The Voronoi Game on Graphs

Muneeb Azfar Nafees

May 2025

Contents

1 Abstract

In competitive facility placement, companies vie for market share by choosing locations on a network, which we model as the two-player Voronoi game on graphs. In this project, I implemented and compared three strategies, a random selector, a greedy chooser based solely on vertex value, and a neighborhood-aware heuristic that scores each candidate by its own value plus the values of its immediate neighbors, to better understand how local information affects outcomes. I drew on core concepts from graph theory and greedy algorithms, using Java's graph structure to run 1,000 simulations per pairing. The results reveal that pure randomness is highly unreliable, greedy selection beats random picking but remains suboptimal, and the neighborhood-based algorithm consistently outperforms both by leveraging adjacent-vertex information.

2 Results

2.1 Analysis 1: Describe your algorithm

The `VoronoiNeighbourGreedyPlayer` implementation scores each unclaimed vertex v by

$$\delta = 1.1, \quad N_\delta(v) = \{\, u \in N(v) \setminus T \mid d(v,u) \le \delta \},$$

$$\text{score}(v) \;=\; \text{value}(v) \;+\; \sum_{u \in N_\delta(v)} \text{value}(u)$$

Where,

- $N(v)$ is the set of all neighbors of v in the graph.
- T is the set of already-chosen (token) vertices, so $N(v) \setminus T$ filters out taken nodes.
- $d(v,u)$ is the edge distance (random in $[1,2]$). By choosing $\delta = 1.1$, we capture exactly those true-unit-length edges ($d(v,u) = 1$) while ignoring any longer edges.

```
public class VoronoiNeighbourGreedyPlayer extends VoronoiPlayerAlgorithm {
    public VoronoiNeighbourGreedyPlayer(VoronoiGraph g){
        super(g);
    }

    public Vertex chooseVertex(int playerIndex, int numRemainingTurns) {
        Vertex best = null;
        int maxScore = -1;
        for (Vertex v : graph.getVertices()) {
            if (!graph.hasToken(v)) {
                int score = graph.getValue(v);
                for (Vertex u : v.adjacentVertices()) {
                    if (!graph.hasToken(u)
                        && v.getEdgeTo(u).distance() <= 1.1) {
                        score += graph.getValue(u);
                    }
                }
                if (score > maxScore) {
                    maxScore = score;
                    best = v;
                }
            }
        }
        return best;
    }
}
```

2.2 Analysis 2: Thought process

At first I observed that the greedy baseline picks the single highest-value vertex, but neglects the surrounding territory. In the Voronoi game, controlling nearby districts can yield far more total revenue than an isolated peak. Thus I devised a heuristic that still prioritizes high-value vertices, but also accounts for the values of their immediate neighbors (within a small distance threshold). This balances raw vertex worth against local "cluster" potential to maximize overall assigned value.

2.3 Analysis 3: Experimental results

I ran four batches of experiments (20 and 1,000 simulations each) pairing the neighbourhood-greedy algorithm (P1) against the random and greedy baselines:

Opponent	Simulations	P1Wins	P2Wins	Ties	P1 Win Rate
Random	20	20	0	0	100%
Random	1000	995	5	0	99.5%
Greedy	20	18	2	0	90%
Greedy	1000	941	59	0	94.1%

Table 1: Performance of the neighbourhood-greedy heuristic (P1) against random and greedy opponents.

These results show that:

- Against the purely random player, the heuristic wins nearly every time (100%–99.5%).
- Against the value-greedy player, it still wins overwhelmingly (90% in 20 games; 94.1% in 1 000 games).

With this information, I conclude that incorporating immediate-neighbour information substantially improves performance over both baselines.

2.4 Analysis 4: Runtime analysis

Let V be the number of vertices and E the number of edges in the graph. For each move, the algorithm:

1. Iterates over all V vertices.
2. For each vertex v, visits each neighbor in v's adjacency list.

Since the sum of all adjacency-list lengths is $2E$, the total work per move is

$$O\big(V + \sum_{v} |N(v)|\big) \;=\; O(V + 2E) \;=\; O(V + E).$$

Therefore, the overall runtime is $O(V + E)$.

