Games on Graphs + Program Decisions

GitHub Repository: https://github.com/LevkoBe/GT_Energy_Wars

Live Demo: https://levkobe.github.io/GT_Energy_Wars/

> This project explores the application of graph theory through a web-based strategy game I developed, *GT: Energy Wars*. Rather than analyzing an existing game on graphs, I chose to build my own from scratch to better integrate specific algorithms and structures from the course. The result is a playable demonstration of concepts such as minimum spanning trees, network connectivity, simplified flow models, and social influence propagation, framed in a scenario where energy companies compete to supply a city via graph-based infrastructure.

▼ Multi-Layered Graph Representation

The game models a city using two graphs:

- **Power Graph**: A directed graph with capacity constraints, representing electricity flow.
- **Social Graph**: An undirected graph reflecting influence between people in the city.

Each node (house) carries metadata like energy demand, owner, position, and social links. This setup supports simulations of satisfaction, sabotage, and energy flow.

▼ Key Graph Algorithms

Kruskal's Algorithm for Initialization

Each player's starting network is built using Kruskal's MST algorithm. This ensures their initial houses are efficiently connected without manual setup:

```
export const kruskalsMST = (nodes: Node[], nodeIds: number[]): PowerEdge
[] => {
  const edges = nodeIds.flatMap((a, i) =>
    nodeIds.slice(i + 1).map((b) => ({
      from: a, to: b,
      weight: distance(nodes[a], nodes[b])
    }))
  );

  edges.sort((a, b) => a.weight - b.weight);
  const uf = new UnionFind(nodeIds);
  const mst: PowerEdge[] = [];

  for (const { from, to } of edges) {
    if (uf.union(from, to)) {
      mst.push(createPowerEdge(from, to));
    }
  }
  return mst;
};
```

Complexity: O(E log E) for sorting edges, optimal for the application's graph sizes.

Union-Find for Cycle Detection

```
find(x: number): number {
  const p = this.parent.get(x);
  if (p !== x) this.parent.set(x, this.find(p!));
  return this.parent.get(x)!;
}

union(x: number, y: number): boolean {
```

```
  const rootX = this.find(x);
  const rootY = this.find(y);
  if (rootX === rootY) return false;

  const rankX = this.rank.get(rootX)!;
  const rankY = this.rank.get(rootY)!;

  if (rankX < rankY) this.parent.set(rootX, rootY);
  else if (rankX > rankY) this.parent.set(rootY, rootX);
  else {
    this.parent.set(rootY, rootX);
    this.rank.set(rootX, rankX + 1);
  }
  return true;
}
```

Complexity: Nearly O(1) time per operation with optimizations.

BFS for Connectivity

```
export const BFSfindPath = (from: number, to: number, edges: PowerEdge[],
player: number): boolean ⇒ {
  const graph: Record<number, number[]> = {};
  edges.filter(e ⇒ e.owner === player).forEach(e ⇒ {
    (graph[e.from] ??= []).push(e.to);
    (graph[e.to] ??= []).push(e.from);
  });

  const visited = new Set([from]);
  const queue = [from];

  while (queue.length) {
    const current = queue.shift()!;
    if (current === to) return true;
    for (const neighbor of graph[current] ?? []) {
      if (!visited.has(neighbor)) {
```

```
        visited.add(neighbor);
        queue.push(neighbor);
      }
    }
  }
  return false;
};
```

Complexity: O(V + E) for each connectivity query.

▼ Overview: Algorithm Comparison

Problem	Chosen Algorithm	Complexity	Alternatives Considered	Justification
Network initialization	Kruskal's MST	O(E log E)	Prim's MST	Simple, efficient for sparse graphs
Cycle detection in MST	Union-Find (with path compression & union by rank)	≈ O(1) per op	DFS with visited sets	Needed for Kruskal's efficiency
Connectivity/path check	BFS	O(V + E)	DFS, Dijkstra	Finds shortest hop path; fast and simple
Flow allocation	Greedy propagation	≈ O(path length)	Ford-Fulkerson, Edmonds-Karp	Prioritizes real-time performance
Sabotage influence propagation	Local voting + neighbor scan	O(degree)	Global influence model, contagion sim	Keeps system reactive and avoids costly simulations

▼ Modeling Simplified Power Flow

I simplified network flow to fit gameplay constraints. Rather than using Ford-Fulkerson, I opted for a greedy allocation that honors edge capacities but runs fast enough for interactive play. While this sacrifices optimality, it keeps the game responsive.

Each turn recalculates flows and updates house satisfaction. Nodes with unmet energy needs shift from "positive" to "neutral" or "negative" states, influencing their loyalty and possibly triggering sabotage.

House Behavior and Sabotage

Houses vote to sabotage their owner's network if consistently underserved. This decision is influenced by their neighbors' attitudes in the social graph — creating an interesting local dynamics and reinforcing the importance of network structure.

```
evaluateSabotageVotes(node: Node): number {
  let vote = node.attitude === "negative" ? 1 : (node.attitude === "positive" ?
-1 : 0);

  node.socialConnections.forEach((neighborId) => {
    const neighbor = this.nodes[neighborId];
    const sameOwner = neighbor.owner === node.owner;

    if (neighbor.attitude === "positive") vote += sameOwner ? -1 : +1;
    if (neighbor.attitude === "negative") vote += sameOwner ? +1 : -1;
  });

  return vote;
}
```

This mechanic mimics real-world instability caused by social contagion and unmet needs, grounded in graph influence propagation.

Performance and Practical Trade-offs

Every design choice balanced algorithmic correctness with real-time constraints. For example:

- Greedy flow over max-flow
- Local sabotage checks over global failure modeling
- BFS over more advanced pathfinding algorithms

These decisions were primarily focused to provide user with a fast-working interactive game, and allow me as a developer to focus on delivery of the application.

▼ Future Work

Potential directions include:

- Replacing greedy flow with proper min-cost max-flow
- Modeling multiple resource types (multi-commodity flow)
- Adding AI players via reinforcement learning
- Making the sabotage system more nuanced via probabilistic models
- Allowing players to construct more than one power tower

▼ Conclusion

Building *GT: Energy Wars* helped me experience graph theory beyond equations and pseudocode. It showed me how classical algorithms hold up — and evolve — when embedded in real-time systems with constraints and trade-offs.

The full source code is available here: GitHub Repository

Play the game here: Live Demo