1 Kod Źródłowy

Eksperymenty zostały zaimplementowane z języku `C++`. Do generowania liczb pseudo losowych użyto generatora `Mersenne Twister`. Grafy są reprezentowane za pomocą list sąsiedztwa. Za generowanie wykresów odpowiedzialna jest funkcja napisana w języku `Python`.

1 Klasa generująca losowe liczby naturalne

```cpp
// RandomNumberGenerator.hpp
// class wrapping radom number generator
// it returns positive integers from given range

#ifndef RANDOM_NUMBER_GENERATOR_HPP
#define RANDOM_NUMBER_GENERATOR_HPP

#include <random>

class RandomNumberGenerator {
private:
    std::mt19937 generator;

public:
    RandomNumberGenerator();
    RandomNumberGenerator(int seed);

    unsigned int get(unsigned int lowerBound, unsigned int upperBound);
};

#endif // RANDOM_NUMBER_GENERATOR_HPP
```

```cpp
// RandomNumberGenerator.cpp

#include "RandomNumberGenerator.hpp"

RandomNumberGenerator::RandomNumberGenerator() : generator(std::random_device{}()) { }

RandomNumberGenerator::RandomNumberGenerator(int seed) : generator(seed) { }

unsigned int RandomNumberGenerator::get(unsigned int lowerBound, unsigned int upperBound) {
    std::uniform_int_distribution<unsigned int> dis(lowerBound, upperBound);

    return dis(generator);
}
```

1 Klasy reprezentujące grafy różnych typów

```cpp
// Graph.hpp
// classes representing different types of undirected graph
// every class can simulate random walk and calculate coverage time

#ifndef GRAPH_HPP
#define GRAPH_HPP

#include <vector>
#include <queue>
#include <ostream>

#include "RandomNumberGenerator.hpp"

class Graph {
protected:
    unsigned int verticesNumber;
    std::vector<std::vector<unsigned int>> adjacencyList;

public:
    Graph(unsigned int n);
    unsigned int runRandomWalkWithCoverage(unsigned int startVertex, RandomNumberGenerator& gen);

    friend std::ostream& operator<<(std::ostream& os, const Graph& graph);

private:
    unsigned int getRandomNeighbour(unsigned int vertex, RandomNumberGenerator& gen);
};


class Clique : public Graph {
public:
    Clique(unsigned int n);
};


class PathGraph : public Graph {
public:
    PathGraph(unsigned int n);
};


class LollipopGraph : public Graph {
public:
    LollipopGraph(unsigned int n);
};


class CompleteBinaryTree : public Graph {
public:
    CompleteBinaryTree(unsigned int n);
};


#endif // GRAPH_HPP
```

```cpp
// Graph.cpp

#include "Graph.hpp"

Graph::Graph(unsigned int n) :
    verticesNumber(n), adjacencyList(n, std::vector<unsigned int>()) { }

unsigned int Graph::runRandomWalkWithCoverage(unsigned int startVertex, RandomNumberGenerator& gen) {
    std::vector<unsigned int> visited(verticesNumber, 0);
    visited[startVertex] = 1;
    unsigned int toVisit = verticesNumber - 1;
    unsigned int moves = 0;
    unsigned int lastVertex = startVertex;

    while(toVisit > 0) {
        lastVertex = getRandomNeighbour(lastVertex, gen);
        if(visited.at(lastVertex) == 0) {
            visited.at(lastVertex) = 1;
            toVisit--;
        }

        moves++;
    }

    return moves;
}

unsigned int Graph::getRandomNeighbour(unsigned int vertex, RandomNumberGenerator& gen) {
    const std::vector<unsigned int> neighbours = adjacencyList.at(vertex);

    return neighbours.at(gen.get(0, neighbours.size() - 1));
}

std::ostream& operator<<(std::ostream& os, const Graph& graph) {
    for(unsigned int v1 = 0; v1 < graph.verticesNumber; v1++) {
        os << v1 << ": ";
        for(const unsigned int& neighbour : graph.adjacencyList.at(v1)) {
            os << neighbour << " ";
        }
        os << "\n";
    }

    return os;
}


Clique::Clique(unsigned int n) : Graph(n) {
    for(unsigned int v1 = 0; v1 < n; v1++) {
        for(unsigned int v2 = v1+1; v2 < n; v2++) {
            adjacencyList.at(v1).push_back(v2);
            adjacencyList.at(v2).push_back(v1);
        }
    }
}
```

```
PathGraph::PathGraph(unsigned int n) : Graph(n) {
    for(unsigned int v = 0; v < (n - 1); v++) {
        adjacencyList.at(v).push_back(v + 1);
        adjacencyList.at(v + 1).push_back(v);
    }
}
```

```
LollipopGraph::LollipopGraph(unsigned int n) : Graph(n) {
    unsigned int cliqueSize = 2 * n / 3;
    unsigned int pathSize = n - cliqueSize;

    for(unsigned int v1 = 0; v1 < cliqueSize; v1++) {
        for(unsigned int v2 = v1+1; v2 < cliqueSize; v2++) {
            adjacencyList.at(v1).push_back(v2);
            adjacencyList.at(v2).push_back(v1);
        }
    }

    for(unsigned int v = cliqueSize - 1; v < (n - 1); v++) {
        adjacencyList.at(v).push_back(v + 1);
        adjacencyList.at(v + 1).push_back(v);
    }
}
```

```
CompleteBinaryTree::CompleteBinaryTree(unsigned int n) : Graph(n) {
    std::queue<int> vertices;
    vertices.push(0);

    while (!vertices.empty()) {
        int v = vertices.front();
        vertices.pop();
        unsigned int v1 = 2 * v + 1;
        unsigned int v2 = 2 * v + 2;

        if (v1 < n) {
            adjacencyList.at(v).push_back(v1);
            adjacencyList.at(v1).push_back(v);
            vertices.push(v1);
        }
        if (v2 < n) {
            adjacencyList.at(v).push_back(v2);
            adjacencyList.at(v2).push_back(v);
            vertices.push(v2);
        }
    }
}
```

