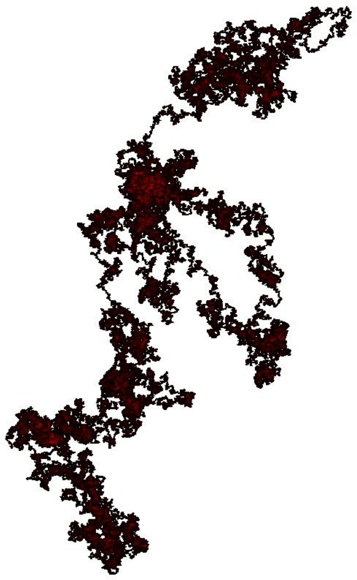
Once-edge reinforced random walk

Goorden, S.A.

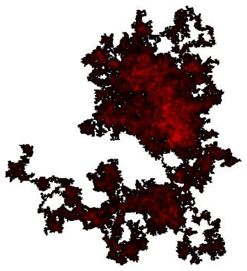
Award date:
2009

Random walks met voorkeur voor bekende paden

Bas Goorden
Technische Wiskunde, TU/e Eindhoven, 2008-2009


Voorbeeld van een simple random walk. Feller rood betekent dat het proces daar vaker geweest is.

Een random walk is een pad dat het gevolg is van een serie opeenvolgende willekeurige stappen. Veel dingen die we in de wereld zien, kunnen verklaard worden door ze te bekijken als een random walk. Een paar voorbeelden van zulke dingen zijn het pad van een molecuul dat door een vloeistof of gas beweegt en het pad dat een naar eten zoekend dier aflegt. Een vaak voorkomende manier om zulke dingen wiskundig te beschrijven, is met behulp van een zogenaamde simple random walk. Een simple random walk vindt plaats op een rooster en de kans om in elk van de mogelijke richtingen te bewegen is altijd gelijk. Het komt echter regelmatig voor dat de beslissing over de richting van de volgende stap afhangt van de geschiedenis van het pad tot dan toe. Een voorbeeld is de situatie waarin een persoon door een stad loopt, waarbij zij een voorkeur heeft voor routes die ze al kent. Een ander voorbeeld is een bacterie die het liefst over slijm glijdt dat ze eerder geproduceerd heeft. Een manier om zulk gedrag te beschrijven, is met de Once-Edge Reinforced Random Walk (OERRW), en die heb ik onderzocht. Op een eenvoudige manier gezegd, betekent 'Edge Reinforced' dat het proces liever over paden gaat waar het eerder al langsgekomen is dan over onbekende paden. 'Once' wil zeggen dat het niet uitmaakt of het proces één of honderd keer langs een pad gegaan is: als het bekend is dan is het bekend. Eén van de dingen die we willen weten is hoe ver het proces van het startpunt weggaat. Ik heb dit bekeken voor verschillende voorkeuren voor bekende paden. Iets anders waar ik naar gekeken heb is de invloed van de geschiedenis van het pad op de richting waarin het verder zal gaan. De manier waarop ik dit gedaan heb, is door te zoeken naar de geschiedenis die het het meest waarschijnlijk maakt dat je van het startpunt naar een vooraf vastgesteld eindpunt gaat. Deze dingen blijken alleen goed onderzocht te kunnen worden door ze met de computer na te doen, wat ook wel simuleren genoemd wordt. Daarom was het belangrijk om een goed simulatieprogramma te schrijven, waarmee ik veel simulaties gedaan heb. Eén van de dingen die ik ontdekt heb, gaat over de OERRW in twee dimensies. Je kunt hierbij denken aan een random walk in een plat vlak, zoals bij een persoon die rondloopt in een stad. Voor een simple random walk, zowel in twee als in drie dimensies, is het bekend dat de verwachte afstand tot het startpunt groeit als de wortel van de lengte van de random walk. Voor een OERRW in twee dimensies blijkt dat langzamer te gaan. Voor random walks in drie dimensies lijkt het nog anders te zijn. Bij drie dimensies is ook de 'hoogte' van belang, zoals bij een molecuul dat in de lucht rondzweeft. Als er in drie dimensies een kleine voorkeur voor bekende paden is, gaat de verwachte verplaatsing wél als de wortel van het aantal stappen. Aan de andere kant groeit bij een grote voorkeur voor bekende paden de verwachte verplaatsing langzamer dan de wortel van het aantal stappen, net zoals wanneer er maar twee dimensies zijn.

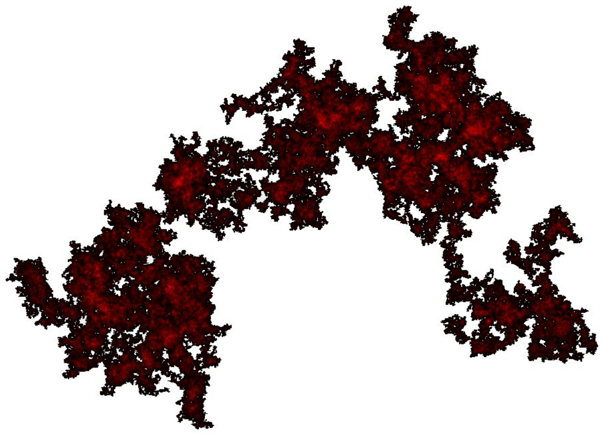

Voorbeeld van een OERRW. Feller rood betekent dat het proces daar vaker geweest is.

Bachelor Thesis

Once-Edge Reinforced Random Walk

Author:
S.A. Goorden

Supervisor:
Prof. Dr. R.W. van der Hofstad



July 21, 2009

Abstract

This report deals with the Once-Edge Reinforced Random Walk (OERRW). In this type of random walk, edges that have not been traversed have the basic weight of 1, while edges that have been traversed before get a weight of $1 + \beta$. Usually $\beta > 0$, and then this implies that edges that are 'known' are more likely to be traversed again than edges that are 'unknown'.

There are two questions that we try to answer. The first one is how the expected displacement of the process can be described. Does it move away as $\sqrt{n}$, like the simple random walk? If not, then what does happen? The second question is which initial history should be chosen if the goal is to move from a point X_0 to a point x in n steps.

Simulations have been done in order to help us answer these questions. A first answer to the first question is that in all cases the expected displacement after N steps can be described by $c\ n^{\alpha}$. We see that α is non-increasing as β is increasing. We also see that α is non-decreasing as the number of dimensions increases. The two-dimensional case seems to be fundamentally different from the higher dimensional cases. When the number of dimensions is two, $\alpha < \frac{1}{2}$ when $\beta > 0$. When the number of dimensions is greater than two, there seems to be a positive critical value of β up to which $\alpha = \frac{1}{2}$. When β is greater than this critical value we see that $\alpha < \frac{1}{2}$ again.

In case one wants to maximize the probability of reaching x after an infinite time with $\beta = \infty$, the optimal history is any L1-path from X_0 to x supplemented with all edges adjacent to x. Another way of looking at the second question is by trying to minimize the expected distance to x. For $\beta = \infty$ and infinitely long random walks a different optimal history is obtained. In this case, one should take the union of any L1-path from X_0 to x and an L1-ball around x of a particular radius. This radius depends on the distance from X_0 to x. This is also suspected to be optimal when the random walk has finite length. When $\beta < \infty$ simulations show that histories with a larger L1-ball around x are better. There are strong indications that as β decreases, the radius of the optimal L1-ball increases and that as n increases, the radius of the optimal L1-ball also increases. There is no reason to assume that another type of history would be better.

Contents

Chapter 1

Introduction

A random walk is a trajectory that results from taking consecutive random steps. Many phenomena that we observe in the real world can be studied by modelling them as a random walk. Some examples of such phenomena are the path of a molecule as it travels through a liquid or gas, the price of a fluctuating stock and the search path of a foraging animal.

In many applications the decision where to step depends on the history of the path up to that point. An example is a person walking around in a city, who prefers routes she already knows. Another is a bacterium that prefers to slide over the slime that it produced earlier. A way to model this kind of behaviour is by the Once-Edge Reinforced Random Walk (OERRW), which is the kind of random walk that will be investigated in this project.

Without getting into too much detail just yet, 'Edge Reinforced' means that known routes have a higher probability of being traversed again than unknown ones and 'Once' means that it does not matter whether a route has been traversed just once or a hundred times: when you know it, you know it.

One thing of interest is how far the process will move from the origin, in different dimensions and with varying preference for known routes. Another thing we will look into is the influence of the history of the process on the direction in which it will develop. A way to do this is by trying to find the history which makes it most likely to get from the starting point to a predefined endpoint. The possibilities of investigating these matters analytically are limited. Therefore, developing a simulation program with which we can study the behaviour of the OERRW is also an important part of this project.

Chapter 2

Model

2.1 The general model

First, the model of the OERRW will be described. The process will take place in a d-dimensional space and it will be discrete in both space and time. In a natural way we obtain the sequence

$$(X_n)_{n\in\mathbb{N}} \subset \mathbb{Z}^d$$

where X_n denotes the position of the process at time n.

Now the stepping rules will be addressed. At each time the process can only move from one point to a neighbour of that point. Let $\sim$ denote the relation of being neighbours. Points are neighbours exactly when the distance between them is one, so if $i, j \in \mathbb{Z}^d$, then

$$i \sim j \Leftrightarrow \|i - j\|_1 = 1.$$

If $i \sim j$ then $\{i, j\}$ denotes the edge between i and j. Edges have no direction, so $\{j, i\} = \{i, j\}$.

In principle all information about the process up to time n is known when $X_i, 0 \leqslant i \leqslant n$ are known. It is convenient, though, to define a history, $(\eta_n)_{n\in\mathbb{N}}$, as follows:

$$\eta_n := \{\{X_i, X_{i+1}\} : 0 \leqslant i \leqslant n-1\} \cup \eta. \tag{2.1}$$

Now η_n is the union of the collection of edges that have been traversed up to time n and η. η is defined as the collection of edges that we already 'know' at $n = 0$ and usually η will be empty. We see that $\eta_0 = \eta$.

It is now time to describe how X_n and η_n depend on X_i and η_i, $0 \leqslant i \leqslant n-1$. To start, X_n is one of the neighbours of X_{n-1}. Which one it will be is random and only depends

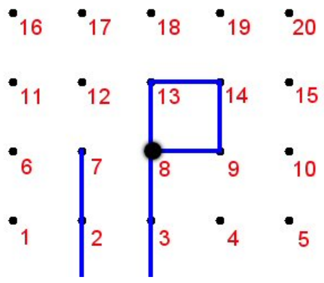


Figure 2.1: Visualization of a situation that could occur during a random walk.

on whether the edge that leads to each neighbour is 'known'. We differentiate between unknown and known edges by giving unknown ones a relative weight of 1 and known ones a weight of $1+\beta$. This $0 \leqslant \beta \leqslant \infty$ is the parameter of the model and the practical meaning of it is that a bigger β results in a stronger preference for known edges. In a formula:

$$P_{ij}^{\eta_n} = \mathbb{P}(X_{n+1} = j | X_n = i, \eta_n) = \frac{\mathbf{1}\{i \sim j\} + \beta \mathbf{1}\{\{i,j\} \in \eta_n\}}{\sum_{k \sim i}(1 + \beta \mathbf{1}\{\{i,k\} \in \eta_n\})}, \tag{2.2}$$

so the probability of traversing a particular edge is equal to the weight of that edge divided by the sum of the weights of all possible edges.

An example can be found in figure 2.1. In this situation, $X_n = 8$ (the big black dot). The neighbours of X_n are 3, 7, 9 and 13, so $X_{n+1} \in \{3, 7, 9, 13\}$. A blue line means that that edge is in the history. Suppose $\beta = 1$, then the weight of $\{8, 7\}$ equals 1 and the weights of $\{8, 3\}, \{8, 9\}$ and $\{8, 13\}$ equal 2. This means that $\mathbb{P}(X_{n+1} = 7) = \frac{1}{7}$ and $\mathbb{P}(X_{n+1} = 3) = \mathbb{P}(X_{n+1} = 9) = \mathbb{P}(X_{n+1} = 13) = \frac{2}{7}$.

