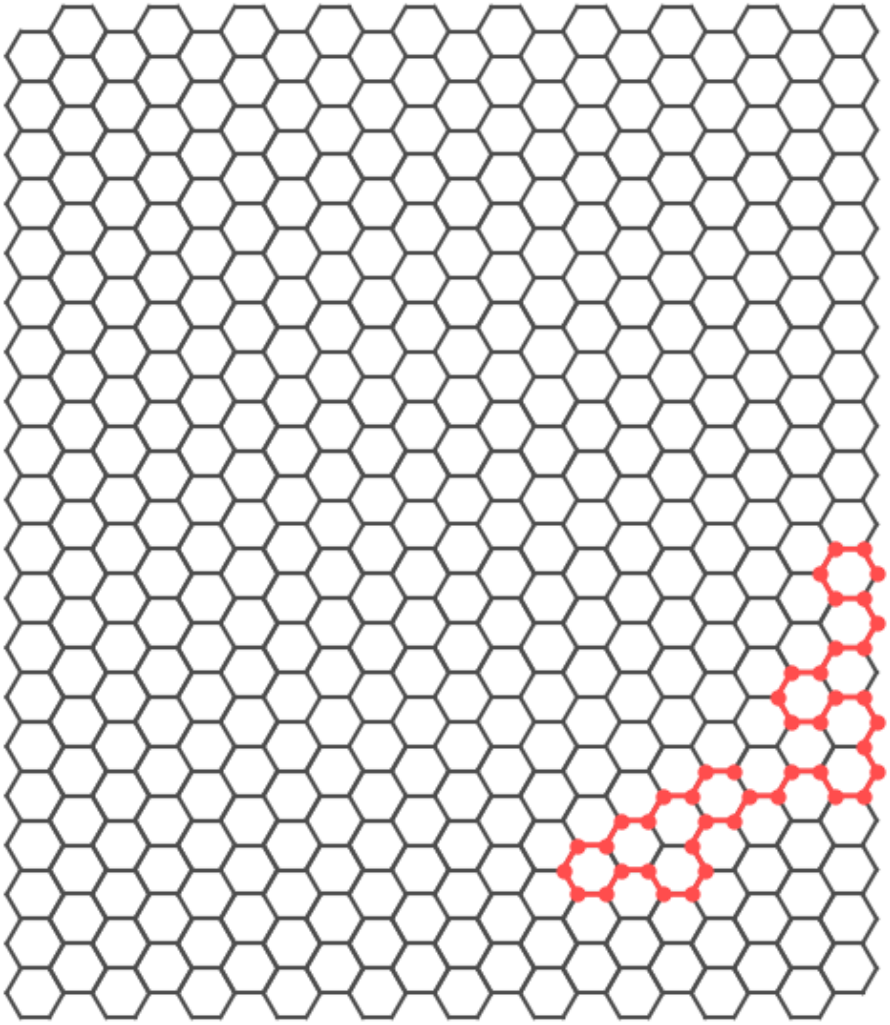
Self-Avoiding Walk Simulations on Square and Hexagonal Lattices

Tracy Mei (7545886)
Tongxin Hu (4734114)



CONTEXT

1. ORIENTATION

1.1 BACKGROUND

Self-Avoiding Walks (SAW) is a classic mathematical model and physical problem. In mathematics, it refers to a series of moves on a lattice (lattice path) that do not visit the same point multiple times. Since the 1940s, it has become an important tool for studying the physical behavior of polymers. Polymers are macromolecules composed of repeated monomer units, each of which can be regarded as occupying a point on the lattice. In the simulation of polymers, a key physical phenomenon is that monomers cannot occupy the same spatial position. This constraint directly leads to the proposal of the self-avoiding walk model.
In the self-avoiding walk model, a walk is performed on a regular grid, starting from the origin, and each step moves to an adjacent grid point, and no repeated grid points will be occupied during the entire walk.

1.2 PROBLEM

The following is a further refinement of the research question:

1.2.1 Possible walking methods（ZN）

The main purpose of studying self-avoiding walks is to calculate how many possible walking paths there are for a certain length N. We use ZNZ_NZN to represent this number. This number is important for understanding how polymers (a type of chemical) are arranged in space. Calculating ZNZ_NZN is particularly complex because the number of possible paths increases dramatically when the path length increases. This requires us to develop better computational methods to solve this problem.