1 Program przeprowadzającay eksperyment

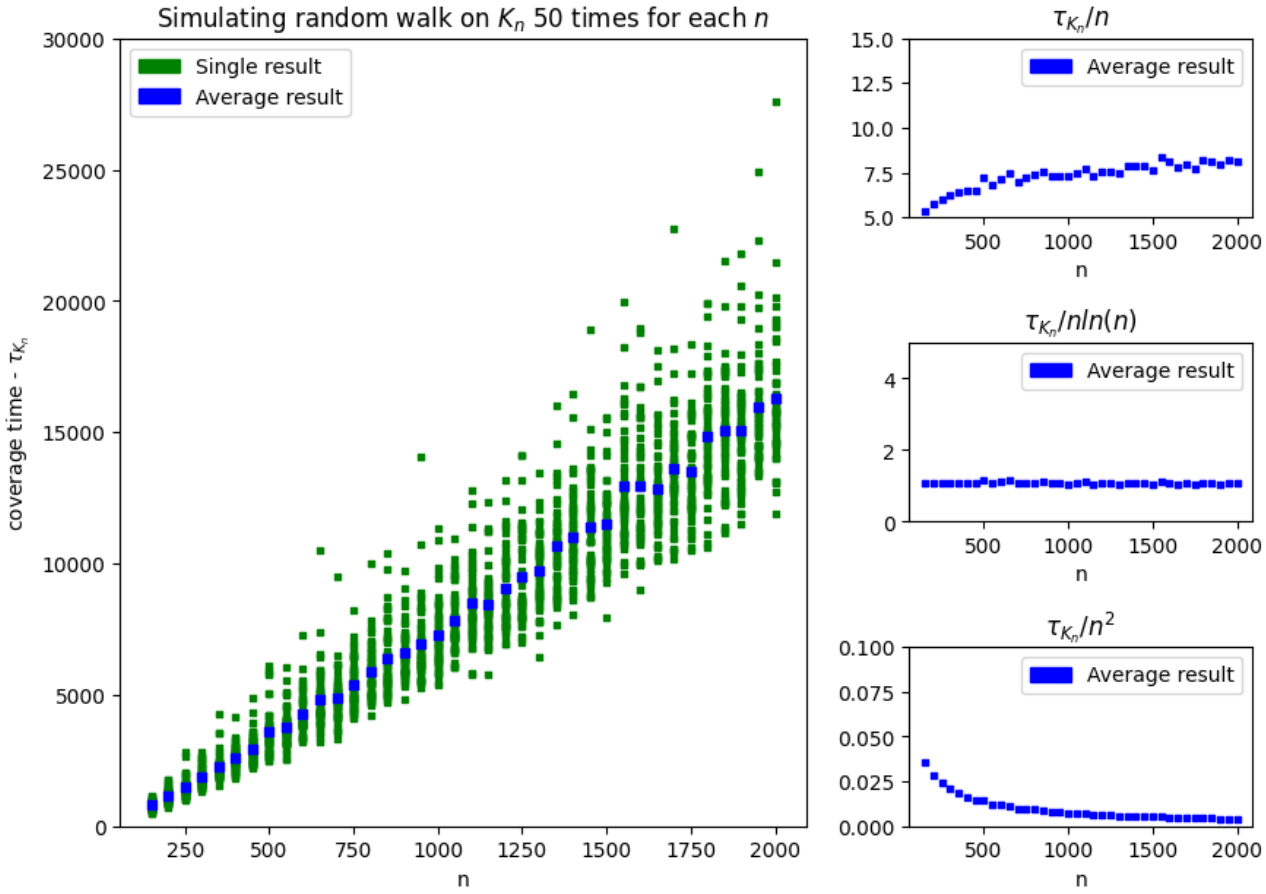
```cpp
#include <iostream>
#include <fstream>

#include "RandomNumberGenerator.hpp"
#include "Graph.hpp"

#define K       100
#define N_MIN   100
#define N_MAX   2000
#define N_STEP  50

int main() {
    ofstream output;
    output.open("lollipop.txt");

    RandomNumberGenerator randomNumberGenerator = RandomNumberGenerator();

    for(unsigned int n = N_MIN; n <= N_MAX; n += N_STEP) {
        Graph graph = LollipopGraph(n);
        output << n;
        for(unsigned int k = 0; k < K; k++) {
            output << ", " << graph.runRandomWalkWithCoverage(0, randomNumberGenerator);
        }
        output << "\n";
    }

    output.close();

    return 0;
}
```

