     def edges(self):
34         edges = []
35         for vertex in self.graph:
36             for neighbour in self.graph[vertex]:
37                 edges.append([vertex, neighbour])
38         return edges
39
40     #Gibt alle Nachbarn des Knoten vertex im Graphen self wieder
41     def getNeighbours(self, vertex):
42         neighbours = []
43         for v in self.graph:
44             if vertex == v:
45                 for neighbour in self.graph[v]:
46                     neighbours.append(neighbour)
47         return neighbours
48
49     #Gibt einen zufaelligen Nachbarn des Knoten vertex
50     #im Graphen self wieder
51     def randomNeighbour(self, vertex):
52         step = random.choice(self.getNeighbours(vertex))
53         return step
54     #Random Walk auf einem Graphen durchfuehren,
55     #wobei self der Graph ist, acutal der Startknoten
56     #und searched der gesuchte Knoten
57     def randomSearch(self, actual, searched):
58         walk = []
59         vert = self.vertices()
60         walk.append(actual)
```

```
61          n = 0
62          while actual != searched and not isIn(vert, walk) :
63              old = actual
64              neigh = self.getNeighbours(old)
65              actual = self.randomNeighbour(old)
66              while actual in walk and (not isIn(neigh, walk)):
67                actual = self.randomNeighbour(old)
68              walk.append(actual)
69              n += 1
70          return walk
71
72  if __name__ == "__main__":
73      #Graph wird erstellt
74      g = { "p1" : ["p2","p7", "p13"],
75            "p2" : ["p1", "p4", "p10"],
76            "p3" : ["p4", "p5", "p6", "p8", "p12"],
77            "p4" : ["p2", "p3", "p5"],
78            "p5" : ["p3", "p4", "p8"],
79            "p6" : ["p3","p14"],
80            "p7" : ["p1", "p10"],
81            "p8" : ["p3", "p9", "p5"],
82            "p9" : ["p8", "p14"],
83            "p10" : ["p7", "p14", "p11", "p2"],
84            "p11" : ["p10"],
85            "p12" : ["p13", "p3"],
86            "p13" : ["p12", "p1"],
87            "p14" : ["p10", "p9", "p6"]
88          }
89
90      graph = Graph(g)
91
92      #Prueft, ob der Startknoten existiert
93      if not start in graph.vertices():
94          print("Unbekannter Start Knoten")
95          exit()
96
97      #Start des Random Walks
98      rw = graph.randomSearch(start, searched)
99      if rw[-1] == searched:
100         print(rw)
101     else:
102         print("Peer nicht gefunden")
```

Die Class Graph basiert auf der Class Graph aus [Kle].
Aufruf: python3 Random_Walk_im_P2P_Netzwerk.py <Startpeer> <gesuchter Peer>
Zum Beispiel: python3 Random_Walk_im_P2P_Netzwerk.py p2 p12

k-Random Walk

Listing A.6: k-Random Walk

```
1  from queue import Queue
2  from threading import Thread
3  import random
4  import numpy
5  import math
6  import time
7  import sys
8
9  #Prueft die Anzahl uebergebener Parameter
10 if len(sys.argv) != 4:
11     print('Es muessen genau drei Argumente uebergeben werden!')
12     exit()
13
14 #Das k im k-Random Walk
15 walker = int(sys.argv[1])
16 #Startknoten
17 start = str(sys.argv[2])
18 #Gesuchter Knoten
19 searched = str(sys.argv[3])
20 processes = []
21
22 #Prueft, ob die Elemente aus list1 in list2 enthalten sind
23 def isIn(list1, list2):
24   for v in list1:
25     if v not in list2:
26       return False
27   return True
28
29 #Startet verschiedene Threads mit der Function fun
30 def runInParallel (fun):
31   q = Queue()
32   global processes
33   for i in range(walker):
34     t = Thread(target = fun, args =(q,  ))
35     processes.append(t)
36   for p in processes:
37     p.start()
38
39 class Graph(object):
40
41     def __init__(self, graph_dict={}):
42         self.graph = graph_dict
43
44   #Gibt alle vertices des Graphen self wieder
45     def vertices(self):
46         return list(self.graph.keys())
47
48   #Gibt alle edges des Graphen self wieder
49     def edges(self):
50         edges = []
51         for vertex in self.graph:
52             for neighbour in self.graph[vertex]:
53                 edges.append([vertex, neighbour])
54         return edges
55
56   #Gibt alle Nachbarn des Knoten vertex im Graphen self wieder
57     def getNeighbours(self, vertex):
58         neighbours = []
59         for v in self.graph:
60             if vertex == v:
```

