MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
V.N. Karazin Kharkiv National University
School of Mathematics and Computer Science
Department of Theoretical and Applied Informatics

Master's Thesis

Constrained random walks on lattices

Author:
Final year Master's Program student,
group MCS-64
specialty - Computer Sciences and
Information Technologies,
educational program: "Informatics"
Cai Wende
Supervisor: Anna Goncharuk
Reviewer: Andrii Chukhrai

Kharkiv, 2024

Table of Contents

Abstract

Random walks on lattices are an essential topic in combinatorics and mathematical modeling, with applications in physics, computer science, and biology. This paper explores the combinatorial aspects of random walks under constraints. The study addresses specific domains, including lines, planes, and bounded rectangles, while providing recursive formulas and practical implementations. We present a generalized algorithm to calculate the number of paths under given constraints and extend the discussion to complex domains, such as obstacle-laden grids.

1 Introduction

Random walks are foundational models in stochastic processes and combinatorics, often used to describe systems involving uncertainty. Traditional studies focus on probabilistic properties, but this paper emphasizes the **combinatorial perspective**. The primary objective is to calculate the number of paths between two points under specific constraints.

1.1 Objectives

- Investigate path counting formulas for various lattice domains.
- Design and implement algorithms to compute constrained random walks.
- Explore applications of constrained random walks in specialized domains.

1.2 Structure of the Paper

The paper is organized as follows:

- **Section 2** introduces the theoretical framework of random walks.
- **Section 3** explores the combinatorial approach to counting paths.
- **Section 4** examines constrained paths in specific domains.
- **Section 5** presents the implementation and results of the computational models.

- **Section 6** concludes with a discussion of findings and future work.

2 Theoretical Foundations

2.1 Definition of Random Walks

A random walk is a sequence of moves through a mathematical space, with each move determined by a probabilistic rule or, in this study, constrained by combinatorial rules.

2.2 Random Walks on a Line

- **Recursive Formula**: $$P_n(x) = P_{n-1}(x-1) + P_{n-1}(x+1)$$

  where Pn(x) represents the number of paths of length nn reaching point x from 0.

- **Initial Conditions**: $$P_0(0) = 1, \quad P_0(x) = 0 \quad (x \neq 0)$$

2.3 Random Walks on a Plane

In a two-dimensional lattice, each point (x,y) can be reached from:

$$(x-1, y), \quad (x+1, y), \quad (x, y-1), \quad (x, y+1)$$

The recursive formula is:

$$P_n(x, y) = P_{n-1}(x-1, y) + P_{n-1}(x+1, y) + P_{n-1}(x, y-1) + P_{n-1}(x, y+1)$$

3 Combinatorial Approach to Path Counting

3.1 Paths on a Line

- **Unbounded Line**: The solution can be derived using combinatorial tools such as Pascal's Triangle.
- **Bounded Interval**: Additional boundary conditions, $P_n(a-1) = 0$ and $P_n(b+1) = 0$, are introduced.

3.2 Paths on a Plane

- Discusses the case of an unrestricted grid.
- Introduces symmetry properties and binomial coefficients for path calculation.

3.3 Constraints and Domains

- **Rectangles**: Bounded grids require adjustments to recursive relations.
- **Avoiding Obstacles**: Paths that avoid specific regions use exclusion principles, making the problem computationally challenging.

---

4 Constrained Domains

4.1 Rectangular Boundaries

Constrained random walks in a rectangle $[a,b]\times[c,d]$ times [c, d] must satisfy:

$$P_n(x,y) = 0, \quad \text{if } x < a, x > b, y < c, \text{or } y > d.$$

4.2 Avoiding Obstacles

In domains with obstacles, additional conditions are applied:

$$P_n(x,y) = 0, \quad \text{if } (x,y) \in \text{Obstacle Set}.$$

Such problems often require computational methods for precise solutions.