1 Funkcja generująca wykresy

```
import pandas as pd

import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

import math

def generate_chart(path_to_data, functions_to_compare, titles, yranges):
    df = pd.read_csv(path_to_data)
    data = np.array(df.values)

    fig = plt.figure()
    gs = fig.add_gridspec(len(functions_to_compare), 2, width_ratios=[2,1])
    fig.add_subplot(gs[:, 0])

    xs = [series[0] for series in data]
    ys = [series[1:] for series in data]
    avgs = [sum(y) / len(y) for y in ys]


    for x, y, a in zip(xs, ys, avgs):
        plt.scatter([x] * len(y), y, color='g', marker='s', s=10)
        plt.scatter(x, a, color='b', marker='s', s=15)

    plt.title('Simulating random walk on ${}$ $50$ times for each $n$'.format(titles[0]))
    plt.xlabel('n')
    plt.ylabel(r"coverage time - $\tau_{" + titles[0] + "}$")
    plt.ylim(*yranges[0])

    greenExtra = Rectangle((0, 0), 1, 1, color='green')
    blueExtra = Rectangle((0, 0), 1, 1, color='blue')
    plt.legend([greenExtra, blueExtra], ['Single result', 'Average result'])

    for index, fun in enumerate(functions_to_compare):
        fig.add_subplot(gs[index, 1])

        ys = [avg / fun(n) for avg,n in zip(avgs, xs)]
        plt.scatter(xs, ys, color='b', marker='s', s=10)

        plt.title(titles[index + 1])
        plt.ylim(*yranges[index + 1])
        plt.xlabel('n')
        plt.legend([blueExtra], ['Average result'])

    plt.subplots_adjust(hspace=0.7)
    plt.rcParams["figure.figsize"] = (10, 7)
    plt.show()
```

2 Wyniki Eksperymentów

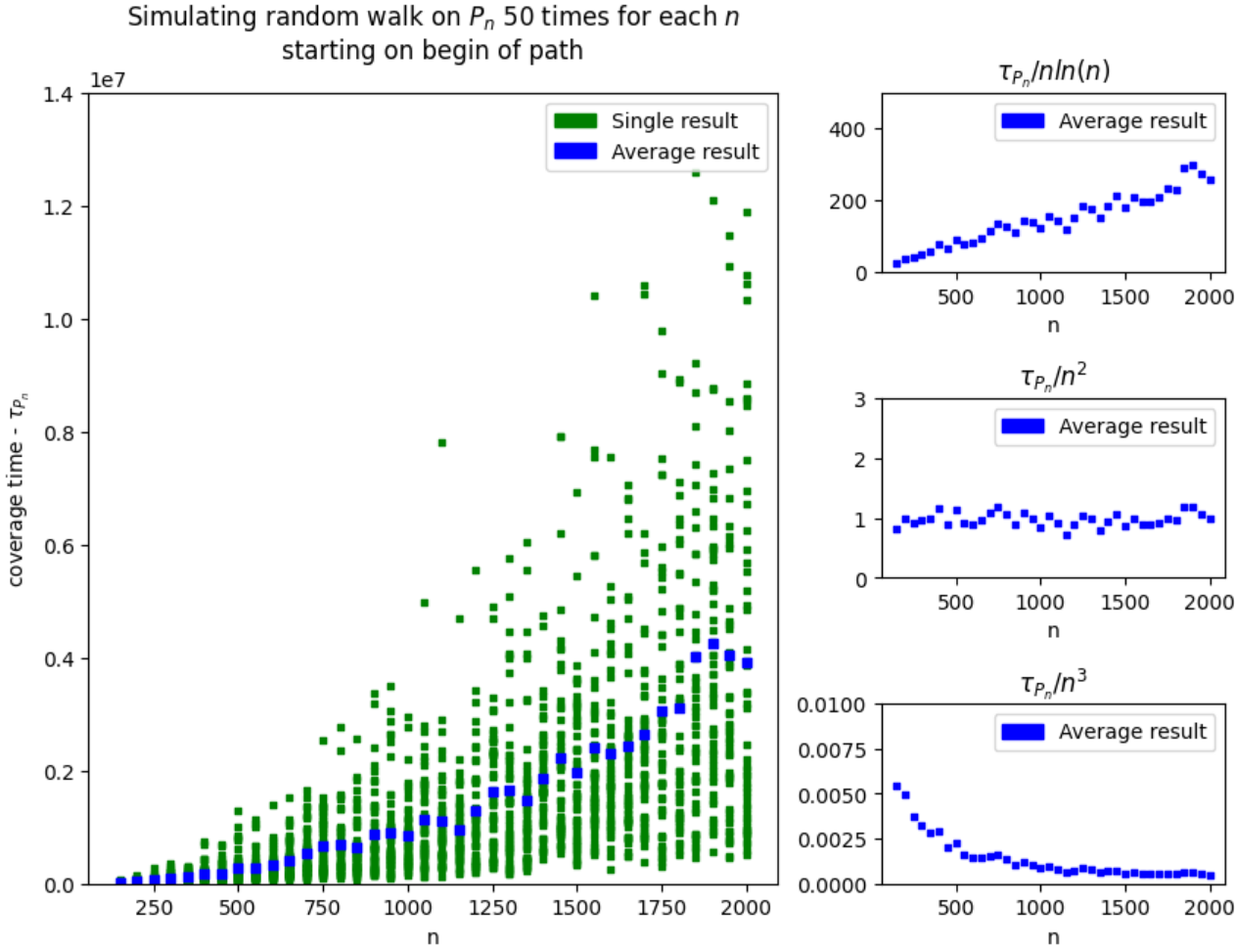
Średni czas pokrycia τ_{K_n} dla procesu błądzenia losowego na klice K_n