Note that, by (2.1),

$$\eta_{n+1} = \eta_n \cup \{X_n, X_{n+1}\}. \tag{2.3}$$

In general the distribution of X_{n+1} depends on η_n and therefore on all of $X_i, 0 \leqslant i \leqslant n$, so X_n is not a Markov chain. It is possible to define $Y_n = (X_n, \eta_n)$, though, and then Y_n is a Markov chain, albeit in a rather complicated state space. We will not go into that.

2.2 Special cases

When $\beta = 0$ known edges will have the same weight as unknown ones. In this case we have

$$P_{ij} = \mathbb{P}(X_{n+1} = j | X_n = i) = \frac{\mathbf{1}\{i \sim j\}}{\sum_{k \sim i} 1} = \frac{\mathbf{1}\{i \sim j\}}{2d}, \tag{2.4}$$

which exactly describes an ordinary random walk in d dimensions. Note that in this case there is no reason to keep track of a history, so that $\{X_n\}_{n\geqslant 0}$ now is a Markov chain.

The other, slightly more complicated, special case is $\beta = \infty$. In this case, by dividing by β and taking the limit of β to infinity in (2.2), we get:

$$P_{ij}^{\eta_n} = \mathbb{P}(X_{n+1} = j | X_n = i, \eta_n) = \begin{cases} \frac{\mathbf{1}\{\{i,j\}\in\eta_n\}}{\sum_{k\sim i}\mathbf{1}\{\{i,k\}\in\eta_n\}}, & \text{if } \exists_{k\sim i} : \{i,k\} \in \eta_n, \\ \frac{\mathbf{1}\{i\sim j\}}{2d}, & \text{if } \neg\exists_{k\sim i} : \{i,k\} \in \eta_n. \end{cases} \tag{2.5}$$

In plain English, we see that if it is possible to traverse a known edge, then the probability of traversing an unknown edge equals zero and the probability of traversing each known edge equals one over the number of traversable known edges. If it is not possible to traverse a known edge, then the probability of traversing each of the $2d$ unknown edges equals one over $2d$. We will analyse this case more closely in section 5.1.

Chapter 3

Simulation

3.1 Introduction

It turns out that analysing the OERRW is difficult for $0 < \beta < \infty$, which is why we have developed a simulation program. This was a major part of the project and a lot of the results has been obtained from it, so in this chapter we will give an explanation of the program. The program has been written in Java and can be found on www.student.tue.nl/R/s.a.goorden/OERRW.

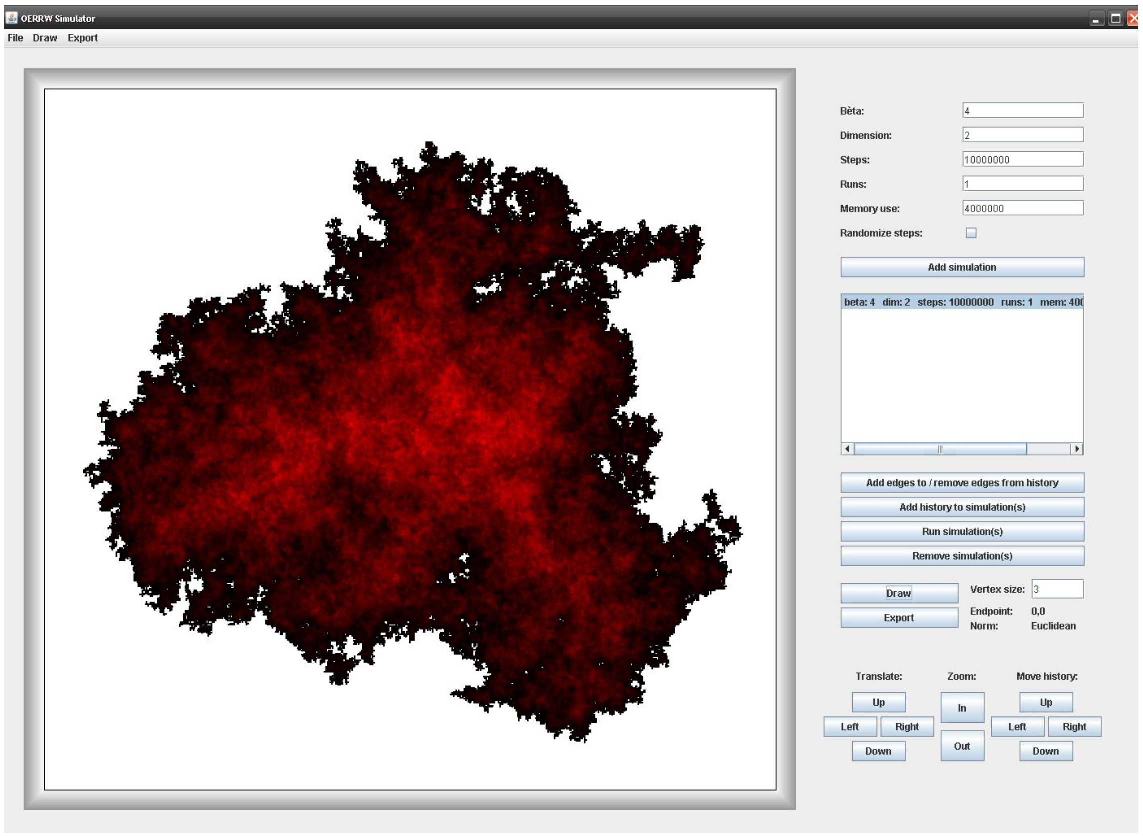
Section 3.2 will provide the general idea of what the program does. An explanation of how the program can be used can be found in section 3.3. We refer the interested reader to appendix A for a more detailed view on the structure of the program and to appendix B for an explanation on the (quite unique) way this program stores and finds data.

3.2 Purpose of the program

The goal was to develop a program that can simulate random walks. The obvious way of simulating a random walk is, essentially, by:

1. looking at the point where the random walk currently is,
2. calculating where it should go next,
3. moving to that point

and repeating this process. For the random walk to start, a starting point $X_0 = \mathbf{0}$ has to be defined. The hardest part is in calculating where to go next. This is done by using a uniform random variable in $(0, 1)$ and the transition probabilities for the OERRW, which we have defined in equation (2.2). In order to use that equation, the program must know



Bèta: 4
Dimension: 2
Steps: 10000000
Runs: 1
Memory use: 4000000
Randomize steps: ☐

beta: 4 dim: 2 steps: 10000000 runs: 1 mem: 400

Vertex size: 3
Endpoint: 0,0
Norm: Euclidean

Figure 3.1: Visualization of the program. A 10.000.000 step random walk with $\beta = 4$ is shown.

β and the history. A value of β and an initial history have to be entered by the user and the program must update the history every time it goes through step 3. Since the program needs to know when to stop, the user can enter the number of steps, which we denote by n.

The process described in the previous paragraph is the simulation of one random walk of n steps. We call this a run. What we call ‘simulation’ in the program will usually consist of several runs. This is necessary to get a good approximation of the average behavior. The user can specify the number of runs. When there are multiple runs, only the endpoint of each run will be saved. When there is just one run, all points that have been visited will be saved.

The program is able to produce a visualization of the data. When there are multiple runs, it shows all the endpoints. When there is just one run, it shows all the points that have been visited (see figure 3.1). At least as important is that the data can be analysed. We have chosen not to integrate that into this program. Instead the program can export the data so that it can be analysed using Matlab or Mathematica. The data that is exported consists of the distances of the endpoints of the runs to a point x. The user can select x and choose whether a Euclidean or an L1 norm should be used.

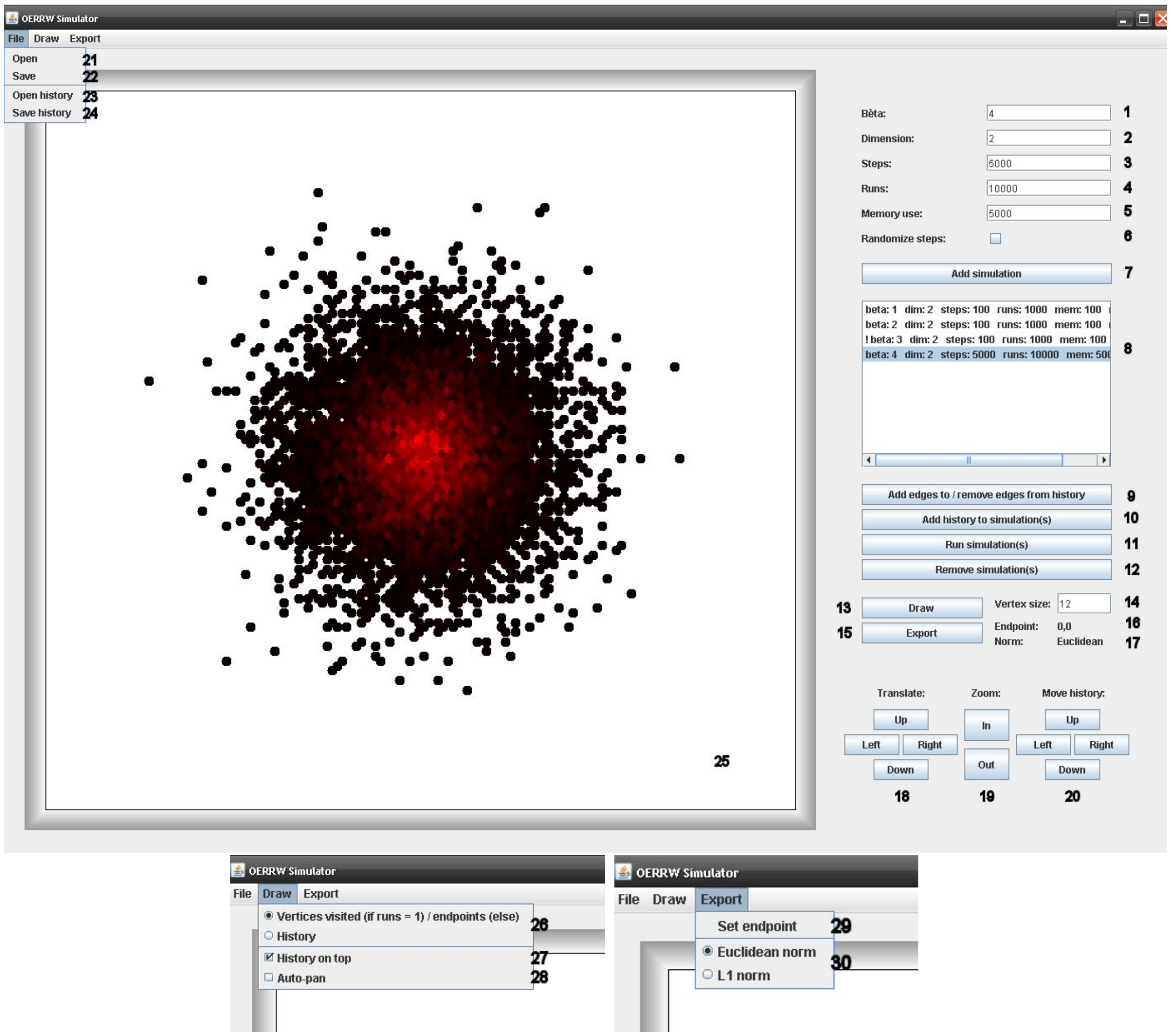
The program can hold multiple simulations (usually with different parameters) at the same time, so that they can be compared more easily. It is possible to save and open simulations, so that one does not have to do the same simulations repeatedly. It is also possible to save and open initial histories, which can then be inserted into simulations.

3.3 Using the program

3.3.1 The interface

In this section we will explain how the program functions. We will start by explaining the interface, which can be found in figure 3.2 with numbers near all the different items.

1. The value of β must be entered here. This value must be larger than -1. One can do simulation with $\beta = \infty$ by entering 'inf'.
2. The number of dimensions must be entered here. This must be an integer $\geqslant 1$.
3. The number of steps must be entered here.
4. The number of runs must be entered here.
5. The number of memory spots that the program uses to store vertices must be entered here. It is advisable to keep this equal to the number of steps. For a more detailed explanation, see appendix B.
6. This box can be checked if one wants to mitigate the periodicity of the random walks. If this is checked, for each run there is a 25% probability that it has one extra step and a 25% probability that it has one step less.
7. When this button is pressed, a new simulation is added with the parameters entered at 1-6.
8. All the different simulations in the program are shown in this information box with their parameters. An exclamation mark means that the simulation has not yet been run (see point 11).
9. When this button is toggled, edges can be added or removed to the history. A detailed explanation of histories will follow in section 3.3.3.
10. When this button is pressed, the current general working history is added to the simulations that are selected in the information box. If the simulations had a history before, that history is lost. A detailed explanation of histories will follow in section 3.3.3.