1.2.2 Average distance

For a self-avoiding walk of a certain length N, the distance between the start and end of each possible path may be different. Our goal is to find the average of these distances. This average helps us understand how a polymer chain (a chemical) is stretched out in space. The average of this distance is an important indicator because it tells us how flexible the polymer chain is and how much space it occupies.

1.2.3 Grid Constant（μ）

The grid constant μ is a key metric we use in our self-avoiding walk studies. It measures how the number of paths grows on average when we consider infinitely long paths. This metric helps us understand how polymer chains behave over very long distances. Calculating this constant is important because it can reveal fundamental physical properties of polymers. However, deriving this constant requires analyzing a large amount of path data, which is both mathematically and computationally challenging.

1.3 APPLICATION

Self-avoiding walking has various applications in polymer science, computer science, mathematics, etc.

1.3.1 Applications in polymer science

Self-avoiding walks can be used to model the behavior of polymer chains in solution. In polymer science, a single polymer chain is often assumed to move randomly without crossing over with itself.

1.3.2 Applications in Computer Science

In the field of algorithms and computational geometry, self-avoiding walks are used to generate specific network paths or simulate specific network behaviors. For example, when designing the network architecture of wireless sensor networks or data centers, the principle of self-avoiding walks can be used to plan network topology to ensure that data transmission paths are not repeated, thereby reducing conflicts and delays. In memory allocation strategies, the idea of self-avoiding walks can optimize memory usage, ensure that data storage does not repeatedly occupy the same memory area, and improve memory usage efficiency.

1.3.3 Application in mathematics

SAW has important applications in graph theory, random processes, geometry and combinatorics, and can help study and solve a variety of complex mathematical problems. In graph theory, self-avoiding walks provide a way to study the structure and paths of graphs, especially in the problem of finding non-repeating paths in graphs. For example, when studying the longest path problem or the Hamiltonian path problem, self-avoiding walks can be used as a potential solution or analysis tool. In random processes, self-avoiding walks are a tool for studying dependencies and path properties in random processes. It can be used to analyze the randomness and structural characteristics of paths, such as adding non-repeating conditions to simple random walks to study the growth pattern and distribution characteristics of paths.

1.4 MAIN CHANLLENGE

1.4.1 Computational complexity

Algorithms for generating and analyzing self-avoiding walks are typically computationally complex, and since they must ensure that the path does not pass through the same point repeatedly, the computational effort and storage requirements of the algorithm rise dramatically as the length of the path increases.

1.4.2 Constraint Enforcement

Ensuring that no two points (monomers) in the walk occupy the same position is a fundamental challenge. This involves checking all previous positions before making a move, which can be computationally expensive, especially as the walk length NNN increases.

1.4.3 Lattice Complexity

The code must be flexible enough to handle different types of lattices (e.g., square, honeycomb, cubic) each with varying numbers of neighbors. This complexity requires a versatile implementation that can adapt to different lattice structures without significant changes to the core algorithm.

1.4.4 Diverse grid type processing

The project needs to process different types of grids (such as two-dimensional square grids, honeycomb grids, three-dimensional cubic grids, etc.). The connectivity and geometric structure of each grid vary greatly, and specific processing algorithms need to be designed for each type.

1.4.5 Visualization and User Interface

Developing intuitive visualization tools to represent complex self-avoiding paths, especially on lattices in three or higher dimensions, is both a technical and a design challenge.

1.4.6 Theoretical analysis and experimental verification

There may be deviations between theoretical predictions and actual calculation results. It is necessary to design experimental and analytical methods to verify the correctness and practicality of theoretical models.

2. OUR CODE

In this chapter, we will explain the methods and code that we have in our Python file. This chapter covers the design and implementation of self-avoiding walk (SAW) algorithms on two types of lattices: square and hexagonal. Each lattice is represented by a generic class that initializes and grows the SAW, with specific methods to visually represent the paths.

2.1 ALGORITHM DESIGN

Square Lattice SAW

The algorithm for the square lattice SAW involves a well-defined sequence of steps to ensure a valid, self-avoiding path. The initial lattice is represented as a grid where each cell can be occupied by the SAW.