Powyższe wykresy przedstawiają wyniki procesu błądzenia losowego na klice K_n. Po lewej stronie znajdują się wyniki poszczególnych(50-ciu niezależnych) prób oraz ich średnie dla każdej z testowanej wielkości grafu(100 - 2000 wierzchołków). Gdy liczba wierzchołków jest niewielka koncentracja wyników wokół średniej jest duża. Wraz ze wzrostem liczby wierzchołków koncetracja szybko maleje. Wraz ze wzrostem liczby wierzchołków rośnie czas potrzebny na odwiedzenie każdego z nich. Po prawej stronie znajdują się wykresu porównujące średni czas pokrycia τ_{K_n} z funkcjami: $f(n) = n$, $f(n) = nln(n)$, $f(n) = n^2$. Wartości ilorazu na wykresie górnym wraz ze wzrostem n rosną zaś na wykresie dolnym zbiegają do zera. Wartość stała utrzymuje się na wykresie środkowym. Na tej podstawie można postawić hipotezę, że:

$$\tau_{K_n} = O(nln(n)).$$

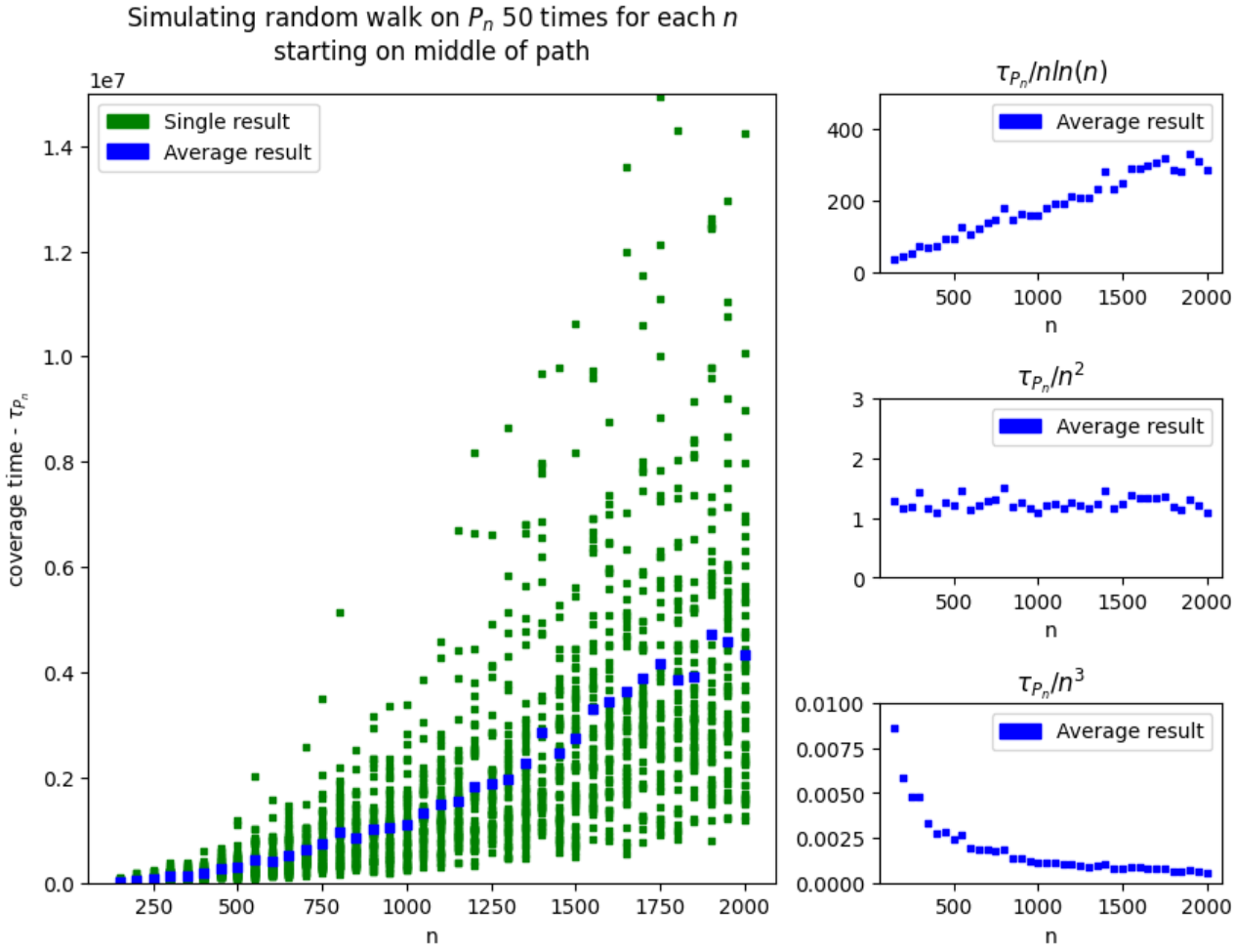
Średni czas pokrycia τ_{P_n} dla procesu błądzenia losowego na ścieżce P_n, gdy proces startuje na początku ścieżki