5 Implementation and Results

5.1 Algorithm Design

The algorithm calculates the number of paths given the step limit nn and domain constraints. The recursive nature of the problem translates naturally into dynamic programming.

Algorithm Steps:

1. **Initialize**: Define the grid and constraints.
2. **Iterative Updates**: Use recursive relations to compute $P_n(x, y)$.
3. **Boundary Conditions**: Apply constraints as zero-valued states.

5.2 Code Implementation

```
def constrained_walks(n, bounds, obstacles=set()):
    from collections import defaultdict

    dp = defaultdict(int)
    dp[(0, 0)] = 1  # Starting point

    for step in range(1, n + 1):
        next_dp = defaultdict(int)
        for (x, y), count in dp.items():
            moves = [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]
            for nx, ny in moves:
                if (nx, ny) in obstacles:  # Avoid obstacles
                    continue
                if bounds and not (bounds[0] <= nx <= bounds[1] and bounds[2] <= ny <=
bounds[3]):
                    continue
                next_dp[(nx, ny)] += count
        dp = next_dp

    return dp
```

5.3 Example Outputs

- **Case 1**: Rectangle bounds $[-2, 2] \times [-2, 2]$.

- **Case 2**: Avoiding a specific obstacle (1,1)

5.4 Performance Analysis

- **Time Complexity**: $O(n \cdot |S|)$, where $|S|$ is the size of the grid.
- **Space Complexity**: Optimized by reusing memory for intermediate steps.

6 Conclusion

6.1 Summary

This paper presented a combinatorial approach to constrained random walks, deriving recursive formulas and demonstrating their application to various domains. The implementation provided an efficient method for practical computation.

6.2 Future Work

- Extend to higher dimensions.
- Develop closed-form solutions for complex domains.
- Explore weighted random walks with direction-specific probabilities.

7.References

Feller, W. (1968). *An Introduction to Probability Theory and Its Applications, Volume 1*. Wiley. Relevant Chapters: Chapter 3 (Fundamentals of stochastic processes) and Chapter 14 (Random walks and recursive relations). Key Content: Provides theoretical background for one-dimensional random walks and recursive formulas; discusses the "drunken sailor problem" as a classic example.

Durrett, R. (2019). *Probability: Theory and Examples*. Cambridge University Press. Relevant Chapters: Chapter 4 (Markov chains and random walks) and Chapter 6 (Asymptotic analysis of path counting). Key Content: Covers

theoretical foundations for two-dimensional and higher-dimensional random walks; explains path restrictions within rectangular boundaries.

Stanley, R. P. (1997). *Enumerative Combinatorics, Volume 1*. Cambridge University Press. Relevant Chapters: Chapter 1 (Applications of generating functions) and Chapter 4 (Recurrence relations and their combinatorial interpretations). Key Content: Provides combinatorial methods for deriving path counting formulas; includes generating functions for paths in specific domains.

Lawler, G. F., & Limic, V. (2010). *Random Walk: A Modern Introduction*. Cambridge University Press. Relevant Chapters: Chapter 2 (Properties of simple random walks) and Chapter 5 (Theory and simulations of random walks in restricted regions). Key Content: Discusses computational methods for random walks with obstacles; includes numerical examples for paths avoiding specific quadrants.

Gessel, I., & Zeilberger, D. (1992). "Random Walk in a Weyl Chamber." *Proceedings of the American Mathematical Society*, 115(1), 27-31. Key Content: Studies random walks avoiding restricted regions (Weyl chamber); provides explicit counting formulas for constrained paths.

Bousquet-Mélou, M., & Petkovšek, M. (2000). "Walks Confined in a Quadrant Are Not Always D-finite." *Theoretical Computer Science*, 307(2), 257-276. Key Content: Analyzes paths confined to quadrants with complex generating functions; explores computational complexity of path counting in constrained domains.