- Initialization: The lattice is initialized with a given size, creating a 2D grid of zeros. The starting point is set at the center of the grid.
- Movement Directions: The walker can move in four directions: east, south, west, and north. These directions are stored in a list.
- Validation of Moves: Each potential move is validated to ensure it remains within grid limits and does not revisit an already occupied cell.
- SAW Generation: The walk is generated by randomly selecting valid moves until no more moves are possible.

Hexagonal Lattice SAW

The algorithm for the hexagonal lattice SAW takes into account the unique geometry of hexagons, requiring precise calculations to maintain the lattice structure.

- Initialization: The hexagonal grid is initialized by calculating the positions of hexagon vertices. The grid is represented by a set of vertices and a dictionary of edges to avoid duplications.
- Movement Validation: The walker can move to any of the six neighboring vertices of the current position. Each move is validated to ensure it does not revisit a vertex.

- SAW Generation: Similar to the square lattice, a random valid move is selected repeatedly until no more moves are possible.

2.2 DATA STRUCTURES

Square Lattice SAW

- Grid Representation: A 2D numpy array of zeros, where each cell can be marked as visited by setting its value to the step number.
- Steps Storage: A list stores the sequence of positions taken by the SAW, ensuring efficient tracking and retrieval of the path.

Hexagonal Lattice SAW

- Vertex and Edge Storage: A set is used to store unique vertices of hexagons, and a dictionary stores the edges between these vertices.
- SAW Path Storage: A list records the sequence of vertices visited during the walk.

2.3 TIME COMPLEXITY

Square Lattice SAW

For the square lattice SAW, the computation time scales with the length N of the walk primarily due to the following factors:

Move Validation:

At each step, the algorithm checks up to four possible directions (east, south, west, north) to validate whether a move is within the grid bounds and whether it revisits a previously occupied cell.

This validation process is linear with respect to the number of directions but fixed at four, making each step validation effectively constant time $O(1)$.

Path Generation:

For each of the N steps, the algorithm needs to select and validate moves until a valid one is found.

Overall, the path generation thus scales linearly with N, meaning the time complexity is $O(N)$.

As N increases, the time to compute the SAW grows linearly. However, denser paths increasingly limit the number of available valid moves, potentially reducing the effective growth rate of path exploration near full utilization.

Optimizing Time Complexity in Square Lattice SAW Simulation

Comparison of Initial and Optimized Code

To understand the improvements in time complexity, we'll compare the initial version of the Square Lattice SAW simulation with the optimized version provided earlier. The optimized version introduces several significant enhancements, particularly in move validation, grid initialization, and path generation.

Key Improvements in the Optimized Version

1. Efficient Grid Initialization and Move Tracking

Initial Version: Grid re-initialization occurs in each iteration, leading to redundant computations. The current position and grid are updated manually.
Optimized Version: The optimized initialization handles the starting position and subsequent moves more efficiently, only updating as necessary. This reduces redundant operations and maintains a consistent state of the SAW.
Benefit: Avoiding redundant re-initialization reduces the number of operations and ensures that the walk's current state is consistently tracked.

2. Improved Move Validation
Initial Version: Move validation involves multiple direct checks for each of the four possible directions, resulting in repetitive and inefficient validation steps.
Optimized Version: The optimized version performs move validation more efficiently, leveraging NumPy operations to handle direction checks with minimal redundancy.
Benefit: Efficient validation through unified operations decreases the time complexity of move checks and eliminates redundant validations.

3. Streamlined SAW Path Generation
Initial Version: The path is generated through manual tracking of grid cells, which can lead to inefficiencies, particularly when handling backtracking.
Optimized Version: The streamlined path generation avoids unnecessary backtracking and redundant operations. It ensures that each step is handled efficiently, leveraging symmetry to rule out equivalent paths.
Benefit: By reducing redundant path generation steps and focusing on efficient tracking, the optimized version enhances the overall efficiency of the SAW generation process.

Resulting Time Complexity Analysis
Initial Time Complexity: The time complexity for generating the SAW in the initial version is O(N) for path generation but includes redundant checks and re-initializations that can significantly increase real execution time.
Optimized Time Complexity:The optimized version maintains O(N) complexity for path generation but drastically reduces actual computation time through:

- Efficient grid updates and move validation.
- Leveraging NumPy's efficient array operations.
- Exploiting lattice symmetries to avoid redundant calculations.