```
61                  for neighbour in self.graph[v]:
62                      neighbours.append(neighbour)
63          return neighbours
64
65    #Gibt einen zufaelligen Nachbarn des Knoten vertex
66    #im Graphen self wieder
67      def randomNeighbour(self, vertex):
68        step = random.choice(self.getNeighbours(vertex))
69        return step
70
71    #Random Walk auf einem Graphen durchfuehren, wobei self
72    #der Graph ist, q die Queue zur Kommunikation
73      def randomSearch(self, q):
74          global start, searched, processes
75          actual = start
76          walk = []
77          vert = self.vertices()
78          walk.append(actual)
79          n = 0
80          while actual != searched and not isIn(vert, walk) :
81              old = actual
82              neigh = self.getNeighbours(old)
83              actual = self.randomNeighbour(old)
84              while actual in walk and (not isIn(neigh, walk)):
85                actual = self.randomNeighbour(old)
86              walk.append(actual)
87              n += 1
88              time.sleep(0.001)
89              if not q.empty():
90                if q.get() == "done":
91                  return
92          for i in range(walker):
93            q.put("done")
94          if walk[-1]!=searched:
95            print ("Peer nicht gefunden")
96            return
97          print(walk)
98
99
100 if __name__ == "__main__":
101
102   #Graph wird erstellt
103   g = { "p1" : ["p2","p7", "p13"],
104           "p2" : ["p1", "p4", "p10"],
105           "p3" : ["p4", "p5", "p6", "p8", "p12"],
106           "p4" : ["p2", "p3", "p5"],
107           "p5" : ["p3", "p4", "p8"],
108           "p6" : ["p3","p14"],
109           "p7" : ["p1", "p10"],
110           "p8" : ["p3", "p9", "p5"],
111           "p9" : ["p8", "p14"],
112           "p10" : ["p7", "p14", "p11", "p2"],
113           "p11" : ["p10"],
114           "p12" : ["p13", "p3"],
115           "p13" : ["p12", "p1"],
116           "p14" : ["p10", "p9", "p6"]
117         }
118   graph = Graph(g)
119
120   #Prueft, ob der Startknoten existiert
121   if not start in graph.vertices():
122       print("Unbekannter Start Knoten")
123       exit()
124
```

```
125     #Start des k-Random Walks
126     runInParallel(graph.randomSearch)
```

Die Class Graph basiert auf der Class Graph aus [Kle].
Aufruf: python3 k-Random_Walk <k> <Startpeer> <gesuchter Peer>
Zum Beispiel: python3 k-Random_Walk.py 2 p2 p12

Alzheimer Random Walk

Listing A.7: Alzheimer Random Walk

```
1  import numpy
2  import pylab
3  import random
4  import sys
5
6  #Prueft die Anzahl uebergebener Parameter
7  if len(sys.argv) != 3:
8      print('Es muessen genau zwei Argumente uebergeben werden!')
9      exit()
10
11 #Ja/Nein Entscheidung mit der Wahrscheinlichkeit p fuer Ja
12 def yes_no(p):
13     return random.random() < p
14
15 #Markierwahrscheinlichkeit
16 p = float(sys.argv[1])
17
18 #prueft ob die Markierwahrscheinlichkeit 0 <= p <= 1
19 if p > float(1) or p < float(0):
20     print("0<=Markierwahrscheinlichkeit<=1")
21     exit()
22
23 #Anzahl der Schritte
24 n = int(sys.argv[2])
25
26 #Zwei Arrays, fuer x und y Achse
27 #Beide Arrays haben die Groesse n
28 #und sind mit 0en gefuellt
29 x = numpy.zeros(n)
30 y = numpy.zeros(n)
31 xmark = []
32 ymark = []
33 possible = []
34 visited = [[0 for x in range(n)] for y in range(n)]
35 posx = n // 2
36 posy = posx
37 if(yes_no(p)):
38     visited[posx][posy] = 1
39     ymark.append(0)
40     xmark.append(0)
41
42 #Wiederholungen des zufaelligen Richtungswechsels
43 for i in range(1, n):
44     possible = []
45     if posx + 1 < n:
46         if not visited[posx + 1][posy]:
47             possible.append('Right')
48     if posx > 0:
49         if not visited[posx - 1][posy]:
50             possible.append('Left')
51     if posy + 1 < n:
52         if not visited[posx][posy + 1]:
53             possible.append('Up')
54     if posy > 0:
55         if not visited[posx][posy - 1]:
56             possible.append('Down')
57     if len(possible) == 0:
58         print('Abbruch')
59         print('x: ')
60         print(x[i-1])
```