Wikipedia: Random Walk. Available at https://en.wikipedia.org/wiki/Random_walk. Relevant Sections: Recursive relations for one-dimensional and two-dimensional random walks; discussions

on path restrictions within rectangles. Key Content: Offers a quick overview of basic random walk concepts and applications.

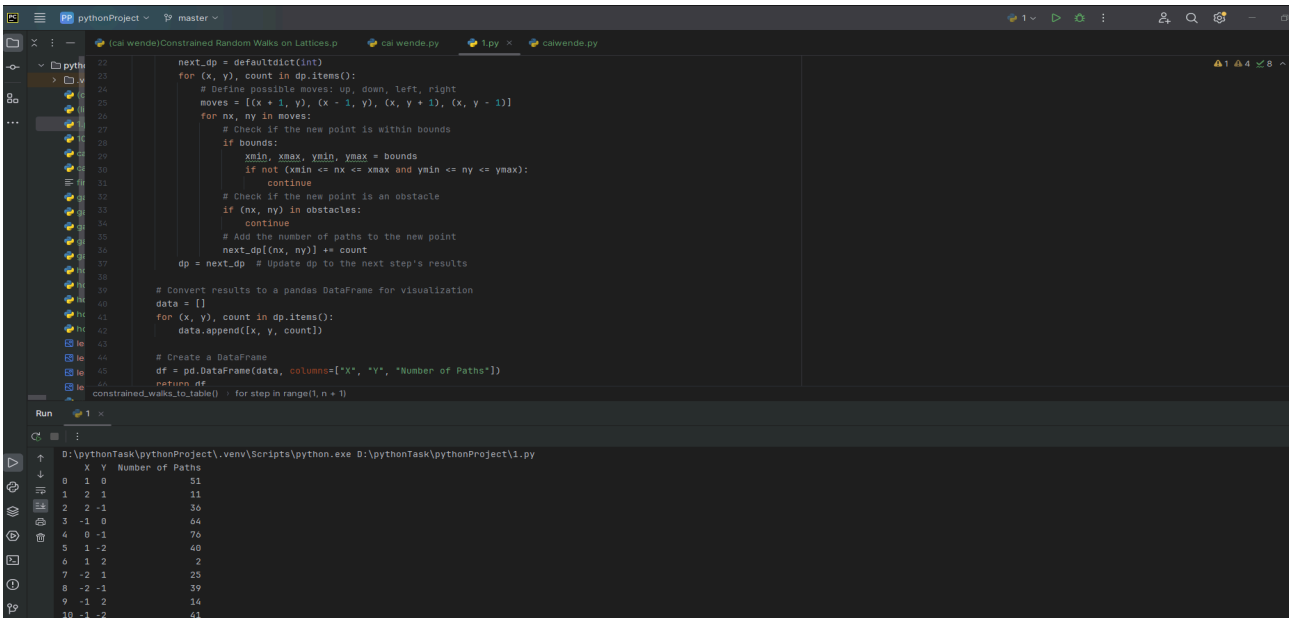
Semanticscholar: Path Counting in Quadrants. Available at https://www.semanticscholar.org/reader/3c1b87567dd0ab5bf76a25428dd10e171308cb9e. Key Content: Focuses on path counting problems within quadrants; provides explicit formulas and references to advanced mathematical results.

Weisstein, E. W. *MathWorld – Random Walks*. Available at https://mathworld.wolfram.com/RandomWalk.html. Relevant Topics: "Random Walks on a Lattice" and "Enumerating Lattice Paths." Key Content: Standard path counting formulas for both lines and planes; explicit solutions for paths with constraints.

Knuth, D. E. (2011). *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley. Relevant Chapters: Chapter 7 (Practical applications of recursions and generating functions) and Chapter 9 (Computational methods for constrained paths). Key Content: Discusses code design and optimization for path counting problems; includes examples for paths in bounded grids.