Bèta: 4
Dimension: 2
Steps: 5000
Runs: 10000
Memory use: 5000
Randomize steps: ☐

```
beta: 1   dim: 2   steps: 100   runs: 1000   mem: 100
beta: 2   dim: 2   steps: 100   runs: 1000   mem: 100
! beta: 3   dim: 2   steps: 100   runs: 1000   mem: 100
beta: 4   dim: 2   steps: 5000   runs: 10000   mem: 500
```

Vertex size: 12
Endpoint: 0,0
Norm: Euclidean

(●) Vertices visited (if runs = 1) / endpoints (else)
( ) History
☑ History on top
☐ Auto-pan

(●) Euclidean norm
( ) L1 norm

Figure 3.2: Visualization of the program. The endpoints of 10.000 runs of 5.000 steps each with $\beta = 4$ are shown.

11. When this button is pressed, the simulations that are selected in the information box are (re)run. This is when the actual simulations happen.

12. When this button is pressed, the simulations that are selected in the information box are removed.

13. When this button is pressed, a visualization will be made in the visualization box (see 25). If 'Vertices visited / endpoints' is selected (see 26), then the simulation that is selected in the information box is visualized. If this simulation consists of 1 run, then the entire run is shown. If the simulation consists of multiple runs, only the endpoints will be shown. If, on the other hand, 'History' is selected (see 26), then a history will be visualized. If a simulation is selected in the information box, then the history corresponding to that simulation will be shown. If no simulation is selected, then the general working history (see 3.3.3) will be shown.

14. The size of the vertices in the visualization box (see 25) can be set here. This value must be an integer of value at least 2 and at most 16. It is advised to try different values to see what looks best.

15. When this button is pressed the data of the simulations that are selected in the information box is exported. This data usually consists of the distances (either Euclidean or L1, see 30) of the endpoints of all the runs to some selected endpoint (see 29). If a simulation consists of just 1 run, the data consists of the distances of all the points that have been visited to the endpoint (this is not as interesting). The data is printed to a text file, in which there is a line for each simulation that is selected. On the line belonging to a simulation, all the distances are printed, separated by tabs. On top of this, an additional text file is created in which one can find the parameters of the simulations to which the data belongs as well as the norm and the endpoint that were used. Both files are stored in the 'Data' folder, which can be found in the same folder as the program itself. In this format the data can easily be analysed, using, for example, Matlab.

16. The selected endpoint is shown here (see 29). This is the point to which the distance of the endpoints of the runs is measured when those distances are exported.

17. The norm of the distances from the endpoints of the runs to the endpoint is shown here. This can be either the Euclidean or the L1 norm (see 30).

18. These buttons are used to navigate in the visualization box.

19. These buttons are used to zoom in and out in the visualization box.

20. These buttons are used to move the 'history point' (see section 3.3.3). An alternative to these buttons are the buttons on the keyboard.

21. A file containing simulations that has been saved earlier (see 22) can be opened here.

22. When this button is pressed, all the simulations that are selected in the information box are saved into one file. When this is opened later, everything functions just like before it was saved. The file is stored in the ‘Simulations’ folder, which can be found in the same folder as the program itself.
23. A general working history that has been saved earlier (see 24) can be opened here. If ‘History’ is selected (see 26) and no simulations are selected in the information box, it will automatically be drawn.
24. The general working history (see section 3.3.3) can be saved here. These are stored in the ‘Histories’ folder, which can be found in the same folder as the program itself.
25. This is the visualization box. Things will be visualized here when the ‘Draw’ button (see 13) is pressed.
26. If ‘Vertices visited / endpoints’ is selected here and the ‘Draw’ button (see 13) is pressed, the results of the simulation will be visualized. If the simulation consists of only 1 run, then the entire path will be shown. Otherwise, only the endpoints of each of the runs will be shown. If ‘History’ is selected here and the ‘Draw’ button (see 13) is pressed, a history will be visualized. If a simulation is selected in the information box, then the history corresponding to that simulation will be shown, otherwise the general working history (see 3.3.3) will be shown.
27. If this box is checked, ‘Vertices visited / endpoints’ is selected (see 26) and the ‘Draw’ button (see 13) is pressed, then the edges that are in the history corresponding to the simulation that is selected in the information box are drawn on top of the points.
28. If this box is checked, ‘Vertices visited / endpoints’ is selected (see 26) and the ‘Draw’ button (see 13) is pressed, then the program automatically puts the center of the image at the center of the visualization box and zooms in or out so that the image fits nicely. This has been used, for example, in figure 3.1. It is especially useful when one long run is simulated for a small value of β.
29. When this button is pressed, the current position of the ‘history point’ *of the general working history* (see 3.3.3) becomes the ‘endpoint’. Indeed, the endpoint must be in the xy-plane, which is a limitation of the program. At least for our research, it never needs to be outside of this plane, though.
30. Here, the norm that is used when exporting (to calculate the distance from the endpoints of the runs to the ‘endpoint’, see 15) is used. This can be either the Euclidean or the L1-norm.

3.3.2 Advice on the parameters

The user will often be working close to the limits of the computer that the program is running on. In order to prevent errors, we have considered imposing limitations for the parameters through the software. The reason for not doing this, is that it is hard to do properly and that we would limit the possibilities of the program too much. Instead, we will briefly explain what can go wrong and how to prevent it.

The biggest risk is that the program needs to use too much memory. When this happens, the program hangs. This usually happens when simulations are run and the program needs to store very large numbers of points. It is also possible, that the program gets close to the memory limit when simulating, but doesn't hang before it has to draw or export.

A big part of the 'problem' is in the number of points that the program needs to store. Within a run, the program needs to know which edges have been traversed. In order to keep track of this, all of the points that have been visited are stored (in the program a 'point' includes its adjacent edges, for details see appendix A). Things go wrong when the program is simulating a run in which too many points are visited. The number of points that are visited is clearly higher when the number of steps is higher, but we also notice that it is higher when β is smaller. On the laptop we are using (4 years old, 1 GB RAM memory), in two dimensions, for $\beta = 1$ a random walk of about 2.5 million steps is the maximum. For $\beta = 4$, we see that we can do random walks of over 10 million steps. In the latter case, it is necessary to enter a value of only 4 million for the memory use parameter, instead of the 10 million we would normally do. This is not a problem, because for this β much less than 10 million points need to be stored.

In three dimensions, for $\beta = 1$ we see that the maximum length goes down to about 0.5 million steps. Points in more dimensions cost a bit more space to store, but moreover we create a second set of data which holds the projection of the data onto the two-dimensional plane. This is done because we want to be able to visualize the data, but the drawback is that it costs quite a bit of memory space. The same applies to the higher dimensional cases. The maximum number of steps decreases a bit more, but not dramatically.

For our research we are more interested in doing large numbers of shorter random walks. It turns out that this is not a problem, memory wise. The reason for that is that only the endpoint of each run is stored 'permanently'. The points that have been visited during a run are all removed from the memory when the next run starts. When doing this kind of simulations, time becomes the problem.

Unfortunately the program doesn't have a feature that predicts how long a simulation will take. It is not very difficult to predict this though. It won't come as a surprise that the time that the program takes linearly depends on the number of runs. What is nice though, is that because of the way the memory is managed the time also depends linearly on the number of steps. This means that one can first try some small numbers of runs and steps and see how long the simulation takes. If it takes, say, 6 seconds, one can then multiply the numbers of runs and steps by 10 and the simulation will then take about 600 seconds or 10 minutes. The time that it takes does not depend much on β and the number of dimensions, although it is slightly faster for larger β and slightly slower for the higher dimensional cases.

3.3.3 Histories

By 'history' here we mean the collection of edges that are 'known' before the random walk starts. Outside the program this is denoted by η. The way the program handles histories is not very obvious, because everything that has to do with the 'general working history' has been added later. In this section we will explain how to work with histories.

It is very important to realize that there are two different types of histories in the program, namely simulation histories and the general working history. We will now discuss both:

1. *Simulation history:* Every simulation in the program has its own history. This history is what the simulation uses when it is run. The history belonging to one particular simulation can be visualized by selecting the simulation in the information box, making sure that 'History' is selected in the 'Draw' menu and pressing the 'Draw' button.

2. *General working history:* There is also one general working history in the program. This history is not directly used by the simulations. When the button 'Add history to simulation(s)' is pressed, the simulation histories of the simulations that are currently selected are replaced by a copy of the general working history. This history is especially useful when one wants to add the same history or almost the same histories to several simulations. This is also the only history that can be saved. This history can be visualized by making sure that no simulation is selected in the information box, making sure that 'History' is selected in the 'Draw' menu and pressing the 'Draw' button. Tip: if one has trouble deselecting simulations, one can simply add and immediately remove a simulation.

Both types of history can be altered by first visualizing them as described above. One will notice that there is a big blue dot, which we call the 'history point'. The history point can be moved around by pressing the 'history move' arrows on the interface or by pressing the arrows on the keyboard. Now, if one wants to add edges to or remove edges from the history, one should press the 'Add edges to / remove edges from history' toggle button. As long as this button is toggled on, all of the edges that are traversed by the history point are 'flipped': if the edge was not in the history, then it is added to the history and if it already was in the history, then it is removed from the history. Of course, when one wants to move to another point without flipping edges, one can press the 'Add edges to / remove edges from history' button again to toggle it off.

Chapter 4

OERRW without initial history

4.1 $\beta = 0$

In this section the case $\beta = 0$ will be mentioned. As noted before, histories will not play any role now. This type of random walk is called a simple random walk and a lot is already known about that. A very important result follows directly from the multivariate central limit theorem [1]. In our case the (M)CLT implies that the position of the random walk after n steps, denoted by X_n, has a multivariate normal distribution. The mean of this distribution is $\mathbf{0}$ and the covariance matrix is equal to the identity matrix. This means that $\mathbb{E}[|X_n|] \sim \sqrt{n}$.

An example of a simple random walk in 2 dimensions can be found in figure 4.1. Because we are interested in the expected displacement after n steps, a figure of the endpoints of 50.000 simple random walks of 1.000 steps has also been included. These pictures are just meant for showing what happens, but it is still worth noting that it looks like the endpoints might, indeed, be normally distributed.

4.2 2D

It is now time to look at what happens when $\beta > 0$. We will first focus on the two-dimensional case, later we will also consider the one-dimensional and higher-dimensional cases. Pictures of the cases $\beta = 2$ and $\beta = 5$, still in 2 dimensions, can be found in figure 4.2. We know that when $\beta > 0$, known edges are preferred over unknown ones and that this effect gets stronger as the value of β increases. It was expected that this means that the process does not move away from the starting point as much, and these pictures seem to confirm that.

Because the individual steps of which the random walk consists are not independent when $\beta > 0$, the multivariate central limit theorem does not apply now. It is also hard to come

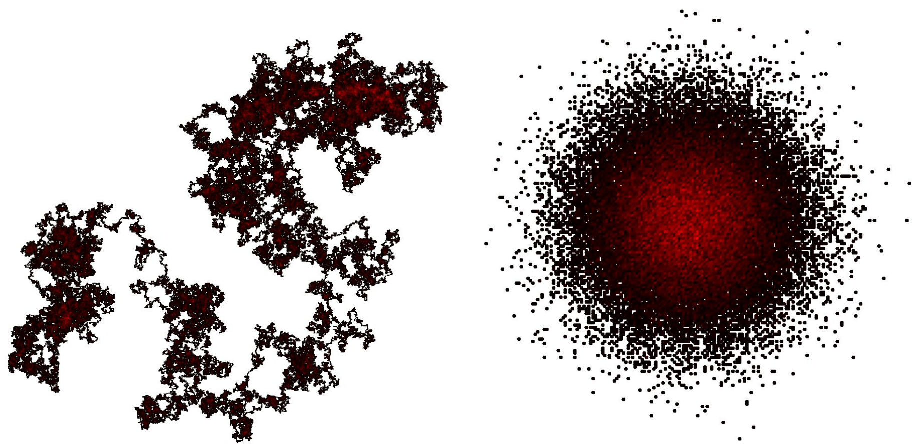


Figure 4.1: Figures of a simple random walk. Left: 1 run of 1.000.000 steps, right: endpoints of 50.000 runs of 1.000 steps.

up with other ways to approach this problem. It is safe to say that we still have $\mathbb{E}[X_n] = \mathbf{0}$, because of symmetry, but in order to find an expression for $\mathbb{E}[|X_n|]$ we will rely on our simulation program.