Powyższe wykresy przedstawiają wyniki procesu błądzenia losowego na ścieżce P_n, gdy proces startuje na jej początku(tj. wierzchołku brzegowym). Po lewej stronie znajdują się wyniki poszczególnych(50-ciu niezależnych) prób oraz ich średnie dla każdej z testowanej wielkości grafu(100 - 2000 wierzchołków). Gdy liczba wierzchołków jest niewielka koncentracja wyników wokół średniej jest bardzo duża. Wraz ze wzrostem liczby wierzchołków koncetracja bardzo szybko maleje. Wraz ze wzrostem liczby wierzchołków szybko rośnie czas potrzebny na odwiedzenie każdego z nich. Po prawej stronie znajdują się wykresu porównujące średni czas pokrycia τ_{P_n} z funkcjami: $f(n) = nln(n)$, $f(n) = n^2$ i $f(n) = n^3$. Wartości ilorazu na wykresie górnym wraz ze wzrostem n rosną zaś na wykresie dolnym zbiegają do zera. Wartość stała utrzymuje się na wykresie środkowym. Na tej podstawie można postawić hipotezę, że:

$$\tau_{P_n} = O(n^2).$$

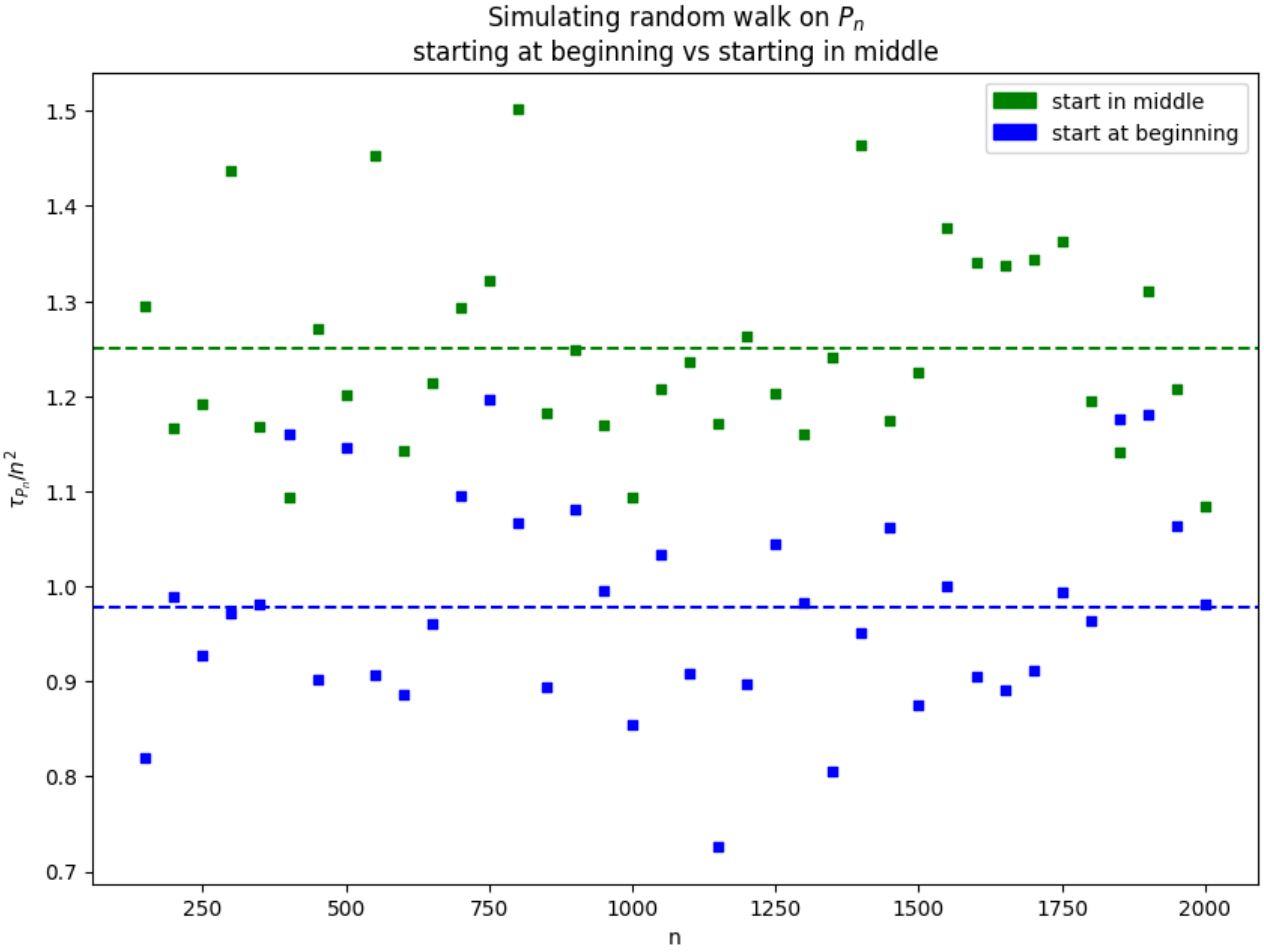
Średni czas pokrycia τ_{P_n} dla procesu błądzenia losowego na ścieżce P_n, gdy proces startuje na środku ścieżki



Powyższe wykresy przedstawiają wyniki procesu błądzenia losowego na ścieżce P_n, gdy proces startuje w jej środku(tj. wierzchołku nr. $\lfloor \frac{n}{2} \rfloor$). Po lewej stronie znajdują się wyniki poszczególnych(50-ciu niezależnych) prób oraz ich średnie dla każdej z testowanej wielkości grafu(100 - 2000 wierzchołków). Gdy liczba wierzchołków jest niewielka koncentracja wyników wokół średniej jest bardzo duża. Wraz ze wzrostem liczby wierzchołków koncetracja bardzo szybko maleje. Wraz ze wzrostem liczby wierzchołków szybko rośnie czas potrzebny na odwiedzenie każdego z nich. Po prawej stronie znajdują się wykresu porównujące średni czas pokrycia τ_{P_n} z funkcjami: $f(n) = nln(n)$, $f(n) = n^2$ i $f(n) = n^3$. Wartości ilorazu na wykresie górnym wraz ze wzrostem n rosną zaś na wykresie dolnym zbiegają do zera. Wartość stała utrzymuje się na wykresie środkowym. Na tej podstawie można postawić hipotezę, że:

