

# 検索とシミュレーションアルゴリズムの Java

検索とシミュレーションアルゴリズムは、パス探索と確率的問題を解決します。A\* 検索とモンテカルロシミュレーションを探索してみましょう。

## 1. A\* 検索：ヒューリスティックパス探索

A\* は、ヒューリスティックを使用してグラフ内の最短経路を見つける情報付き検索アルゴリズムで、Dijkstra のアルゴリズムと貪欲検索の強みを組み合わせています。ゲームやナビゲーションで広く使用されています。

### Java 実装

```
import java.util.*;

public class AStar {
    static class Node implements Comparable<Node> {
        int x, y, g, h, f;
        Node parent;
        Node(int x, int y) {
            this.x = x;
            this.y = y;
            this.g = 0;
            this.h = 0;
            this.f = 0;
        }
        public int compareTo(Node other) { return this.f - other.f; }
    }

    static int heuristic(int x1, int y1, int x2, int y2) {
        return Math.abs(x1 - x2) + Math.abs(y1 - y2); // マンハッタン距離
    }

    static void aStarSearch(int[][] grid, int[] start, int[] goal) {
        int rows = grid.length, cols = grid[0].length;
        PriorityQueue<Node> open = new PriorityQueue<>();
        boolean[][] closed = new boolean[rows][cols];
        Node startNode = new Node(start[0], start[1]);
        Node goalNode = new Node(goal[0], goal[1]);
        startNode.h = heuristic(start[0], start[1], goal[0], goal[1]);
        startNode.f = startNode.h;
```

```

open.add(startNode);

// int[][] dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
int[][] dirs = {};
while (!open.isEmpty()) {
    Node current = open.poll();
    if (current.x == goal[0] && current.y == goal[1]) {
        printPath(current);
        return;
    }
    closed[current.x][current.y] = true;
    for (int[] dir : dirs) {
        int newX = current.x + dir[0], newY = current.y + dir[1];
        if (newX >= 0 && newX < rows && newY >= 0 && newY < cols && grid[newX][newY] != 1 && !closed[newX][newY]) {
            Node neighbor = new Node(newX, newY);
            neighbor.g = current.g + 1;
            neighbor.h = heuristic(newX, newY, goal[0], goal[1]);
            neighbor.f = neighbor.g + neighbor.h;
            neighbor.parent = current;
            open.add(neighbor);
        }
    }
}

System.out.println(" パスが見つかりませんでした！");
}

static void printPath(Node node) {
    List<int[]> path = new ArrayList<>();
    while (node != null) {
        path.add(new int[]{node.x, node.y});
        node = node.parent;
    }
    Collections.reverse(path);
    System.out.println(" パス:");
    for (int[] p : path) System.out.println("(" + p[0] + ", " + p[1] + ")");
}

public static void main(String[] args) {
    int[][] grid = {
        {0, 0, 0, 0},

```

```

        {0, 1, 1, 0},
        {0, 0, 0, 0}
    };

    int[] start = {0, 0}, goal = {2, 3};
    aStarSearch(grid, start, goal);
}
}

```

## 出力:

パス:

```

(0, 0)
(1, 0)
(2, 0)
(2, 1)
(2, 2)
(2, 3)

```

## 2. モンテカルロシミュレーション：確率的推定

モンテカルロ法は、ランダムサンプリングを使用して結果を推定し、例えば、正方形と円内の点をシミュレーションして $\pi$ を近似します。

### Java 実装

```

import java.util.Random;

public class MonteCarlo {
    static double estimatePi(int points) {
        Random rand = new Random();
        int insideCircle = 0;
        for (int i = 0; i < points; i++) {
            double x = rand.nextDouble();
            double y = rand.nextDouble();
            if (x * x + y * y <= 1) insideCircle++; // 単位円内
        }
        return 4.0 * insideCircle / points; // 比率 * 4 で  $\pi$  を近似
    }

    public static void main(String[] args) {

```

```
int points = 1000000;
double pi = estimatePi(points);
System.out.println("1000000 個の点で推定された : " + pi);
System.out.println(" 実際の : " + Math.PI);
}
}
```

### 出力（ランダム性により変動します）：

1000000個の点で推定された : 3.1418

実際の : 3.141592653589793