4.2.1 Square root

The first thing we would like to find out is whether we still have $\mathbb{E}[|X_n|] \sim \sqrt{n}$, as is the case when $\beta = 0$. In order to answer this question we simulate random walks with $\beta = 3$. The length of the simulations, n, varies from 200 to 10.000 steps and each of the simulations consists of 3.000 runs. For each of the simulations we take all endpoints and calculate the average Euclidean distance to the starting point, which estimates $\mathbb{E}[|X_n|]$. We then plot this displacement and a square root (in this case the least-squares square root). The result can be found in figure 4.3. This figure quite convincingly shows that $\mathbb{E}[|X_n|] \nsim \sqrt{n}$, at least for $\beta = 3$.

4.2.2 Alternative

That same figure leads us to believe that perhaps $\mathbb{E}[|X_n|] = cn^\alpha$, for some $\alpha < 0.5$. Again, we try to find out by doing simulations for different values of n. This time, we determine the least-squares fit of the form cn^α and plot that on top of the values found by the simulations. This is done for several different values of β in figure 4.4. The corresponding values of α and c can be found in tables 4.1 and 4.2 and figures 4.9 and 4.10. The fitted functions seem to describe the behaviour very well for random walks of less than 10.000 steps.

β	α in 1D	α in 2D	α in 3D	α in 4D	α in 5D
0	0.4976	0.4964	0.4932	0.5002	0.4979
1	0.4864	0.4725			
2	0.4829	0.4423	0.4925	0.4976	0.5003
3	0.4810	0.4129			
4	0.4728	0.3825	0.4819	0.5003	0.4967
6	0.4738	0.3359	0.4572	0.4881	0.4983
9	0.4719	0.3064	0.4018	0.4786	0.4990
15	0.4669	0.2876	0.3236	0.4390	0.4811

Table 4.1: Estimated values of α.

β	c in 1D	c in 2D	c in 3D	c in 4D	c in 5D
0	0.8143	0.9123	0.9661	0.9423	0.9614
1	0.4694	0.7194			
2	0.3474	0.6329	0.6412	0.7191	0.7591
3	0.2836	0.5835			
4	0.2567	0.5776	0.4861	0.5527	0.6430
6	0.2052	0.5814	0.4310	0.4831	0.5303
9	0.1678	0.5360	0.4448	0.3880	0.4171
15	0.1330	0.4539	0.4721	0.3232	0.3210

Table 4.2: Estimated values of c.

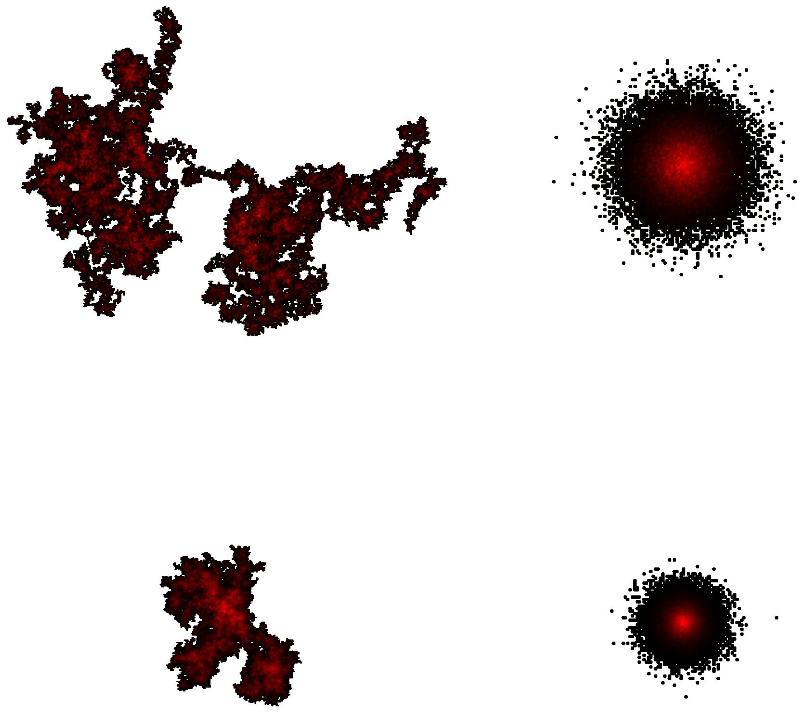


Figure 4.2: **Top-left:** $\beta = 2$, 1 run of 1.000.000 steps, **top-right:** $\beta = 2$, endpoints of 50.000 runs of 1.000 steps, **bottom-left:** $\beta = 5$, 1 run of 1.000.000 steps, **bottom-right:** $\beta = 5$, endpoints of 50.000 runs of 1.000 steps.

It might still be possible, though, that the behaviour in the long run will become different. We take the simulation program to it's limit and do simulations of 30.000 steps and 10.000 runs. If we extend the previous plot to include this new data, we obtain figure 4.5, where the new data (depicted by the red dots) has not been included in the fitting process. We notice that the new dots are quite close to the fitted graphs, which supports the idea that the graphs describe the behaviour properly. The biggest deviation occurs when $\beta = 0$, but this one was actually expected. We can see in table 4.1 that we found $\alpha = 0.4964$, but in section 4.1 we have seen that for $\beta = 0$ we should have $\alpha = 0.5$. This means that the fitted graph is actually a bit lower than it should be in the long run, and that it is actually nice to find this red dot slightly above the graph.

It would be interesting to take a closer look at the profiles of α and c. We will do that in section 4.4. We now first go to the one-dimensional case.

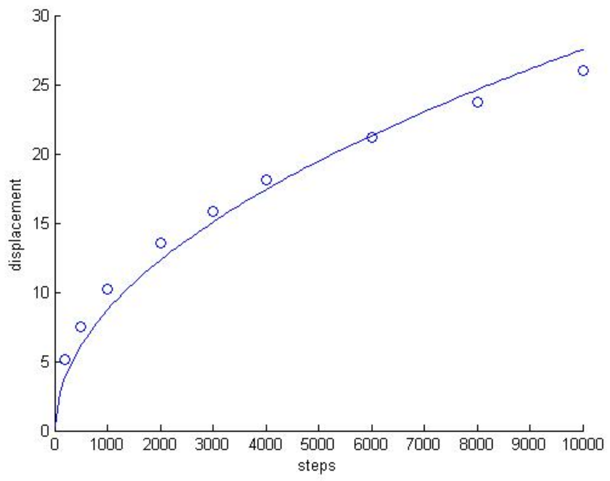


Figure 4.3: Expected displacement in 2D for $\beta = 3$ and a plot of the least-squares square root.

β	c
0	0.7971
1	0.4158
2	0.2997
3	0.2410
4	0.2022
6	0.1641
9	0.1321
15	0.1000

Table 4.3: Estimated values of c in 1D when $\alpha = 0.5$.

4.3 1D

For the one-dimensional case we also try to find α and c so that $\mathbb{E}[|X_n|] = cn^\alpha$. We do this in the same way. This time we do simulations for less different values of n, but our simulations consist of 5.000 runs so that they should be slightly more accurate.

The values of α and c that we obtain can be found in tables 4.1 and 4.2. We notice that the values of α seem to go down slightly as β increases, but they stay close to 0.5 now. Burgess Davis has proven that in 1D one does have $\mathbb{E}[|X_n|] = c\sqrt{n}$ for all β [3]. We are not sure why the values that we find are not exactly correct. Perhaps this is because for larger values of β the process does not move away much and more steps need to be done in order to obtain more accurate results.

A least-squares fit of the form $\mathbb{E}[|X_n|] = c\sqrt{n}$ gives us figure 4.6, and indeed the square

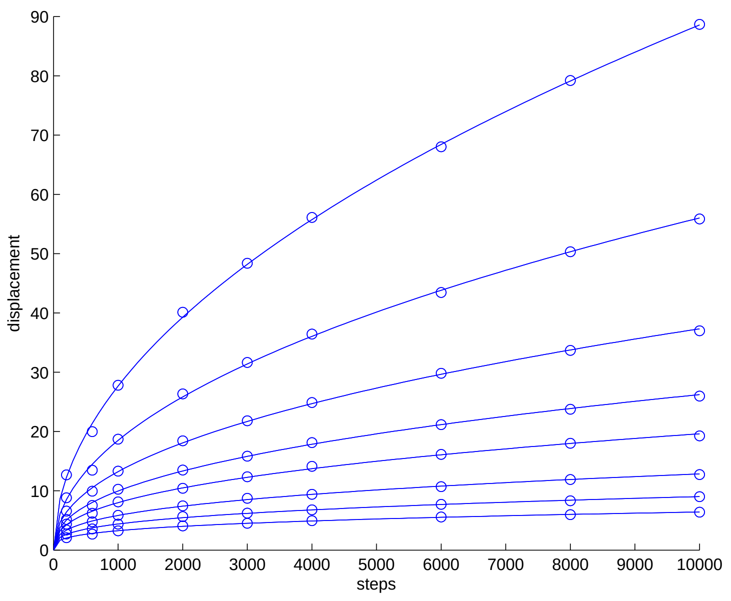


Figure 4.4: Expected displacement in 2D for $\beta = 0, 1, 2, 3, 4, 6, 9$ and 15.

root works well. The corresponding values of c can be found in table 4.3. We see that the c behave quite nicely. Therefore we try to fit them, and it turns out that this works out very nicely, at least if we ignore the value at $\beta = 0$. We obtain $c(\beta) = \frac{0.41}{\sqrt{\beta}}$. A plot of this can be found in figure 4.7. Putting these things together delivers us the following nice result for the one-dimensional case: $\mathbb{E}[|X_n|] = 0.41\sqrt{\frac{n}{\beta}}$.

4.4 Higher dimensions

For the higher-dimensional cases we also assume right away that $\mathbb{E}[|X_n|] = cn^{\alpha}$. We do simulations and fits for 3, 4 and 5 dimensions in a way similar to the one- and two-dimensional cases. This time, we only simulate up to 6.000 steps and we omit the cases $\beta = 1$ and $\beta = 3$, because of the time it costs. The resulting estimations for α and c can be found in tables 4.1 and 4.2 and figures 4.9 and 4.10. A plot of the 3D case (figure 4.8) shows that the fits seem quite good for random walks of up to 6.000 steps. Checks with longer random walks for the higher values of β show that in these cases the fitted values of α are not so accurate and usually a bit too small. Remember that we also saw that they were too small in the one-dimensional case. An estimated value of α that is too small implies that the estimated value of c is too large.