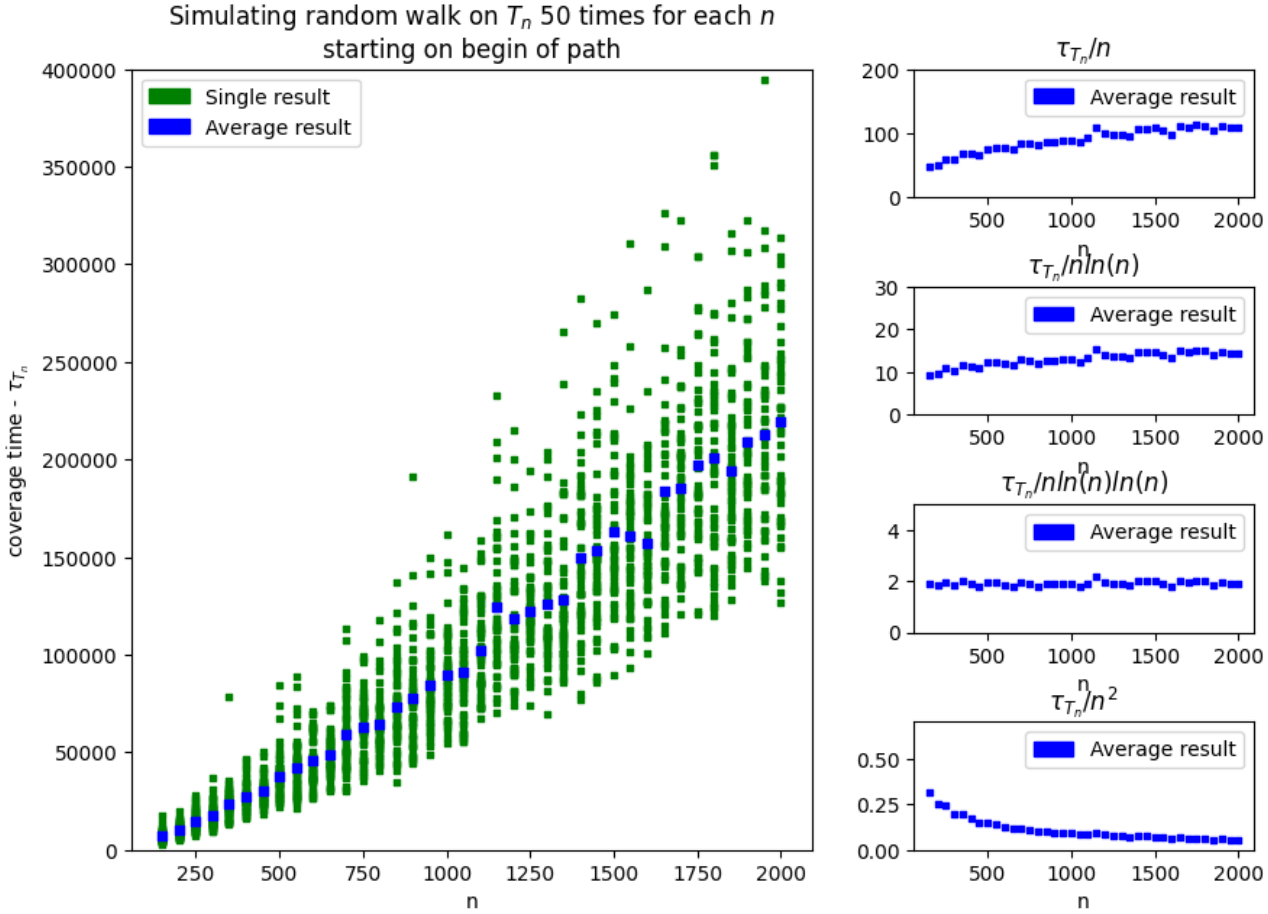
$$\tau_{P_n} = O(n^2).$$

Średni czas pokrycia τ_{P_n} dla procesu błądzenia losowego na ścieżce P_n w zależności od wierzchołka startowego



Powyższy wykres przedstawia porównanie czasów pokrycia dla ścieżek P_n w zależności od wierzchołka początkowego. Średnie czasy dla każdego z n zostały przeskalowane przez n^2 w celu ułatwienia porównania. Na podstawie wykresu można wysnąć wniosek, że czas pokrycia jest dłuższy, gdy spacer startuje na środku ścieżki.