8. APPENDIX

8.1 Python code




```
import pandas as pd
from collections import defaultdict
def constrained_walks_to_table(n, bounds=None, obstacles=set()):
    """
     Calculate the number of constrained random walk paths and return the result as a table.

    Parameters:
    - n (int): Number of steps in the random walk.
    - bounds (tuple): Bounds of the domain as (xmin, xmax, ymin, ymax).
    - obstacles (set): A set of points (x, y) that are considered obstacles.

    Returns:
    - pd.DataFrame: A DataFrame containing the number of paths to each point.
    # Initialize a dictionary to store the number of paths to each point
    dp = defaultdict(int)
    dp[(0, 0)] = 1  # Starting point has one path
    # Iterate through each step
    for step in range(1, n + 1):
        next_dp = defaultdict(int)
        for (x, y), count in dp.items():
            # Define possible moves: up, down, left, right
            moves = [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)]
            for nx, ny in moves:
                # Check if the new point is within bounds
                if bounds:
                    xmin, xmax, ymin, ymax = bounds
                    if not (xmin <= nx <= xmax and ymin <= ny <= ymax):
                        continue
                # Check if the new point is an obstacle
```

```
                if (nx, ny) in obstacles:
                    continue
                # Add the number of paths to the new point
                next_dp[(nx, ny)] += count
        dp = next_dp  # Update dp to the next step's results

    # Convert results to a pandas DataFrame for visualization
    data = []
    for (x, y), count in dp.items():
        data.append([x, y, count])

    # Create a DataFrame
    df = pd.DataFrame(data, columns=["X", "Y", "Number of Paths"])
    return df

# Example Usage
if __name__ == "__main__":
    # Define number of steps
    steps = 5

    # Define bounds and obstacles
    bounds = (-2, 2, -2, 2)  # Rectangle bounds
    obstacles = {(0, 1), (1, 1)}  # Obstacles at specific points

    # Calculate paths and convert to table
    df_result = constrained_walks_to_table(steps, bounds, obstacles)

    # Display the table
    print(df_result)

    # Save the DataFrame to an Excel file
    file_path = "random_walk_paths.xlsx"
    df_result.to_excel(file_path, index=False)
    print(f"Results saved to {file_path}")
```

8.2 Python running results

- The Excel file will have three columns: "X", "Y", and "Number of Paths", where:
    - "X" and "Y" represent the lattice points.
    - "Number of Paths" shows the number of paths that reach each point after the specified number of steps.

X	Y	Number of Paths
1	0	51
2	1	11
2	-1	36
-1	0	64
0	-1	76
1	-2	40
1	2	2
-2	1	25
-2	-1	39
-1	2	14
-1	-2	41

8.3 Code Introduction

8.3.1 Overview of the Method:

The Python code implements a **constrained random walk** path-counting model, which calculates the number of possible paths from a starting point to a target point under specific constraints. These constraints include:

- **Rectangular boundaries**: The path can only be within a defined rectangular region.
- **Obstacles**: Some grid points are considered obstacles, and the path cannot go through these points.

8.3.2 Basic Method Used in the Code:

- **Recursive Formula**: The code uses a classic recursive method to calculate the number of paths at each step. The number of paths to a point is the sum of the number of paths to its neighboring points.
- **Dynamic Programming (DP)**: To avoid redundant calculations, the code uses dynamic programming to store intermediate results. Each point's path count is stored and updated at each step.
- **Table Output**: The results are stored in a **pandas DataFrame** for easy visualization and export as a table, making it easier to display and analyze the path counts for each point.