3 Extensions

3.1 Enhanced Neighbour-Greedy Heuristic

3.1.1 Analysis 1: Describe your algorithm

```
public class VoronoiNeighbourGreedyPlayer2 extends VoronoiPlayerAlgorithm {

    public VoronoiNeighbourGreedyPlayer2(VoronoiGraph g){
        super(g);
    }

    public Vertex chooseVertex(int playerIndex, int numRemainingTurns){
        Vertex out = null;
        double maxValue = -1.0;

        for (Vertex v : graph.getVertices()) {
            if (!graph.hasToken(v)) {
                double totalValue = 0.0;
                // Inverse-distance neighbor loop with opponent-aware filtering
                for (Vertex neighbour : v.adjacentVertices()) {
                    if (!graph.hasToken(neighbour)) {
                        double distYou = v.getEdgeTo(neighbour).distance();
                        Vertex oppTok = graph.getClosestToken(neighbour);
                        if (oppTok == null
                            || graph.getDistance(oppTok, neighbour) >= distYou) {
                            totalValue += graph.getValue(neighbour)
                                        / graph.getDistance(v, neighbour);
                        }
                    }
                }
                // Include the value of v itself
                totalValue += graph.getValue(v);

                if (totalValue > maxValue) {
                    maxValue = totalValue;
                    out = v;
                }
            }
        }
        return out;
    }
}
```

This version differs from the original neighbor-greedy in three key respects:

- **Floating-point scoring**: Uses `double` for `totalValue` and `maxValue`, enabling fractional contributions.
- **Inverse-distance weighting**: Each neighbor's value is divided by the exact edge length $d(v, u)$, so closer neighbors contribute more.
- **Opponent-aware filtering**: Before adding a neighbor's contribution, we compare $d(v, u)$ to the distance from the neighbor to the opponent's closest token (via `getClosestToken` and `getDistance`). Only neighbors that remain contestable by us are counted.

3.1.2 Analysis 2: Thought process

In the original neighbor-greedy approach, every adjacent district counted the same, which often overvalued far-away or already-claimed vertices. To fix this, I switched to a floating-point score and divided each neighbor's value by its exact edge length, so that truly close districts carry more weight. Then, before adding any neighbor's contribution, I used `getClosestToken` and `getDistance` to see if the opponent's nearest store would reach that vertex sooner; if so, I skip that neighbor entirely. These two simple changes keep the algorithm's one-hop scan but steer it toward high-value vertices it actually stands a chance of capturing.

3.1.3 Analysis 3: Experimental results

I tested the enhanced neighbor-greedy (P1) against three baselines, random, greedy, and the original neighbor-greedy, over two batch sizes:

Opponent	Sims	P1Wins	P2Wins	Ties	P1 Win Rate
Random	20	20	0	0	100%
Random	1000	1000	0	0	100%
Greedy	20	20	0	0	100%
Greedy	1000	989	11	0	98.9%
Old function	20	17	3	0	85%
Old function	1000	872	127	1	87.2%

Table 2: Performance of the enhanced neighbor-greedy heuristic (P1) against random, greedy, and original neighbor-greedy opponents.

Discussion:

- Against random play, the new heuristic maintains a perfect 100% win rate.
- Versus the value-greedy baseline, it wins 100% of short runs and 98.9% of long runs.

- Against the original neighbor-greedy, it wins 85% (20 sims) and 87.2% (1000 sims) games respectively.

3.1.4 Analysis 4: Runtime analysis

The enhanced algorithm still visits each vertex and its adjacency list exactly once per turn. If V is the number of vertices and E the number of edges, each move costs

$$O\Big(V + \sum_{v \in V} |N(v)|\Big) \;=\; O(V + 2E) \;=\; O(V + E).$$

Thus the overall complexity remains $O(V + E)$, with only constant-factors from distance divisions and owner checks.