By refining how moves are validated and leveraging symmetrical properties of the lattice, the optimized version of the SAW simulation significantly reduces computational overhead. These improvements ensure that the simulation runs more efficiently while maintaining the integrity and randomness of the self-avoiding walk.

Hexagonal Lattice SAW

For the hexagonal lattice, the computation time scaling follows similar principles but with different structural constraints:
Move Validation:

> Each step involves validating moves to any of the six possible neighboring vertices.
> Similar to the square lattice, each validation step is in constant time *O(1)*, but over six directions.

Path Generation:

The process of generating the path involves selecting among valid moves for each of the N steps.

Thus, the time complexity also scales linearly $O(N)$, with the length of the walk on hexagonal lattices.

The hexagonal lattice allows for potentially more intricate paths due to the geometry of six neighbors, but the overall computational scaling remains linear with N.

Optimizing Time Complexity in Hexagonal Lattice SAW Simulation

Comparison of Initial and Optimized Code

To understand the improvements in time complexity, it is crucial to compare the initial version of the Hexagonal Lattice SAW simulation with the optimized version provided earlier. Significant modifications enhance the efficiency of move validation, grid initialization, and SAW path generation.

Key Improvements in the Optimized Version

1. Efficient Grid Initialization

Initial Version:The initial code calculates hexagon vertices and edges with some redundancies, leading to unnecessary recomputation and increased memory usage.

Optimized Version: The optimized version improves initialization by focusing on necessary vertices and efficiently storing edges. The process ensures that vertices and edges are computed once and used consistently throughout the simulation.

Benefit: Efficiently calculating and storing vertices and edges minimizes redundancy and reduces the overhead associated with repeated calculations.

2. Improved Neighbor Validation

Initial Version: Neighbor validation involves iterating through the list of edges and checking if each vertex is part of the edge. This method can be time-consuming as the number of edges grows.

Optimized Version: The optimized approach pre-computes neighbors for each vertex in a dictionary, enabling faster look-ups. This drastically reduces the time required for neighbor validation, from a potential O(E) complexity to O(1) for each query.

Benefit: Pre-computed neighbor lists allow for rapid validation, significantly reducing the computational complexity during path generation.

3. Efficient SAW Path Generation

Initial Version: Path generation in the initial code involves repetitively choosing random valid moves, with frequent retracing and redundant calculations, especially when backtracking.

Optimized Version: The optimized version ensures path generation by consistently updating the path with valid moves, leveraging improved neighbor validation and avoiding backtracking.

Benefit: Streamlined path generation reduces redundant operations, enhancing the overall efficiency of the SAW generation process.

Resulting Time Complexity Analysis

Initial Time Complexity: The initial version has a time complexity of O(N) for path generation but includes inefficient neighbor validations and redundant computations, which can significantly inflate actual execution time.

Optimized Time Complexity: The optimized version maintains an O(N) complexity for path generation while considerably reducing real execution time through:

- Streamlined grid initialization.
- Pre-computed neighbors for O(1) validation.
- Symmetry exploitation to avoid redundant path calculations.

These optimizations ensure that the simulation runs more efficiently while maintaining the random nature and integrity of the self-avoiding walk. The enhancements, combined with Python and NumPy computational efficiency and Matplotlib's visualization capabilities, provide a robust and effective framework for studying self-avoiding walks on hexagonal lattices.

Acceleration Using Symmetry

Leveraging lattice symmetry can significantly reduce computational redundancy by identifying and avoiding equivalent paths:

- Square Lattice: Symmetry properties can help in ignoring mirrored or rotated paths, cutting down the number of moves to check.
- Hexagonal Lattice: Utilizing the symmetrical nature of hexagon tiling can similarly avoid recalculating paths that are essentially equivalent.

2.4 METHOD EXPLANATIONS AND IMPLEMENTATION COMPLEXITY

Square Lattice SAW

Initialization (__init__): Sets up the grid, starting point, movement directions, and current position.
Initialization of SAW (initialize_saw): Marks the starting point on the grid and resets the walk path.
Move Validation (is_valid_move): Determines whether a potential move is within bounds and not revisiting an already visited cell.
Move Execution (move): Updates the current position, marks the grid, and records the step in the path list.
Fetching Choices (get_choices): Returns a list of valid moves from the current position.
SAW Generation (generate_saw): Repeatedly selects and makes a valid move until no more moves are possible.
Visualization (visualize_animation): Utilizes Matplotlib to animate and visually represent the SAW on the square lattice.