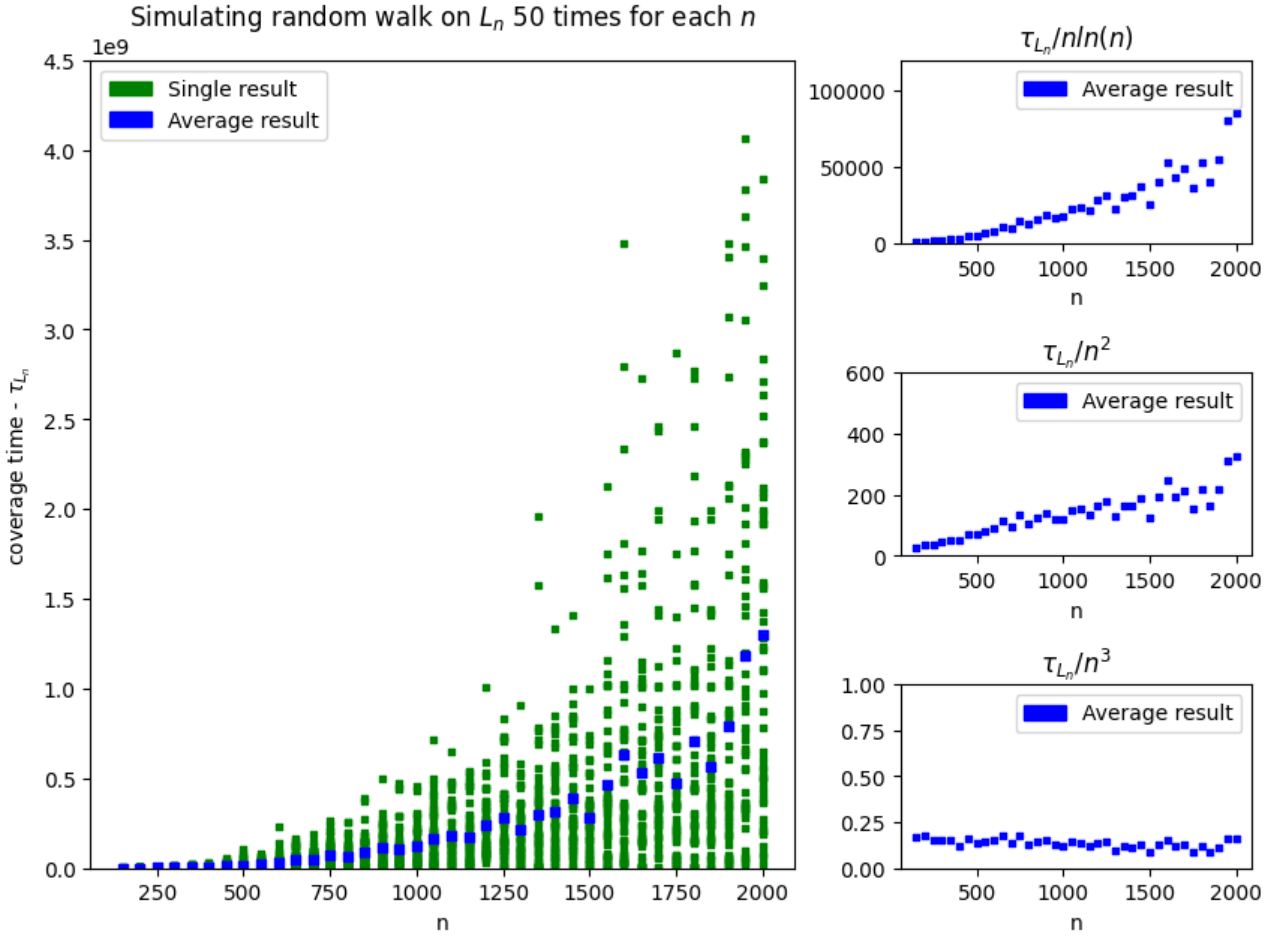
Średni czas pokrycia τ_{T_n} dla procesu błądzenia losowego na pełnym drzewie binarnym T_n, gdy proces startuje w korzeniu drzewa



Powyższe wykresy przedstawiają wyniki procesu błądzenia losowego na pełnym drzewie binarnym T_n. Po lewej stronie znajdują się wyniki poszczególnych(50-ciu niezależnych) prób oraz ich średnie dla każdej z testowanej wielkości grafu(100 - 2000 wierzchołków). Gdy liczba wierzchołków jest niewielka koncentracja wyników wokół średniej jest duża. Wraz ze wzrostem liczby wierzchołków koncetracja maleje. Wraz ze wzrostem liczby wierzchołków rośnie czas potrzebny na odwiedzenie każdego z nich. Po prawej stronie znajdują się wykresu porównujące średni czas pokrycia τ_{T_n} z funkcjami: $f(n) = n$, $f(n) = nln(n)$, $f(n) = nln(n)ln(n)$, $f(n) = n^2$. Wartości ilorazu na dwóch górnych wykresach wraz ze wzrostem n rosną zaś na wykresie dolnym zbiegają do zera. Wartość stała utrzymuje się na wykresie środkowym. Na tej podstawie można postawić hipotezę, że:

$$\tau_{T_n} = O(nln(n)ln(n)).$$