```
61            print('y: ')
62            print(y[i-1])
63            while i < n:
64                x[i] = x[i - 1]
65                y[i] = y[i - 1]
66                i+=1
67            break
68        step = random.choice(possible)
69        if step == 'Right' :
70            x[i] = x[i - 1] + 1
71            posx += 1
72            y[i] = y[i - 1]
73        elif step == 'Left':
74            x[i] = x[i - 1] - 1
75            posx -= 1
76            y[i] = y[i - 1]
77        elif step == 'Up':
78            x[i] = x[i - 1]
79            y[i] = y[i - 1] + 1
80            posy += 1
81        elif step == 'Down':
82            x[i] = x[i - 1]
83            y[i] = y[i - 1] - 1
84            posy -= 1
85        if(yes_no(p)):
86            visited[posx][posy] = 1
87            xmark.append(x[i])
88            ymark.append(y[i])
89
90
91    #Graph anzeigen lassen
92    pylab.title("Alheimer Random Walk ($n = " + str(n) + "$ Schritte)")
93    pylab.plot(x, y, xmark, ymark, 'ro')
94    pylab.show()
```

Aufruf: python3 Alzheimer_Random_Walk.py <markier Wahrscheinlichkeit> <Schrittanzahl>

k-Alzheimer Random Walk

Listing A.8: k-Alzheimer Random Walk

```
1  from queue import Queue
2  from threading import Thread
3  import random
4  import numpy
5  import math
6  import time
7  import sys
8
9  #Prueft die Anzahl uebergebener Parameter
10 if len(sys.argv) != 5:
11     print('Es muessen genau vier Argumente uebergeben werden!')
12     exit()
13
14 #Das k im k-Random Walk
15 walker = int(sys.argv[1])
16 #Startknoten
17 start = str(sys.argv[2])
18 #Gesuchter Knoten
19 searched = str(sys.argv[3])
20 #Markierwahrscheinlichkeit
21 p = float(sys.argv[4])
22
23 #prueft ob die Markierwahrscheinlichkeit 0 <= p <= 1
24 if p > float(1) or p < float(0):
25     print("0<=Markierwahrscheinlichkeit<=1")
26     exit()
27
28 processes = []
29
30 #Ja/Nein Entscheidung mit der Wahrscheinlichkeit p fuer Ja
31 def yes_no(p):
32     return random.random() < p
33
34 #Prueft, ob die Elemente aus list1 in list2 enthalten sind
35 def isIn(list1, list2):
36   for v in list1:
37     if v not in list2:
38       return False
39   return True
40
41 #Startet verschiedene Threads mit der Function fun
42 def runInParallel(fun):
43   q = Queue()
44   global processes
45   for i in range(walker):
46     t = Thread(target = fun, args =(q, ))
47     processes.append(t)
48   for p in processes:
49     p.start()
50
51 class Network(object):
52
53     def __init__(self, graph_dict={}):
54         self.graph = graph_dict
55
56     #Gibt alle Peers des Netzwerks self wieder
57     def vertices(self):
58         return list(self.graph.keys())
59
60     #Gibt alle Links des Netzwerks self wieder
```