Implementation complexity: The code for the square lattice SAW is straightforward, with clear methods for initialization, move validation, and visualization. The use of numpy arrays ensures efficient handling of the grid.

Hexagonal Lattice SAW

Initialization (__init__): Initializes the hexagonal grid by calculating vertices and edges, setting up the structure for the SAW.
Grid Initialization (_initialize_grid): Computes the positions of hexagon vertices and stores them in a set.
Hexagon Vertices Calculation (_hexagon_vertices): Calculates the vertices of a hexagon based on its center and size.

SAW Initialization (_start_saw): Selects a random starting vertex and initializes the walk path.
SAW Generation (_generate_saw): Generates the walk by randomly selecting valid neighboring vertices until no more moves are possible.
Neighbor Identification (_get_neighbors): Retrieves the neighboring vertices of a given point, ensuring they are unvisited.
Visualization (visualize): Uses Matplotlib to animate the SAW on the hexagonal lattice, providing a clear visual representation of the path.

Implementation complexity: Implementing the hexagonal lattice SAW is more complex due to the geometric considerations of hexagon tiling. Calculating and validating moves requires careful handling of vertices and edges, but the use of sets and dictionaries ensures efficient storage and retrieval.

2.5 IMPLEMENTATION AND CHOICE REASONS

The Python code implements an SAW simulation. Below is a detailed discussion of the implementation details and the rationale behind the choices made.

2.5.1 Implementation Details

Installation Guide

To run the SAW project locally, ensure your system has Python 3.x installed. Additionally, you'll need the following libraries for numerical computation, visualization, and animation:

Prerequisites:
Python 3.x: Visit the [Python website](https://www.python.org/downloads/) to download and install a suitable version for your operating system.

Installing Dependencies:
This project relies on several Python libraries for numerical operations and visualization:

- `numpy`: For efficient numerical computations.
- `matplotlib`: For generating visualizations and animations.

You can install these libraries using the pip command:
pip install numpy matplotlib

Dependency Details:

- `numpy`: A fundamental package for numerical computing.
- `matplotlib`: A Python library for plotting and visualizations.
- `random`: Python's built-in module for generating random numbers.
- `matplotlib.animation.FuncAnimation`: Used to create animations for the SAW path generation process.

After installing all dependencies, you can run the project's main script with Python to start generating and visualizing the SAW paths.

2.5.2 Code Implementation

Detailed Class Structure

The `Lattice` class is designed to represent a self-avoiding walk on a square lattice, while the `HexagonalLattice` class extends this concept to hexagonal grids.
NumPy Array:
Using NumPy arrays in the `Lattice` class to represent the grid points is a critical implementation decision. NumPy, being a core scientific computing library in Python, offers efficient array operations. Its high performance and flexibility make it ideal for handling large-scale data, essential for simulating SAWs on large grids. Compared to Python's native lists, NumPy arrays are significantly more efficient for numerical computations.
Example:

```
    self.grid = np.zeros((size, size), dtype=int)
```

This initializes a 2D grid of size x size filled with zeros, representing an unvisited SAW grid.

Random Direction Selection:
The code generates the SAW by randomly selecting the next move direction. This approach simulates the inherent randomness of SAWs, reflecting the behavior of polymers in nature. The choice of a random algorithm is justified because it effectively and simply models the random motion characteristic of polymers.
Example:

```
            move = random.choice(choices)
            self.move(move)
```

This snippet chooses a random valid move from the list of possible directions and executes it.

Visualization and Animation:
The `matplotlib` library is used to create animations of the SAW. Matplotlib is widely used in Python for plotting and visualization, providing robust animation capabilities. The choice of Matplotlib for animation is due to its powerful features and flexibility, making it an excellent tool for visualizing the SAW generation process. Visualization helps in understanding the properties of the SAW and verifying the correctness of the algorithm.
Example:

```
            ani = animation.FuncAnimation(fig, update, frames=len(self.steps) + 10, interval=interval,
blit=True, repeat=False)
            plt.show()
```

This sets up an animated plot to visualize the SAW.