For each number of dimensions, denoted by d, we now consider α and c to be functions of β: $\alpha = \alpha_d(\beta)$ and $c = c_d(\beta)$. It does not seem easy to find general expressions for $\alpha_d(\beta)$

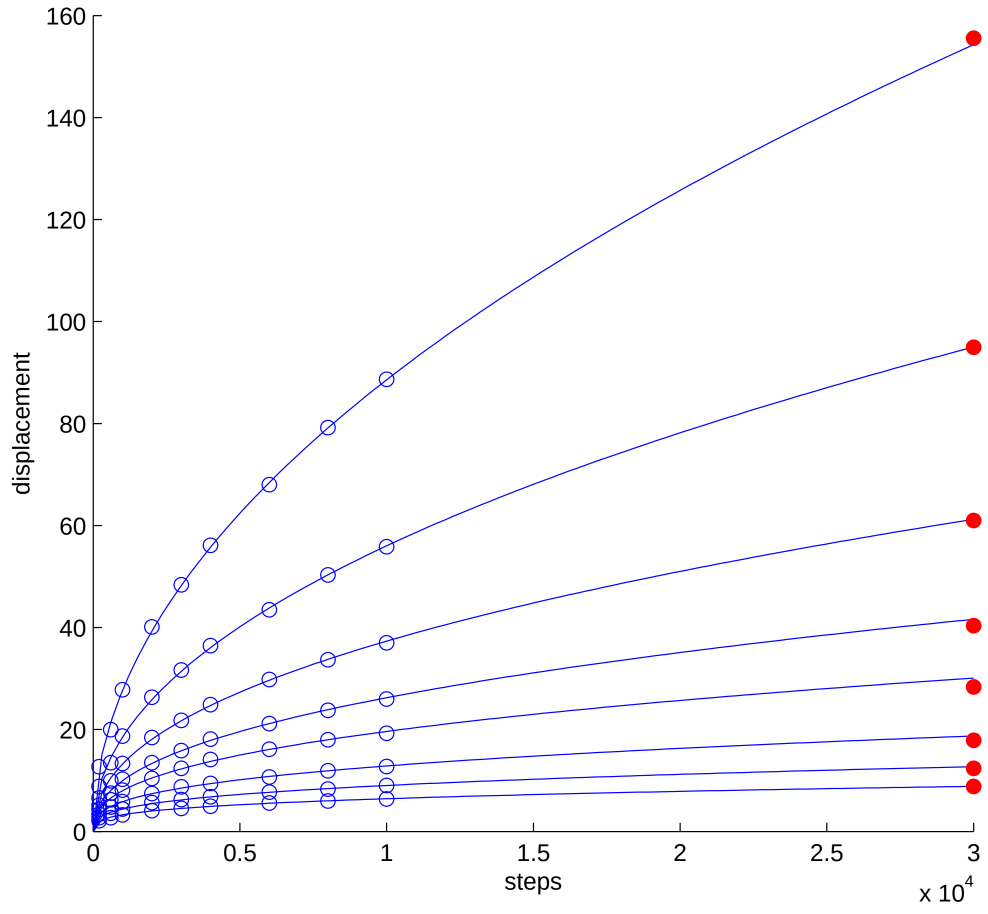


Figure 4.5: Expected displacement in 2D for $\beta = 0, 1, 2, 3, 4, 6, 9$, and 15, where we have plotted the simulated expected displacement after 30.000 steps on top of the fitted graphs.

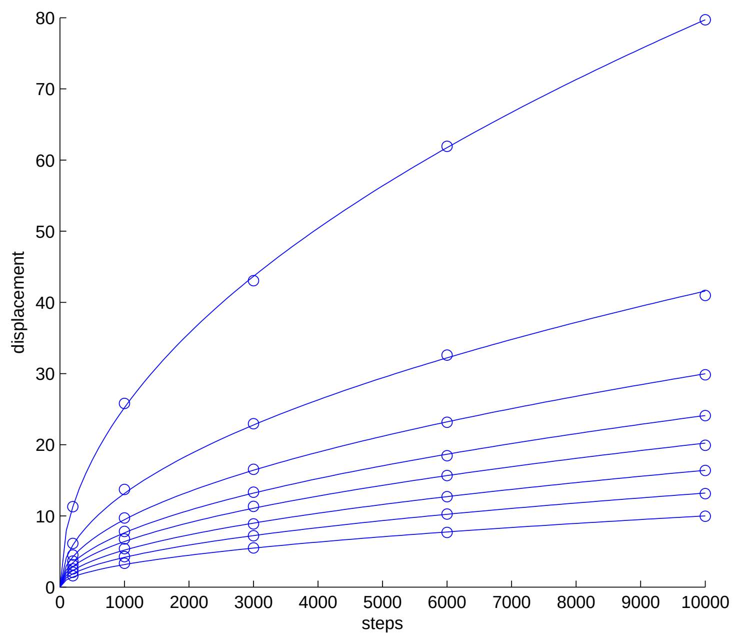


Figure 4.6: Expected displacement in 1D with square-root fits for $\beta = 0, 1, 2, 3, 4, 6, 9$ and 15.

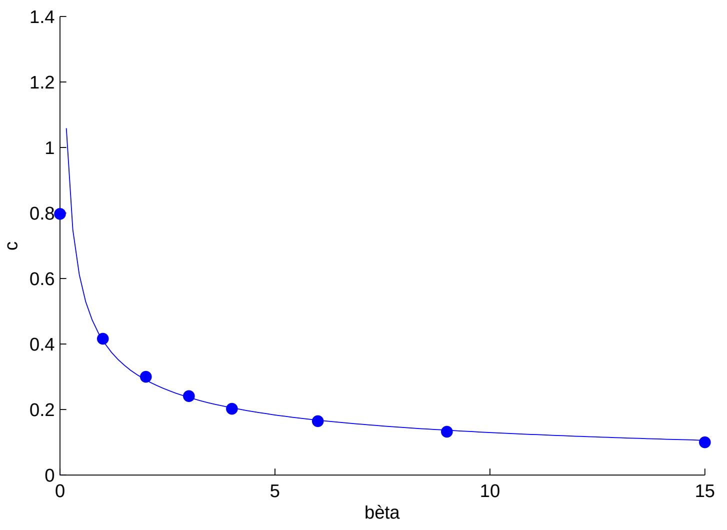


Figure 4.7: Estimated values of c in 1D when $\alpha = 0.5$.

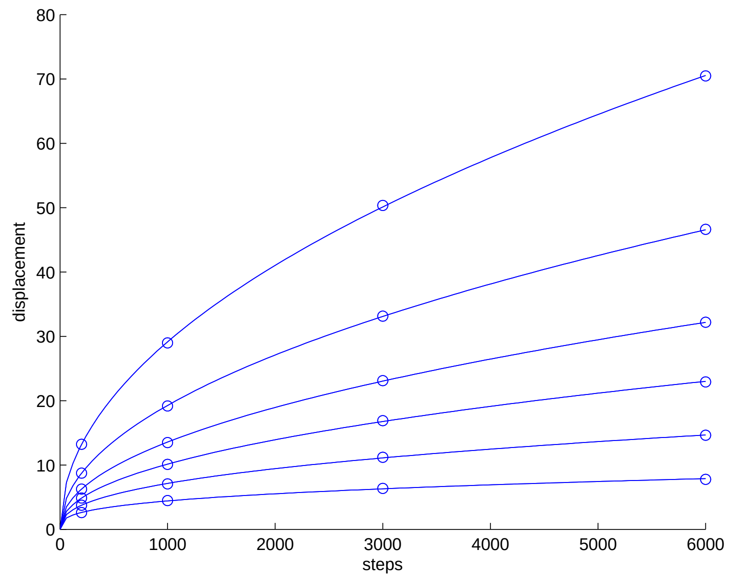


Figure 4.8: Expected displacement in 3D for $\beta = 0, 2, 4, 6, 9$ and 15.

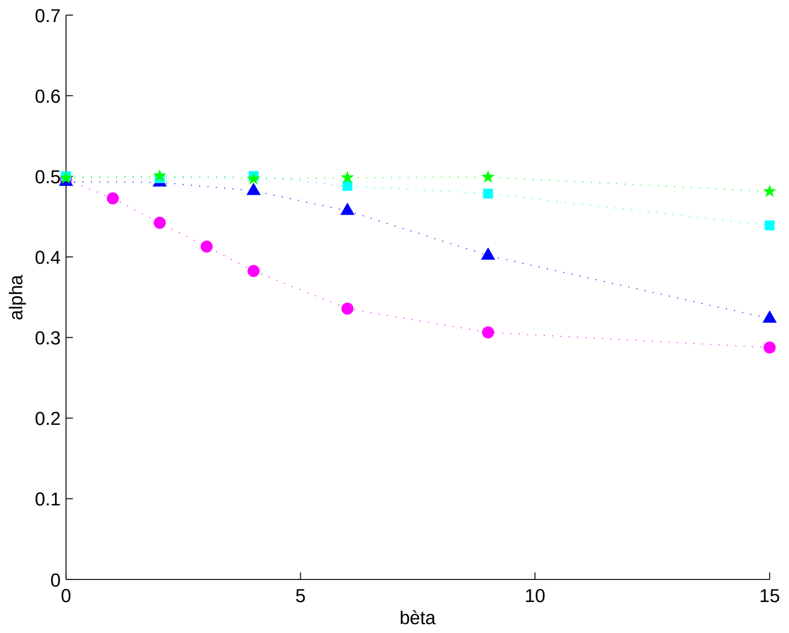


Figure 4.9: Estimations of α. Circle: 2D, triangle: 3D, square: 4D, pentagram: 5D.

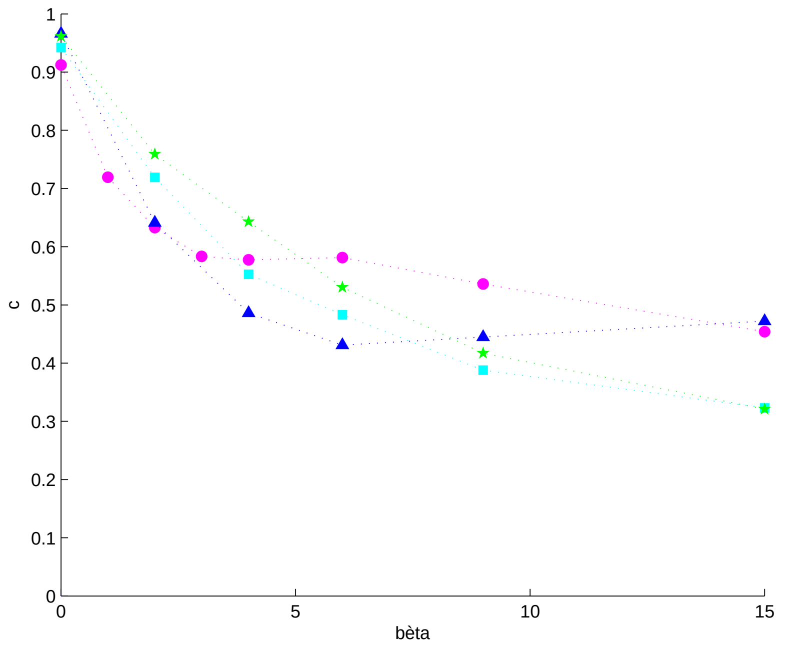


Figure 4.10: Estimations of c. Circle: 2D, triangle: 3D, square: 4D, pentagram: 5D.

and $c_d(\beta)$ from figures 4.9 and 4.10. Instead of looking for those, we will just draw some qualitative conclusions. We include the two-dimensional case.

1. α decreases as β increases: $\alpha'_d(\beta) \leqslant 0$. Intuition tells us that a higher β means a higher tendency to stay near the starting point, so at least it makes sense that α is not increasing.

2. α increases as the number of dimensions increases (for each $\beta \leqslant 15$): $d_1 > d_2 \Rightarrow \alpha_{d_1}(\beta) \geqslant \alpha_{d_2}(\beta)$. This implies that the influence of the reinforcements is less when there are more dimensions. We will get back to this later.

3. The two-dimensional case seems to be fundamentally different from the rest. The slope of α at $\beta = 0$ is strictly negative in 2D, while it seems to be equal to zero for higher dimensions: $\alpha'_2(0) < 0$, while $\alpha'_d(0) = 0 \ \forall_{d>2}$.

4. When the number of dimensions is at least three, there seems to be a critical value of β, up to which $\alpha = \frac{1}{2}$ and after which $\alpha < \frac{1}{2}$: $\forall_{d>2} \exists_{\beta^*} [\alpha_d(\beta) = \frac{1}{2}$ if $\beta \leqslant \beta^*$ and $\alpha_d(\beta) < \frac{1}{2}$ if $\beta > \beta^*]$.

A small error in the estimated value of α results in a much larger error in the value of c. There is a good chance that our estimates of c are perturbed significantly because of this, so we will not try to say anything about c.

One last thing we can say is about the limit of the number of dimensions to infinity. We have the following lemma:

Lemma 4.4.1 $\lim_{d\to\infty} |X_n|_2 = \sqrt{n}$*, almost surely.*

Proof Denote component j of X_n by X_n^j. Denote $K_n = \{j|X_n^j \neq 0\}$. Because $X_0 = \mathbf{0}$, we have $K_0 = \emptyset$. As $d = \infty$, $\mathbb{P}[(X_{n+1}-X_n = X_{i+1}-X_i)\cup(X_{n+1}-X_n = X_i-X_{i+1})] = 0, \forall_{0\leqslant i\leqslant n}$. This implies that $|K_{n+1}| = |K_n|+1$ (a.s.) and $|X_n^j| = 1, \forall_{j\in K_n}$(a.s.). By induction we obtain $|K_n| = n$ (a.s.). Putting things together we get $|X_n|_2 = \sqrt{\sum_{1\leqslant j\leqslant n}(X_n^j)^2} = \sqrt{n}$, a.s..