Średni czas pokrycia τ_{L_n} dla procesu błądzenia losowego na lizaku L_n, gdy proces startuje na wierzchołku tworzącym klikę



Powyższe wykresy przedstawiają wyniki procesu błądzenia losowego na lizaku L_n. Po lewej stronie znajdują się wyniki poszczególnych(50-ciu niezależnych) prób oraz ich średnie dla każdej z testowanej wielkości grafu(100 - 2000 wierzchołków). Gdy liczba wierzchołków jest niewielka koncentracja wyników wokół średniej jest duża. Wraz ze wzrostem liczby wierzchołków koncentracja bardzo szybko maleje. Wraz ze wzrostem liczby wierzchołków rośnie czas potrzebny na odwiedzenie każdego z nich. Dodatkowo wraz ze wzrostem liczby wierzchołków przedział w jakim zawarte jest kilka kolejnych średnich zwiększa swoją długość. Po prawej stronie znajdują się wykresu porównujące średni czas pokrycia τ_{L_n} z funkcjami: $f(n) = nln(n)$, $f(n) = n^2$, $f(n) = n^3$. Wartości ilorazu na górnym oraz środkowym wykresie wraz ze wzrostem n szybko rosną. Wartość stała utrzymuje się na wykresie dolnym. Na tej podstawie można postawić hipotezę, że:

$$\tau_{L_n} = O(n^3).$$