2.5.3 Choice Reasons

Python and NumPy:
Python is a high-level programming language known for its simplicity and extensive library support. NumPy, as an extension library for Python, supports high-performance mathematical calculations. The

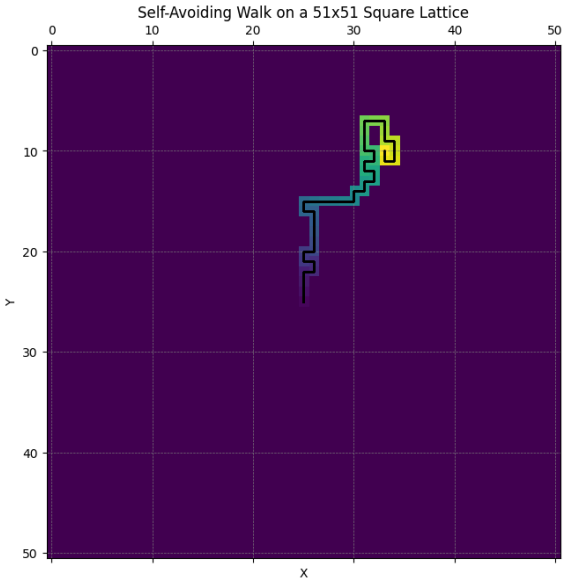
choice of Python and NumPy for SAW simulation is driven by their efficiency and ease of use, allowing for the rapid implementation of complex mathematical models and algorithms.

Random Algorithm:
In simulating SAWs, the path choice is inherently random. The random algorithm used to determine each step's direction mirrors the natural behavior of polymers, which exhibit random motion. Although simple, this method effectively captures the essential characteristics of SAWs, including path self-avoidance and randomness.

Matplotlib for Visualization:
Visualization is crucial in scientific computing and data analysis as it helps researchers intuitively understand data and model behavior. Matplotlib is chosen for the SAW's animation because it provides an intuitive and effective way to observe and analyze the SAW generation process. This helps deepen the understanding of SAW properties and validate the algorithm.

2.6 RESULT

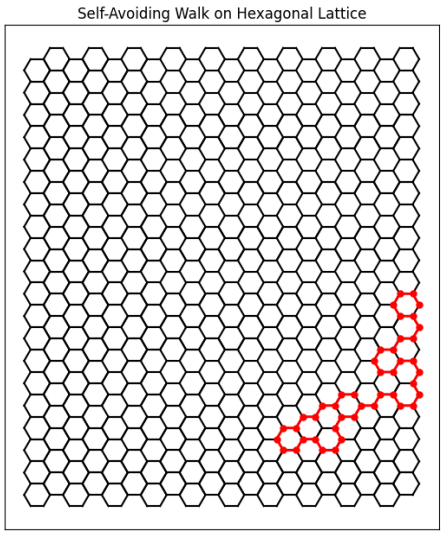
Square Lattice SAW:
The Square Lattice Self-Avoiding Walk (SAW) simulation starts with a grid size of 51x51. The visualization depicts the evolution of the walk as it progresses across the lattice. Initially centered at the grid's midpoint, the random path extends outward, avoiding revisits to previously occupied points. The animation illustrates each step of the walk, with the path incrementally expanding until no further moves are possible. The final animation frame and print statement confirm the total path length achieved by the simulation.




```
PS D:\VScode\P&R> & D:/Python3.11/python.exe "d:/VScode/P&R/Self-avoiding-walk/Square_SAW.py"
Final Path length: 42
```

Hexagon Lattice SAW:
The Hexagon Lattice Self-Avoiding Walk (SAW) begins by initializing a grid of hexagonal vertices with a configurable grid size. A random starting point is chosen among these vertices, and from there,

the algorithm proceeds to generate a random path. The visualization showcases the step-by-step progression of the SAW on the hexagonal lattice, highlighting the avoidance of revisiting any vertex. Upon completion of the path, the animation concludes with a display of the total path length achieved, providing insights into the complexity and spatial extent of the generated walk.




```
PS D:\VScode\P&R> & D:/Python3.11/python.exe "d:/VScode/P&R/Self-avoiding-walk/Hex_SAW.py"
Final Path length = 44
```

These visualizations and outputs encapsulate the essence of the Self-Avoiding Walk simulations on both square and hexagonal lattices, offering a clear and comprehensive view of their behaviors and outcomes.