The key element in the proof is that, since $d = \infty$, the process will move into a 'new' direction at every step. The lemma follows directly from that.

We now refer back to the second of our observations concerning the behaviour of α. When the number of dimensions is higher, the probability of moving into a 'new' direction is higher, which results in α being closer to 0.5.

Chapter 5

OERRW with initial history

5.1 $\beta = \infty$

This section is about trying to find the history η_0 that maximizes the likelihood that a random walk starting at a point X_0 ends up at a point x after N steps, in the case that $\beta = \infty$. Two different approaches will be investigated: first we try to maximize the probability of moving *exactly* to the point x and then we try to minimize the expected distance to x. But before doing that, we will give a general analysis of the situation and derive an important general result.

5.1.1 General analysis

There are a few situations that may occur when $\beta = \infty$. The less interesting case is when η_0 does not include any edges from X_0 to a neighbour. In this case X_1 will be a random neighbour of X_0 and then there are again two options. The first is that the edge between X_1 and X_0 is the only edge from X_1 in η_1. In this case the process will just move from X_0 to X_1 and back forever. In the other case η_1 does include edges from X_1 to other points. This situation is a bit of a stretch, so we will not further discuss it, although it is quite similar to the following case.

We will now look at the case where η_0 includes at least one edge from X_0 to a neighbour. Because of (2.5) the process will start by traversing one of these edges. After that we will have (at least) $\{X_n, X_{n-1}\} \in \eta_n$ for all n, so (again by (2.5)) only edges that already were in the history can ever be traversed. We now have

$$\eta_n = \eta_0 = \eta \qquad \forall n, \tag{5.1}$$

which means that in this situation (2.5) turns into:

$$P_{ij} = \mathbb{P}(X_{n+1} = j | X_n = i) = \frac{\mathbf{1}\{\{i,j\} \in \eta\}}{\sum_{k \sim i} \mathbf{1}\{\{i,k\} \in \eta\}}. \tag{5.2}$$

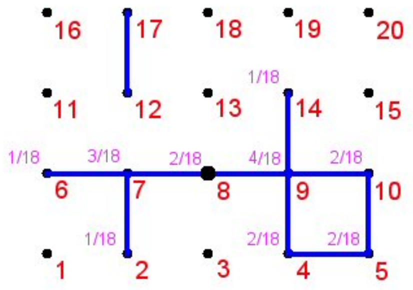


Figure 5.1: Visualization of a situation and it's stationary distribution ($\beta = \infty$).

It can easily be seen that, given a certain η, X_{n+1} now only depends on X_n and that $(X_n)_{n\in\mathbb{N}}$ therefore is a Markov chain. Since only known edges can be traversed, the state space S of $(X_n)_{n\in\mathbb{N}}$ consists of the points j for which there is a path from X_0 to j in η:

$$S = \left\{ j \in \mathbb{Z}^d \mid \exists_{i_1=X_0, i_2, \ldots, i_{n-1}, i_n = j \in \mathbb{Z}^d} : \{i_k, i_{k+1}\} \in \eta, 1 \leqslant k < n \right\} \tag{5.3}$$

For an example, look at figure 5.1. In that case $X_0 = 8$ (the big black dot) and $S = \{2, 4, 5, 6, 7, 8, 9, 10, 14\}$. The other points, including 12 and 17, can never be reached. Removing the edges that can never be traversed from the history will simplify things later:

$$\eta^* = \eta \cap \{\{i,j\} | i,j \in S\}. \tag{5.4}$$

In our example this means $\eta^* = \eta \backslash \{12, 17\}$.

We would like to find the stationary distribution for the Markov chain we now have. If $|\eta^*| = \infty$ we obtain $\pi_i = 0, \forall_i$. We will now assume $|\eta^*| < \infty$. In order to find the stationary distribution we now look for the reversible probability distribution π on S, which means π should satisfy

$$\pi_i P_{ij} = \pi_j P_{ji}. \tag{5.5}$$

P_{ij} are given by (5.2) and since $\sum_i \pi_i = 1$ we obtain:

$$\pi_i = \frac{|\{j : \{i,j\} \in \eta^*\}|}{2|\{\{j,k\} : \{j,k\} \in \eta^*\}|}. \tag{5.6}$$

Now [4] tells us that π is also the stationary distribution for the Markov chain. This leads us to the main result of this section:

Lemma 5.1.1 *π_i is proportional to the number of edges from point i to other points.*

π_i can be interpreted as the fraction of time spent in point i in the long run. The π_i for our example are the pink numbers in figure 5.1.

Remark The Markov chain is periodic, with period 2. Therefore, $\pi_i \neq \lim_{n\to\infty} \mathbb{P}(X_n = i)$. Instead:

$$\lim_{n \text{ even},\ n\to\infty} \mathbb{P}(X_n = i) = \begin{cases} 2\pi_i, & \text{if } |i - X_0|_1 = \text{even}, \\ 0, & \text{if } |i - X_0|_1 = \text{odd}, \end{cases}$$
$$\lim_{n \text{ odd},\ n\to\infty} \mathbb{P}(X_n = i) = \begin{cases} 0, & \text{if } |i - X_0|_1 = \text{even}, \\ 2\pi_i, & \text{if } |i - X_0|_1 = \text{odd}. \end{cases} \tag{5.7}$$

5.1.2 Movement precisely to x

In this section the goal is to find the history that maximizes the probability of moving from X_0 to x in N steps, when $\beta = \infty$. To make things easier, we will first look at the case $N = \infty$. This means that the result of the previous section, equation (5.6), applies and that we must maximize π_x.

In order for π_x to be positive, we need $x \in S$. This means that η has to contain a path from X_0 to x. Because the fraction in (5.6) needs to be minimized, it makes sense to take the shortest possible path. In case X_0 and x are on one line, one should add the line from X_0 to x to η_0, like in the upper history in figure 5.2. In general, all shortest paths are equivalent as the situation can be mapped onto a straight line.

Again because the fraction in (5.6) needs to be minimized, we add all edges that are adjacent to x. In the two dimensional case we obtain the lower history in figure 5.2, but the procedure is the same for any dimension. It is easy to check that π_x decreases if any of the edges is removed from η. It is also clear that if any more edges are added the numerator of (5.6) remains the same, while the denominator increases and therefore π_x decreases again. Thus, η consisting of the union of any shortest path from X_0 to x and all edges adjacent to x maximizes π_x.

It is interesting to see what happens when the number of dimensions is taken to infinity:

Lemma 5.1.2 $\lim_{d\to\infty} \pi_x = \frac{1}{2}$.

Proof x will be reached at some finite time and the process will then keep moving from x to one of it's infinitely many neighbours and back.

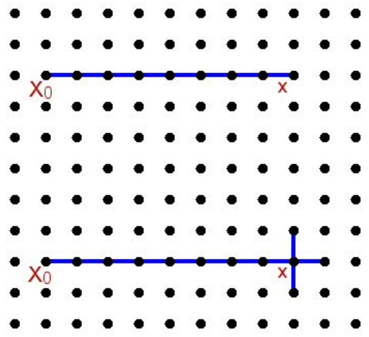


Figure 5.2: Visualization of two possible histories.

If we take into account the aperiodicity of the Markov chain (equation (5.7)), we see that:

Corollary 5.1.3

$$\lim_{d\to\infty} \lim_{n \text{ even},\ n\to\infty} \mathbb{P}(X_n = x) = \begin{cases} 1, & \text{if } |x - X_0|_1 = \text{even}, \\ 0, & \text{if } |x - X_0|_1 = \text{odd}, \end{cases}$$

$$\lim_{d\to\infty} \lim_{n \text{ odd},\ n\to\infty} \mathbb{P}(X_n = x) = \begin{cases} 0, & \text{if } |x - X_0|_1 = \text{even}, \\ 1, & \text{if } |x - X_0|_1 = \text{odd}. \end{cases}$$

The η that has been obtained is strongly suspected to be optimal also when $N < \infty$. This makes sense if we look at the distribution of X_n as a sort of diffusion process. At time $n = 0$ we have some mass at point X_0 and this mass diffuses through the system towards the stationary distribution that we have seen (at least if we somehow average over even and uneven timesteps). The speed of this diffusion process is not influenced by which η is taken and therefore there seems to be no reason why another η would be better. This is not so easy to prove, so we will leave it as a conjecture:

Conjecture 5.1.4 *η consisting of the union of any shortest path from X_0 to x and all edges adjacent to x maximizes $\mathbb{P}(X_N = x), \forall_{N>0}$.*

5.1.3 Minimizing distance to x

Another way of looking at this problem is looking for η that minimizes the expected distance to x at time N, which we denote as $\mu = \mathbb{E}[|X_N - x|]$. A reason for doing this, is that the simulation program can handle this much more accurately. We will start by analyzing the situation again, for the case $N = \infty$. This will be done using the L1-norm, because

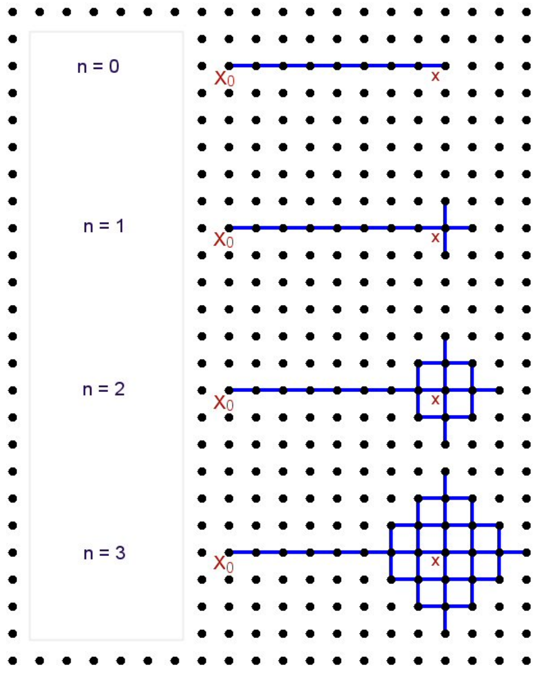


Figure 5.3: A series of possible histories.

the Euclidean norm makes the calculations too complicated. We will, however, relate our results back to the Euclidean case. Only the two dimensional case will be investigated, even though things can easily be generalized to general dimensions.

Basically the idea is that one should keep adding the edges to η that induce the biggest decrease of μ. This means that we should start, again, by adding any straight (L1-) line from X_0 to x. After this, we keep adding increasingly large L1-spheres around x as long as doing so decreases μ, which is the case if the distance of the L1-sphere to x is smaller than the value of μ without adding it. The series of histories we hereby end up with have been visualized in figure 5.3, where n now denotes the distance of the largest L1-sphere to x. In this case there is only one straight line from X_0 to x and that line enters the L1-ball exactly in the west-most point. All results remain the same if there are several straight lines possible. Nothing changes if the line enters the ball at, for example, the south-west. We will denote the distance between X_0 and x by D and the history corresponding to n and D by $\eta^{D,n}$. The optimal of these histories seems to be the optimal history overall, as it is not hard (but a lot of work) to check that removing or adding any edge increases μ.

The job now is to find n such that μ is minimized. Because $\beta = \infty$, we have

$$\mu = \sum_{i \in S} \pi_i |i - x|_{L1}, \tag{5.8}$$

in which the π_i are given by (5.6). In case we have a history $\eta^{D,n}$, as described above, we now obtain:

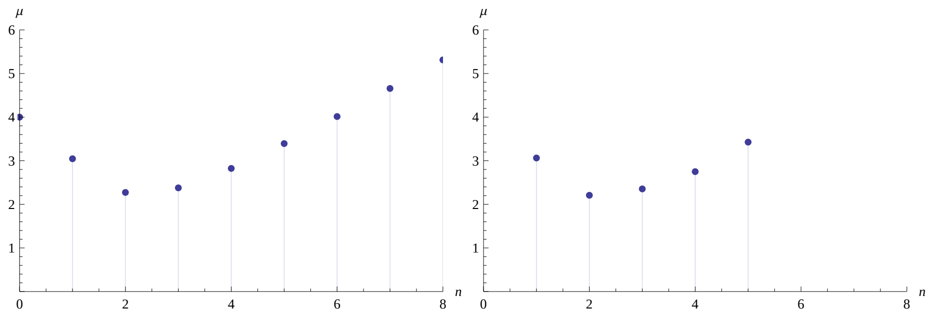


Figure 5.4: Expected distance for D=8. Left: exact values, right: some simulated values.