8.3.3 Execution Steps of the Code:

Step 1: Initialization

The starting point (0, 0) is initialized with one path, as it's the beginning of the random walk. A dictionary dp is used to store the number of paths to each point.

```
dp = defaultdict(int)
dp[(0, 0)] = 1  # Starting point has one path
```

Step 2: Iterative Path Counting

From step 1 to step n, the number of paths to each point is updated iteratively. For each point, the algorithm checks all possible moves (North, South, East, West) and calculates the number of paths to the neighboring points.

```
for step in range(1, n + 1):
    next_dp = defaultdict(int)
    for (x, y), count in dp.items():
        # Define possible moves: up, down, left, right (NSEW steps)
        moves = [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)]
        for nx, ny in moves:
            # Check if the new point is within bounds
            if bounds and not (bounds[0] <= nx <= bounds[1] and bounds[2] <= ny <=
bounds[3]):
                continue
            # Check if the new point is an obstacle
            if (nx, ny) in obstacles:
                continue
            # Add the number of paths to the new point
            next_dp[(nx, ny)] += count
    dp = next_dp  # Update dp to the next step's results
```

- dp: Stores the number of paths to each point.
- next_dp: Temporarily stores the results for the next step.
- moves: Defines the possible moves from the current point (NSEW steps).

Step 3: Path Update

The path count for each point (x, y) is the sum of the path counts from its neighboring points. The program iterates over all points and updates the path counts for their neighboring points.

Step 4: Handling Boundaries and Obstacles

For each step, the algorithm checks:

- **Boundary**: If the new point lies within the defined rectangular bounds, it will be considered.
- **Obstacles**: If the new point is an obstacle, it will be skipped.

Step 5: Output and Table Storage

Once the calculations are complete, the results are stored in a **pandas DataFrame** and can be exported to an Excel file. Each row represents a point (x, y) with its corresponding path count.

```
# Convert result to DataFrame
data = [(x, y, count) for (x, y), count in dp.items()]
df = pd.DataFrame(data, columns=["X", "Y", "Number of Paths"])

# Export to Excel
df.to_excel("random_walk_paths.xlsx", index=False)
```

return df

- pandas.DataFrame: Converts the path counts into a tabular format, making it easy to display and analyze.
- df.to_excel("random_walk_paths.xlsx", index=False): Saves the results to an Excel file for further use.

8.3.4 Summary of the Execution Steps:

1. **Initialization**: Define the starting point and initialize the path dictionary.
2. **Path Counting**: Use the recursive formula and dynamic programming to update the number of paths to each point at each step.
3. **Boundary and Obstacle Handling**: Ensure that the path calculations stay within the given bounds and avoid obstacles.
4. **Result Output**: Store the results in a table format and export them to an Excel file for easy visualization.

8.3.5 Additional Extensions and Flexibility:

The code can be easily extended to handle different types of step rules, such as:

- **Diagonal steps**: Modify the moves list to include diagonal moves.
- **Knight's moves**: Modify the moves list to handle knight's moves, which can span two squares in one direction and one square in another direction.

For example, here's the code modification to include diagonal steps:

```
# Modify the moves for diagonal steps
moves = [
   (x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1),  # NSEW steps
   (x + 1, y + 1), (x - 1, y - 1), (x + 1, y - 1), (x - 1, y + 1)  # Diagonal steps
]
```

To include **knight's moves**, you would modify the moves list as follows:

```
# Modify the moves for knight's steps (two squares in one direction, one in the other direction)
moves = [
   (x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1),  # NSEW steps
   (x + 1, y + 1), (x - 1, y - 1), (x + 1, y - 1), (x - 1, y + 1),  # Diagonal steps
   (x + 2, y + 1), (x + 2, y - 1), (x - 2, y + 1), (x - 2, y - 1),  # Knight's moves
   (x + 1, y + 2), (x + 1, y - 2), (x - 1, y + 2), (x - 1, y - 2)  # Knight's moves continued
]
```

These modifications allow for greater flexibility in handling different movement rules.

8.3.6 Execution Flow:

1. **Initialization**: Define the starting point and path dictionary.

2. **Iterative Calculation**: For each step, compute the paths to neighboring points and update the path counts.
3. **Boundary & Obstacle Check**: Ensure that the new points are within bounds and not blocked by obstacles.
4. **Table Output**: Convert the results into a table and save it as an Excel file for easy analysis.