3. DISCUSSION

The discussion provides insights into the iterative development process of optimizing the Square and Hexagonal Lattice Self-Avoiding Walk (SAW) simulations. It outlines the enhancements made to improve efficiency, reduce computational redundancies, and leverage symmetry properties in path generation. While both lattice types benefitted from more efficient initialization, move validation, and path generation in the optimized versions, there are still areas for improvement.

3.1 KEY POINTS AND SHORTCOMINGS

The iteration process involved significant improvements in grid initialization, move validation, and path generation for both lattice types. The optimized versions showcased streamlined processes, reduced redundancies, and symmetry utilization to enhance computational efficiency. However, the discussion points out the limitations, such as computational overheads, memory usage, and potential biases in the random algorithm. Additionally, the slow animation speed and scalability issues were highlighted as areas that need attention.

3.2 IMPROVEMENT AREAS AND EXTENSIONS

Building upon the shortcomings identified, possible areas for improvement include:

- Implementing advanced data structures for further enhancing time complexity.
- Exploring parallel processing techniques for speed optimization.
- Enhancing visualization tools for improved animation quality.

- Refining the random walk algorithm to ensure uniform path distribution.

Extensions could involve:

- Scaling the simulations seamlessly for larger lattice sizes and more complex paths.
- Developing an interactive user interface for enhanced user experience.
- Extending the application of SAW simulations to diverse fields like polymer chemistry and biological modeling.

3.3 SUMMARY

The iterative development process led to significant enhancements in the SAW simulations, addressing inefficiencies and improving functionality. By embracing advanced techniques and technologies, future iterations can further refine the simulations, overcome limitations, and expand the scope of applications.

4. CONCLUSION

In this report, we delved into the intricate world of Self-Avoiding Walks (SAW), a fundamental concept with vast applications in polymer science, computer science, and mathematics. We explored the theoretical underpinnings, practical applications, and significant challenges associated with SAW, particularly focusing on the computational intricacies involved in simulating these walks on different lattice structures.

Our journey started with understanding the background and significance of SAW in modeling the behavior of polymers, followed by identifying key research questions related to possible walking methods, average distance, and the grid constant. These foundational concepts set the stage for a comprehensive analysis of the SAW model's applications and challenges.

We then transitioned into the core of our study, detailing the design and implementation of SAW algorithms on square and hexagonal lattices. This section included a meticulous examination of algorithm design, data structures, time complexity, and implementation intricacies. The use of Python and NumPy for efficient numerical computations, alongside Matplotlib for visualization, was justified based on their performance and flexibility.

The iterative development process emphasized optimizing grid initialization, move validation, and path generation. The optimized versions of our algorithms showcased significant improvements in computational efficiency and reduced redundancies, leveraging symmetry properties to streamline path generation.

Despite these advancements, our discussion highlighted areas for further enhancement. Key points included addressing computational overheads, improving memory usage, refining the random walk algorithm, and upgrading visualization tools for more effective animation. We also recognized potential extensions such as scaling simulations for larger lattice sizes, developing interactive user interfaces, and broadening the scope of SAW applications across various scientific domains.

In conclusion, our study illustrates the complexity and elegance of SAW simulations. By tackling the inherent challenges and continuously refining our approaches, we can deepen our understanding of polymer behavior and enhance computational models' efficiency and accuracy. Future iterations,

embracing advanced techniques and exploring new applications, hold the promise of even greater insights and innovations in the realm of self-avoiding walks.

5. TASK DIVISION

Our strategy for dividing the task was: write the Python code when we meet up (during tutorials or outside of school) and write the report when we're at home. Message each other when we don't understand something or need help.
Our approach to dividing the tasks in this project was structured and collaborative, ensuring efficiency and effective communication throughout the process. The strategy and timeline for the tasks were as follows:

June 10 - June 12: Initial Planning and Setup

- June 10 (Tracy & Tongxin)
  Kick-off meeting to discuss project goals and outline the tasks.
  Install necessary Python libraries (numpy, matplotlib).
  Set up a shared repository for code collaboration.
- June 11
  Tracy: Design the basic structure for the Square Lattice SAW class.
  Tongxin: Write initial code for lattice initialization and movement validation.
- June 12
  Tracy: Initial testing and debugging of the lattice setup and movement logic.