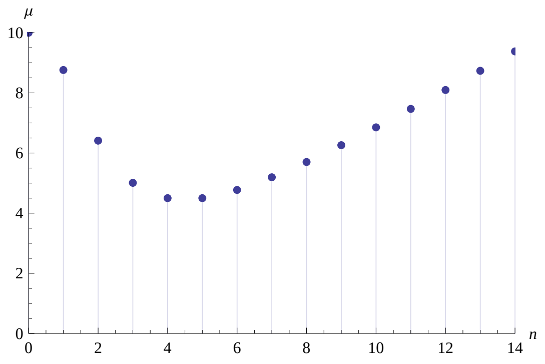


Figure 5.5: Expected distance to x for D=20.

Lemma 5.1.5

$$\mu(\eta^{D,n}) = \frac{16n^3 - 3n^2 - 4n + 3D^2}{24n^2 - 6n + 6D}.$$

Proof In our case S consists of all the points on the line and the L1-ball. We know the distance of each point to x and we know the π_i, so with a little effort we can calculate the sum in equation (5.8).

For $D = 8$, plots of μ as a function of n can be found in figure 5.4. It can be seen that, in this case, $n = 2$ gives the optimal history. When we took the approach of maximizing the probability of ending up exactly in x, $n = 1$ gave the optimal history, so there is a discrepancy here.

A plot of some simulation results has been included for comparison's sake. The results are very close to the analytical result, indicating that the analytical result as well as our simulation program are valid. At this point we will not further go into the simulation.

A plot of μ as a function of n for the case $D = 20$ can be found in figure 5.5. Apparently, for $D = 20$ one should pick either $n = 4$ or $n = 5$. It would be nice to have an analytical

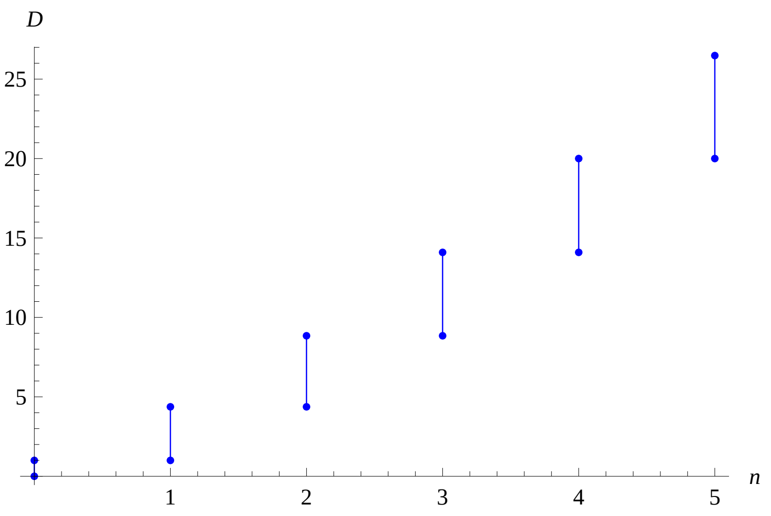


Figure 5.6: The range of D for which an L1-sphere of size n is optimal.

expression for the optimal n, or n_{opt}, as a function of D. Unfortunately, this requires solving a polynomial of degree four, which is hard. Instead, we assume that n_{opt} is increasing in D and we look for $D(n)$ that satisfies $\mu(\eta^{D,n}) = \mu(\eta^{D,n+1})$. This $D(n)$ corresponds to the distance at which the transition takes place from the optimum being equal to n to the optimum being equal to $n+1$. The solution to the equation is the positive root to a polynomial of degree three, which is a lot easier to solve:

$$D(n) = \frac{1}{6}(3 + 6n + \sqrt{3}\sqrt{3 + 16n + 48n^2 + 32n^3}) \tag{5.9}$$

For each n we obtain a range of D for which that n is optimal. We notice that, in the long run, $D \sim {n_{opt}}^{\frac{3}{2}}$, which suggests that $n_{opt} \sim D^{\frac{2}{3}}$. A plot of the ranges can be found in figure 5.6. It is nice to see that $D = 20$ is, indeed, close to the transition from $n = 4$ to $n = 5$.

For the same reason as in the previous section, we pose:

Conjecture 5.1.6 *η consisting of the union of any shortest path from X_0 to x and the L1-ball of size n as found above maximizes $\mathbb{P}(X_N = x), \forall_{N>0}$.*

As promised, the result will now be related to the case of a Euclidean norm. We suspect that exactly the same ideas will work with the Euclidean norm (and other equivalent norms). This means that first a path from X_0 to x is added to η, such that each of edges maximally reduces μ. This will result in a path that is as close as possible to the Euclidean straight line from X_0 to x. Now one should take a Euclidean sphere around x and increase it's radius in a continuous way. Whenever this growing sphere 'eats' new points, all new edges within the sphere should be added to η. Just like in the L1-case, this process must stop when the radius of the sphere, and therefore the distance of new points to x, exceeds the current μ. That is all we will say about it, as this situation is harder to analyse.

5.2 $\beta < \infty$

In this section the question is whether the optimal history that was found in section 5.1.3 is still optimal when $\beta < \infty$. We will still look at L1-norms and restrict ourselves to the two-dimensional case. In the previous chapter we have seen that for $\beta < \infty$ and $N \to \infty$ the expected distance to X_0, and therefore also to x, goes to infinity. Therefore, it makes no sense to think about infinitely long random walks in this case.

We assume (by conjecture 5.1.6) that when the distance from X_0 to x equals 8 and $\beta = \infty$, the optimal history is $\eta^{8,2}$, which is the third from the top in figure 5.3. We will do simulations for $\beta < \infty$, using $\eta^{8,2}$ as well as other η. If we find that the smallest $\mu = \mathbb{E}[|X_N - x|]$ occurs for some $\eta \neq \eta^{8,2}$, then we know that $\eta^{8,2}$ is not optimal.

We do random walks of 10.000 steps, which is quite a lot compared to the distance we want to traverse. We use the 'randomize steps' feature in order to mitigate the periodicity. We do 25.000 runs.

To start, we do simulations for $\beta = 100$ and $\beta = 20$, using the histories $\eta^{8,1}, \eta^{8,2}$ and $\eta^{8,3}$. The results can be found in table 5.1 and a visualization of the endpoints in figure 5.7. We see that $\eta^{8,3}$ is better than $\eta^{8,2}$ in both cases. It is safe to assume that histories with larger L1-balls would be even better in the $\beta = 20$ case. The following conjecture is carefully posed:

Conjecture 5.2.1 *Let η be any shortest path from X_0 to x. Add increasingly large L1-spheres around x to η as long as doing so decreases the expected distance to x after N steps. Now η maximizes $\mathbb{P}(X_N = x)$.*

Remark The optimal η depends on both β and N. As β decreases, the size of the L1-ball increases. As N increases, the size of the L1-ball also increases.

We notice that for β not very large and N large enough, X_0 will be included in the L1-ball around x. We suspect that as $N \to \infty$, the radius of the L1-ball goes to infinity. Indeed, having a history at all does not make a difference now, because after an infinite time the expected distance to X_0 and therefore to x will be infinite anyway.

There does not seem to be a reason why another history would be better, but this needs further investigation.

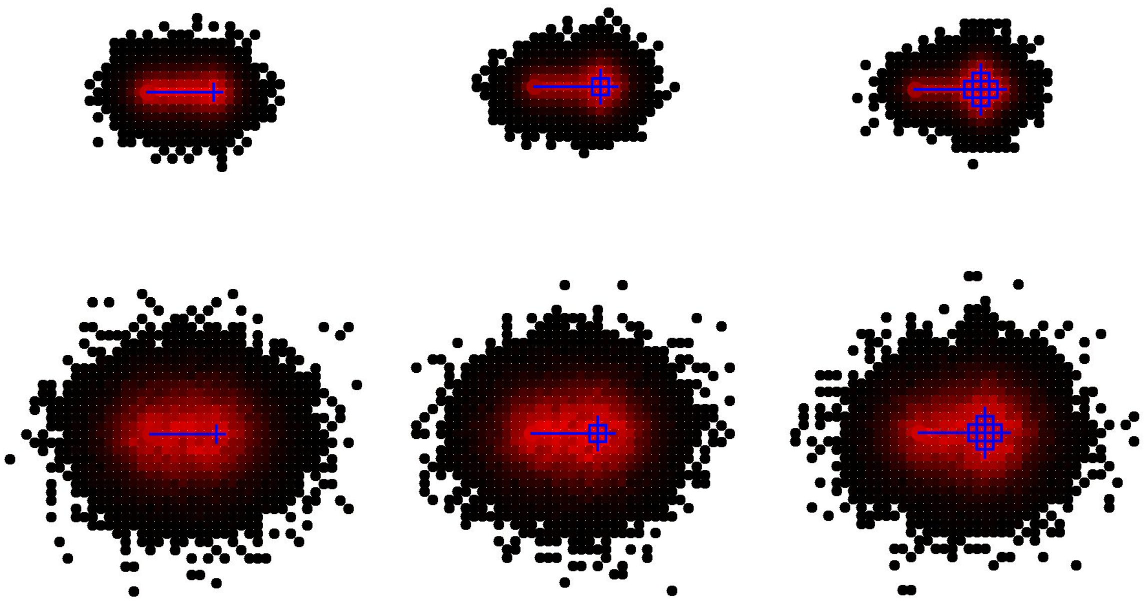


Figure 5.7: Endpoints of simulations of 10.000 steps. Top: $\beta = 100$, bottom: $\beta = 20$.

n	$\beta = 100$	$\beta = 20$
1	5.2597	8.1999
2	4.7016	7.8484
3	4.2396	7.5006

Table 5.1: Estimated distance to x after 10.000 steps.

Chapter 6

Conclusion

Because it is hard to analyse the Once-Edge Reinforced Random Walk analytically, we have started by writing a simulation program. The user can enter the parameters, such as β and an initial history (η). The main functionality of the program is that it simulates a lot of random walks and then provides us with the distances of the endpoints of all of the simulations to a fixed endpoint: x.

Our research has been concentrated on two things. The first question is how far the random walk will move away from the origin, when there is no initial history. For $\beta = 0$ the OERRW reduces to a simple random walk. In that case, the expected distance to the origin after a random walk of n steps behaves as $\sqrt{n}$, by the Multivariate Central Limit Theorem. For the one-dimensional case, Burgess Davis [3] has proven that, for all values of β, the random walk still moves away as $\sqrt{n}$. The multi-dimensional cases are harder. Therefore, we rely completely on our simulations. It turns out that when the number of dimensions is greater than one, the behaviour changes.

The process still moves away as $c\, n^{\alpha}$, but we often find $\alpha < \frac{1}{2}$. The first things we notice are that α decreases as β increases and that α increases as the number of dimensions, d, increases. It is not surprising that the process moves away less when β increases, because in that case the tendency to stay near 'known' areas is higher. It is also not surprising that the process moves away more when d increases, because in that case the ratio of 'known' and 'unknown' adjacent edges will generally be smaller. The thing that is worth noting, is that the difference is so fundamental that it also influences α, and not just c.

The two-dimensional case seems to differ from the higher dimensional cases in the sense that for the higher dimensional cases, there seems to be some positive critical value, up to which $\alpha = \frac{1}{2}$ and after which $\alpha < \frac{1}{2}$. In the two-dimensional case, on the contrary, α starts decreasing right away: $\alpha_2'(0) < 0$. Apparently, for higher dimensions, small reinforcements do not change the behaviour in a way that α is influenced. This is very interesting, but we can not explain it at this point. More research needs to be done.

The second question is which initial history should be chosen if the goal is to move from a point X_0 to a point x in N steps. We only look at this in the two-dimensional case, but most results can easily be generalized. This problem is approached in two different ways: we look what happens when we try to maximize the probability of ending up exactly in x, and we look what happens when we try to minimize the expected distance to x. For both cases, we first analyze what happens when $\beta = \infty$.