```
61      def edges(self):
62          edges = []
63          for vertex in self.graph:
64              for neighbour in self.graph[vertex]:
65                  edges.append([vertex, neighbour])
66          return edges
67      #Gibt alle Nachbarn des Peers vertex im Netzwerk self wieder
68      def getNeighbours(self, vertex):
69          neighbours = []
70          for v in self.graph:
71              if vertex == v:
72                  for neighbour in self.graph[v]:
73                      neighbours.append(neighbour)
74          return neighbours
75
76      #Gibt einen zufaelligen Nachbarn des Peers vertex
77      #im Netzwerk self wieder
78      def randomNeighbour(self, vertex):
79          step = random.choice(self.getNeighbours(vertex))
80          return step;
81
82      #ARW im P2P Netzwerk durchfuehren, wobei self das Netzwerk ist,
83      #q die Queue zur Kommunikation
84      def randomSearchArw(self, q):
85          global start, searched, processes,p
86          actual = start
87          walk = []
88          marked = []
89          vert = self.vertices()
90          walk.append(actual)
91          if yes_no(p):
92            marked.append(actual)
93          n = 0
94          while actual != searched and not isIn(vert, walk) :
95              old = actual
96              neigh = self.getNeighbours(old)
97              actual = self.randomNeighbour(old)
98              while actual in marked and (not isIn(neigh, walk)):
99                actual = self.randomNeighbour(old)
100             walk.append(actual)
101             if yes_no(p):
102               marked.append(actual)
103             n += 1
104             time.sleep(0.004)
105             if not q.empty():
106               if q.get() == "done":
107                 return
108         for i in range(walker):
109           q.put("done")
110         if walk[-1] != searched:
111           print("Peer nicht gefunden")
112           return
113         print('ARW')
114         print(walk)
115
116 if __name__ == "__main__":
117   #Graph wird erstellt
118   g = {   "p1" : ["p11", "p13"],
119           "p2" : ["p20"],
120           "p3" : ["p5","p10","p14"],
121           "p4" : ["p15", "p19"],
122           "p5" : ["p3", "p12"],
123           "p6" : ["p14", "p17"],
124           "p7" : ["p11", "p16"],
```

```
125             "p8" : ["p15"],
126             "p9" : ["p11","p16","p18"],
127             "p10" : ["p3","p20"],
128             "p11" : ["p1","p7","p9","p16","p20"],
129             "p12" : ["p5","p15","p21"],
130             "p13" : ["p1","p14","p17"],
131             "p14" : ["p3","p6","p13","p17"],
132             "p15" : ["p4","p8","p12","p19"],
133             "p16" : ["p7","p9","p11"],
134             "p17" : ["p6","p13","p14"],
135             "p18" : ["p9","p20"],
136             "p19" : ["p4","p15"],
137             "p20" : ["p2","p10","p11","p18"],
138             "p21" : ["p12"]
139         }
140
141     network = Network(g)
142
143     #Prueft, ob der Startknoten existiert
144     if not start in network.vertices():
145       print("Unbekannter Start Knoten")
146       exit()
147
148     #Start des k-Alzheimer Random Walks
149     runInParallel(network.randomSearchArw)
```

Die Class Network basiert auf der Class Graph aus [Kle].
Aufruf: python3 k-Alzheimer_Random_Walk <k> <Startpeer> <gesuchter Peer> <Markierwahrscheinlichkeit>
Zum Beispiel: python3 k-Alzheimer_Random_Walk.py 2 p2 p12 0.8

maximum-degree Random Walk

Listing A.9: maximum-degree Random Walk

```
1  import random
2  import numpy as np
3  import scipy.linalg as la
4  import math
5
6  #Klasse eines ungerichteten Graphen
7  class UndirGraph(object):
8      #initialisiert den Graphen
9      def __init__(self, graph_dict={}):
10         self.graph = graph_dict
11
12     #Gibt alle Knoten des Graphen wieder
13     def vertices(self):
14         return list(self.graph.keys())
15
16     #Erstellt eine Uebergangsmatrix des Graphen
17     def adjMatrix(self):
18         matrix = []
19         for node in self.vertices():
20             nodeAdj = []
21             neighbours = self.getNeighbours(node)
22             degree = len(neighbours)
23             for testNode in self.vertices():
24                 if testNode in neighbours:
25                     nodeAdj.append(1/degree*neighbours.count(testNode))
26                 else:
27                     nodeAdj.append(0)
28             matrix.append(nodeAdj)
29         return np.array(matrix)
30
31     #Gibt die Nachbarn eines Knoten vertex wieder
32     def getNeighbours(self, vertex):
33         neighbours = []
34         for neighbour in self.graph[vertex][0]:
35             neighbours.append(neighbour)
36         return neighbours
37
38     #Gibt einen zufaelligen Nachbarn eines Knoten vertex wieder
39     def randomNeighbour(self, vertex):
40         step = random.choice(self.getNeighbours(vertex))
41         return step;
42
43     #Bestimmt die Anzahl an notwendigen Schritten fuer eine Verteilung
44     #nahe der Gleichverteilung
45     def steps(self, errorRate):
46         vertices = len(self.vertices())
47         pmin = 1/vertices
48         adjMatrix = self.adjMatrix()
49         eigenvalues = la.eig(adjMatrix)[0].real
50         eigenvalues.sort()
51         eigenvalues = eigenvalues[::-1]
52         if eigenvalues[0]>=1:
53             maxEigval = eigenvalues[1]
54         else:
55             maxEigval = eigenvalues[0]
56         spekGap = 1 - maxEigval
57         trel = 1/spekGap
58         return int(math.log(1/(errorRate * pmin)) * trel)
59
60
```