3.2 Monte Carlo Player Algorithm - Contest Winner!

3.2.1 Analysis 1: Describe your algorithm

```
public class VoronoiMonteCarloPlayer extends VoronoiPlayerAlgorithm {
    private static final int TOTAL_TURNS = 5;
    private static final int SIMULATIONS_PER_CANDIDATE = 25;
    private final Random rand = new Random();

    public VoronoiMonteCarloPlayer(VoronoiGraph g) {
        super(g);
    }

    @Override
    public Vertex chooseVertex(int playerIndex, int numRemainingTurns) {
        Set<Vertex> myTokens = new HashSet<>();
        Set<Vertex> oppTokens = new HashSet<>();
        for (Vertex v : graph.getVertices()) {
            if (graph.hasToken(v)) {
                Integer owner = graph.getCurrentOwner(v);
                if (owner == playerIndex) {
                    myTokens.add(graph.getClosestToken(v));
                } else {
                    oppTokens.add(graph.getClosestToken(v));
                }
            }
        }

        List<Vertex> allVerts  = new ArrayList<>(graph.getVertices());
        List<Vertex> available = allVerts.stream()
```

```
        .filter(v -> !graph.hasToken(v))
        .collect(Collectors.toList());

    int movesDone = 2 * (TOTAL_TURNS - numRemainingTurns);
    int movesLeft = 2 * TOTAL_TURNS - movesDone;

    Vertex best = null;
    double bestScore = Double.NEGATIVE_INFINITY;

    for (Vertex candidate : available) {
        double sumDelta = 0;
        for (int sim = 0; sim < SIMULATIONS_PER_CANDIDATE; sim++) {
            Set<Vertex> simMy  = new HashSet<>(myTokens);
            Set<Vertex> simOpp = new HashSet<>(oppTokens);
            simMy.add(candidate);

            List<Vertex> pool = new ArrayList<>(available);
            pool.remove(candidate);
            Collections.shuffle(pool, rand);

            boolean myTurn = false;
            for (int i = 0; i < movesLeft - 1; i++) {
                Vertex pick = pool.get(i);
                if (myTurn)  simMy.add(pick);
                else         simOpp.add(pick);
                myTurn = !myTurn;
            }

            int myScore  = 0, oppScore = 0;
            for (Vertex u : allVerts) {
                double bestMy  = simMy.stream()
                    .mapToDouble(t -> graph.getDistance(u, t))
                    .min().orElse(Double.POSITIVE_INFINITY);
                double bestOpp = simOpp.stream()
                    .mapToDouble(t -> graph.getDistance(u, t))
                    .min().orElse(Double.POSITIVE_INFINITY);

                if (bestMy < bestOpp)  myScore += graph.getValue(u);
                else                   oppScore += graph.getValue(u);
            }
            sumDelta += (myScore - oppScore);
        }
```

```
            double avgDelta = sumDelta / SIMULATIONS_PER_CANDIDATE;
            if (avgDelta > bestScore) {
                bestScore = avgDelta;
                best      = candidate;
            }
        }
        return best;
    }
}
```

This Monte Carlo player works by, for each unclaimed vertex, running a fixed number of random "playouts" to the end of the game. In each simulation both players pick remaining vertices at random, and we compute the resulting score difference under the Voronoi assignment. By averaging these score differences, the algorithm estimates which initial move is most likely to yield the highest net gain. The vertex with the highest average advantage is selected. This approach lets the player look ahead across possible futures without hard-coding any specific greedy heuristic.

3.2.2 Analysis 2: Thought process

I wanted a strategy that looks ahead by simulating random completions of the game from each candidate pick. For each unclaimed vertex, I run a fixed number of Monte Carlo playouts where both players pick randomly for the remaining turns. After each simulation, I compute the Voronoi assignment and record the score difference. Averaging these differences gives an estimate of how strong each initial move is. This lets the algorithm favor moves likely to produce high revenue under random play, without hard-coding any greedy rule.

3.2.3 Analysis 3: Experimental results

Opponent	Sims	P1Wins	P2Wins	Ties	P1 Win Rate
Random	20	20	0	0	100%
Random	100	100	0	0	100%
Greedy	20	20	0	0	100%
Greedy	100	100	0	0	100%
Original Neighbour-Greedy	20	20	0	0	100%
Original Neighbour-Greedy	100	100	0	0	100%
Enhanced Neighbour-Greedy	20	17	3	0	85%
Enhanced Neighbour-Greedy	100	84	16	0	84%

Table 3: Performance of the Monte Carlo heuristic (P1) against various baselines.

Against both the random and greedy opponents, Monte Carlo wins every game in both short and long runs. It also defeats the original neighbour-greedy player 100% of the time. When facing the enhanced neighbour-greedy, it still wins 85% of 20 games and 84% of 100 games, showing strong robustness across all matchups.

3.2.4 Analysis 4: Runtime analysis

Let V be the number of vertices. Each turn, the Monte Carlo algorithm:

1. Considers all V candidate vertices.
2. Runs a constant S simulations per candidate.
3. In each simulation, performs $O(V)$ work to compute the Voronoi assignment.

Thus the per-move time is $O(S \cdot V \cdot V) = O(V^2)$ (since S is fixed). This is significantly slower than the $O(V + E)$ neighbor-greedy methods, but remains practical for $V = 100$.

4 Acknowledgments

1. Wikipedia: Nearest neighbour algorithm
2. Wikipedia: Greedy algorithm
3. Voronoi game on graphs