When $\beta = \infty$ the history that maximizes the chance of ending up in x after an infinite time is the history that consists of the union of any shortest L1-path from X_0 to x and all edges adjacent to x. We conjecture that this does not change when the random walk has a finite length N. We also think the same is true when $\beta < \infty$, although it might be necessary to take a shortest Euclidean path from X_0 to x in this case. Unfortunately, the simulation program in it's current state can not contribute much here. The variance in the number of simulations that end up exactly in x is large. It would be possible to obtain a proper value for the expected number of times the random walk ends up in x by assuming that the number of times it ends up in x should behave nicely compared to the number of times it ends up in the points in x's neighbourhood. Some sort of fit could be used, and while doing that also things like symmetry can be used. We have not looked into this, though.

In case we want to minimize the expected L1-distance to x, the analysis is a little more difficult. For $\beta = \infty$ and infinitely long random walks, it still is possible though. We find that the optimal history consists of any shortest L1-path from X_0 to x and an L1-ball around x. The radius of the L1-ball is obtained from equation (5.9). Again, we conjecture that this is still optimal when the random walk has a finite length. In this case, simulations tell us that the same history is not optimal when $\beta < \infty$. Histories containing a larger L1-ball around x are better. We suspect that a history of the same form, but with a larger L1-ball is optimal and that the size of this L1-ball is a function of β and the length of the random walk. Both when β decreases and when N increases, the radius of the ball increases. But, for this also, more research is necessary.

Bibliography

[1] L. Breiman. Probability. pages 237–238.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. pages 221–245, 2005.

[3] B. Davis. Brownian motion and random walk perturbed at extrema. *Probab. Theory Related Fields*, 113(4):501–518, 1999.

[4] Olle Häggström. Finite markov chains and algorithmic applications. page 39, 2002.

Appendix A

Simulation: structure of the program

In order to get a better idea of how the program works it is useful to look at the structure of the program. A class diagram can be found in figure A.1. The program consists of classes, each of which is depicted by a box containing the name of the class, the important variables and the important methods. The lines correspond to relations between the classes. A line with a 1 and a * means that one object of the first type holds many objects of the second type, and a line with two 1's indicates a one-to-one relationship. A short explanation of each of the classes will be given below. It is useful to remember that a class is like a blueprint, while objects are instantiations of classes. For example, there is only one vertex class, but there are a lot of vertex objects in the program, that all have their own coordinates etcetera.

- ***interface*** This is the main class. The interface of the program is created here (hence the name of the class). When the user interacts with the interface, this class makes sure the right things happen. One example of something that is done here, is when the user presses the 'Add simulation'. The interface then creates a new simulation object, which in instantiation of the simulation class (see below). Another example is that when the user presses the 'Run simulation(s)' button, the interface goes to the simulation object that belongs to the line that is currently selected in the information box, and calls the *runSimulation()* method inside that object. The interface also holds several other objects. There is also one drawingPanel object, which makes sure things can be drawn in the visualization box. There is one field object, which is the general working history. There are also one dataExporter object, one openAndSave object and one data object.

- ***simulation*** This class represents a simulation. There will be one simulation object for each simulation in the program. Each simulation object holds parameters like β, the number of steps and the number of runs. Each simulation object also holds several field objects. One of these (*hist*) holds the (initial) simulation history. Another one (f) is the working field. Each individual run is simulated here, and after each run

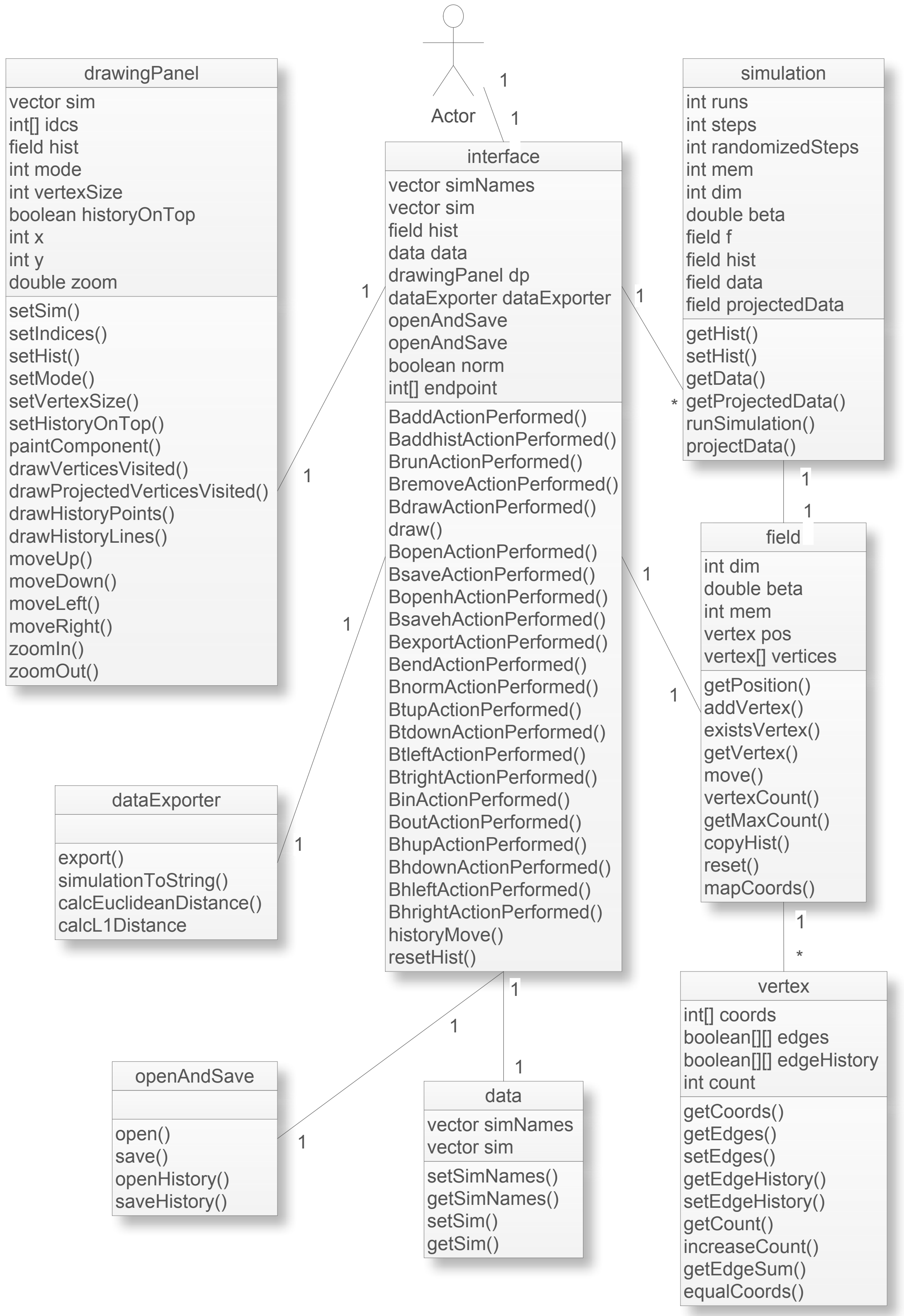


Figure A.1: Class diagram of the program.

this field is cleared. The final two (*data* and *projectedData*), hold the data that needs to be stored. The most important method of this class is the *runSimulation()* method, which does the actual simulating. Basically, the *move()* method inside the f field is called *steps* times. This corresponds to one run. The *move()* method is the method that makes sure one step is done in the random walk and will be mentioned in the description of the field class. After this run is simulated, the relevant data is copied to the *data* field. When there are multiple runs, the relevant data is just the endpoint, otherwise it is f completely. When there are multiple runs, f is cleared and the whole process is repeated *runs* times.

- ***field*** A field represents the $\mathbb{Z}^d$ space and as such holds a bunch of vertices (described at the vertex class). Instances of this class are used for several things. Each simulation has a working field, in which random walks are simulated. But histories and the collection of endpoints of runs are also stored in this format. The most important method of this field is the *move()* method. There are two versions of it, one that needs to be given a direction to move in and one that does not need any information. The first one is, for example, used when the 'history point' is moved around in the general working history (see section 3.3.3). The second and more important one is the one that is called when an actual random walk is simulated. This version decides where to go randomly, based on the rules of the OERRW.

- ***vertex*** A vertex represents a point in the space. This consists not only of the coordinates of the point in the space, but it also holds information about which of the adjacent edges have been traversed in the past and about the number of times the point has been visited.

- ***drawingPanel*** Of the drawingPanel, as well as of the dataExporter, openAndSave and data classes, there is only one instantiation. These are all held directly by the interface. The drawingPanel object holds the methods that draw everything in the visualization box.

- ***dataExporter*** This object holds the methods that export the distances of the points in the *data* fields of the selected simulations to the endpoint that can be set in the interface.

- ***openAndSave*** This object holds the methods that open and save sets of simulations as well as general working histories.

- ***data*** The only goal of this object is to make saving and opening simulations easier. This way, both the names of the simulations, as seen in the information box, and the actual simulations can be saved into one object.

Appendix B

Simulation: memory management

An important aspect of this kind of simulation program is the way information is stored. To decide where to step next one needs to know which of the traversable edges are already 'known', so it is necessary to keep track of all edges that have been traversed. First the two simplest and most obvious ways of doing that kind of thing will be discussed.

The first one is creating a vector of either all edges that have been traversed or all points that have been visited. The advantage is that very little memory is used. The disadvantage, though, is that the program has to search through the list at every step in order to find out whether the edges are in it. When more steps are done the list will get longer and it will take increasingly long to do this. Keeping the list ordered and performing binary searches would decrease the time significantly, but the search times will still keep increasing.

The simplest way of getting a constant searching time is by designating a memory spot for each item (edge or point) that could possibly be reached (in a matrix, for example), and saving whether it has been reached in that memory spot. The problem here is that the amount of memory that will be needed when doing n steps in d dimensions will be of the order $(2n)^d$, which gets much too large.

We will now explain what we do instead. It is not exactly as described, but what matters are the basic principles. Rather than the above methods we use some variation of a hash table (see for example [2]). The idea is to exploit the fact that there are much less points that are actually visited than points that could possibly be visited. This is done by first fixing a number of memory spots, let us say n. This should for example be two times the number of steps the process is going to take. We then divide our infinite space ($\mathbb{Z}^d$) into n disjoint subsets. We then assign each of these subsets (consisting of infinitely many points) to a memory spot. So, each point has one easily traceable memory spot that it will be stored at if it is visited, and each memory spot has infinitely many points that could be stored at it. Of course, it is now possible that more than one of the points in the subset that has been assigned to the same memory spot actually get visited. In that case, all of these visited points want to be stored at the same spot. There are a lot of different ways to deal with this 'problem'.

What we do is assign a different list of memory spots to each of the points in $\mathbb{Z}^d$. If a point is visited, the program will first try to store it at the first spot in the list. In the unlucky case that this spot is already taken, it will try to store it at the second spot in the list, and so on. Now when the program needs to search for a point, it will first look at the first of the memory spots in that point's list. If this spot is empty, the point has never been stored and therefore never been visited. If there is a point in this spot and this point is the point we are looking for, we are also done. It is also possible (but relatively rare), that the point in this spot is another point than the one we are looking for, as there are infinitely many points that could (in principle) be stored at this spot. When this happens, the program checks out what the second spot is in the list of the point that it is looking for, and it then looks in that spot, etcetera.

It is good to notice the importance of the ratio of different points that are actually visited (and therefore stored) and the total number of memory spots. When the number of different visited points increases, there will be less empty spots in the memory and this slows the system down. As long as this ratio does not exceed 0.5 it will be fast, though.

What remains to be mentioned is the way in which we assign a list of memory spots to each point. There are a lot of ways to do this, and the ones that work best attempt to create a list that has some kind of randomness, for reasons that can be found in [2]. What we do is simpler, but effective. When a point needs to be stored or found, we create a seed based on the coordinates of the point. Using this seed, a random number generator then produces a random sequence of indices of memory spots. It is crucial that each time a random number generator starts generating random numbers using the same seed, it produces the same sequence of numbers.

Using this method, we can do a run of millions of steps in a short period of time. We have compared this to the vector-method described above (without sorting and binary search), and at least in that form it is impossible to do that many steps in an acceptable amount of time with that method.