```
61
62 #Klasse eines gerichteten Graphen
63 class DirGraph(object):
64
65     #Initialisiert den Graphen
66     def __init__(self, graph_dict={}):
67         self.graph = graph_dict
68
69     #Gibt alle Knoten des Graphen wieder
70     def vertices(self):
71         return list(self.graph.keys())
72
73     #Gibt die Nachbarn wieder, auf die ein Knoten vertex verweist
74     def getOutEdgeNeighbours(self, vertex):
75         neighbours = []
76         for neighbour in self.graph[vertex][0]:
77             neighbours.append(neighbour)
78         return neighbours
79
80     #Gibt die Nachbarn wieder, die auf einen Knoten vertex verweisen
81     def getInEdgeNeighbours(self, vertex):
82         neighbours = []
83         for neighbour in self.graph[vertex][1]:
84             neighbours.append(neighbour)
85         return neighbours
86
87     #Gibt die Nachbarn eines Knoten vertex wieder
88     def getNeighbours(self, vertex):
89         if self.graph[vertex][1]:
90             neighbours = self.getInEdgeNeighbours(vertex)
91                         + self.getOutEdgeNeighbours(vertex)
92         else:
93             neighbours = self.getInEdgeNeighbours(vertex)
94         return neighbours
95
96     #Berechnet den maximalen Grad des Graphen
97     def maxDegree(self):
98         d = 0
99         for i in self.vertices():
100            degree = len(self.getNeighbours(i))
101            if degree > d:
102                d = degree
103        return d
104
105
106 #Macht aus einem Graphen einen ungerichteten regulaeren Graphen
107 def makeRegUndGraph(dirGraph):
108     d = dirGraph.maxDegree()
109     graph = dirGraph
110     for node in graph.graph.keys():
111         if dirGraph.__class__ == DirGraph:
112             graph.graph[node][0] += graph.graph[node][1]
113             graph.graph[node].pop()
114         while len(graph.graph[node][0]) < d:
115             graph.graph[node][0].append(node)
116     graph.__class__ = UndirGraph
117     return graph
118
119 #Fuehrt den maximum-degree Random Walk aus
120 def maxDegreeRW(DirGraph, node):
121     g = makeRegUndGraph(DirGraph)
122     tmix = g.steps(0.0002)
123     walk = []
124     walk.append(node)
```

```
125         oldnode = node
126         for i in range(0, tmix):
127             node = g.randomNeighbour(node)
128             if node != oldnode:
129                 walk.append(node)
130                 oldnode = node
131         return walk
132 
133 
134 
135 
136 if __name__ == "__main__":
137         #Ungerichteter Graph mit
138         #"node" : [[ausgehende Kanten],[eingehende Kanten]]
139         g = { "p1" : [["p4","p3"], ["p2"]],
140               "p2" : [["p1","p3"], ["p5"]],
141               "p3" : [["p5"], ["p1","p2","p4"]],
142               "p4" : [["p3","p6"], ["p1"]],
143               "p5" : [["p2"], ["p3","p6"]],
144               "p6" : [["p5", "p7"], ["p4"]],
145               "p7" : [[], ["p6"]]
146             }
147         graph = DirGraph(g)
148         print(maxDegreeRW(graph,"p1"))
```

Die Class UndirGraph und die Class DirGraph basieren auf der Class Graph aus [Kle].
Aufruf: python3 maximum-degree_Random_Walk.py