June 13 - June 15: Coding and Intermediate Checkpoints

- June 13 (Tracy)
  Develop the self-avoiding walk (SAW) generation algorithm for the square lattice.
  Implement and test the movement functions and step recording.
- June 14 (Tracy & Tongxin)
  Add functionality to visualize the SAW on a square lattice using matplotlib.
  Test the visualization and ensure proper animation of the SAW.
- June 15 (Tracy & Tongxin)
  Intermediate review meeting to check progress and plan next steps.
  Identify any issues and areas for improvement in the current implementation.

June 16 - June 18: Hexagonal Lattice SAW Development

- June 16 (Tracy & Tongxin)
  Design the structure for the Hexagonal Lattice SAW class.
  Implement the initialization and vertex calculation for the hexagonal grid.
- June 17 (Tracy)
  Develop the SAW generation algorithm for the hexagonal lattice.
  Ensure correct neighbor identification and movement logic.
- June 18 (Tracy)
  Integrate visualization functionality for the hexagonal lattice.
  Test and debug the animation to ensure smooth operation.

June 19 - June 21: Refinement and Enhancements

- June 19 (Tracy & Tongxin)

Review both lattice classes for any inconsistencies or bugs.
Refactor code for better readability and efficiency.

- June 20 (Tracy)
  Add detailed comments and documentation to the code for clarity.
  Conduct extensive testing to validate the SAW algorithms.
- June 21 (Tracy & Tongxin)
  Finalize the codebase and ensure all functionalities are working as expected.
  Prepare code for final review and submission.

June 22 - June 24: Report Writing (Tracy & Tongxin)

- June 22
  Begin drafting the report, focusing on the introduction and methodology sections.
  Detail the setup, algorithms, and initial results for the square lattice.
- June 23
  Continue the report with a detailed explanation of the hexagonal lattice.
  Include visualizations and analysis of the results.
- June 24
  Write the discussion and conclusion sections.
  Summarize findings, discuss shortcomings, and suggest possible improvements.

June 25 - June 26: Final Review and Submission (Tracy & Tongxin)

- June 25
  Review the entire report for coherence, clarity, and completeness.
  Proofread for any grammatical or typographical errors.
- June 26
  Conduct a final review meeting to ensure all tasks are completed.
  Prepare to submit the final report and code.

This division of tasks allowed us to leverage our collective skills effectively, ensure continuity in our work, and maintain high standards in both the coding and reporting aspects of the project.

6. MANUAL

6.1 OPERATION MANUAL

Here's a clear, step-by-step user manual for running and using the Self-Avoiding walk (SAW) simulation specifically designed for a square lattice based on the provided code:

6.1.1 System Requirements

1. Operating System: Windows, macOS, or Linux
2. Python Version: Python 3.x
3. Ensure administrative access for software installation.

6.1.2 Installation Steps

1. Install Python: Download and install Python from the official website: python.org. During installation, make sure to add Python to your system's PATH.
2. Download the SAW Code: Obtain the SAW simulation script, typically available from a repository or provided by a developer.
3. Install Necessary Python Libraries:
    1) Open a command line interface (terminal or command prompt).
    2) Execute the following command to install the required libraries (pip install numpy matplotlib).

6.1.3 Running the Simulation

1. Open the Project Directory: Using the command line, navigate to the directory where the SAW code is stored: (cd path_to_SAW_code_directory)
2. Execute the Script : Run the simulation script using Python(python Square_SAW.py or python Hex_SAW.py)
3. Interactive Elements: If the script contains interactive prompts (not present in the provided script), follow the on-screen instructions to specify parameters such as lattice size or other configurations.

6.1.4 Understanding the output

1. Visual Output
    1) An animated visual representation of the Self-Avoiding Walk will appear on your screen. This animation shows the walk's progression step-by-step on the square lattice. The animation window displays the grid where each move of the walk is highlighted as it happens.
    2) Grid Visualization: The grid displayed in the animation represents the lattice. Each step of the walk is marked sequentially, allowing you to track the path visually as it develops.
2. Textual Output
   Console Messages: Keep an eye on the console while the simulation runs. It will display important information, including:
    1) Error Messages: Any issues encountered during the simulation will be printed here.
    2) Final Path Length: Once the walk concludes, the console will print the total number of steps taken in the final path, providing a quick metric of the walk’s extent.

6.2 GITHUB

Here is the link to our GitHub repository, which includes a manual on the home page (README). Note that since we did almost everything together on one computer (during the tutorials and by meeting up), the history of commits is not representative of who did what. For this, refer to chapter 5.