Gameboxd

Montador de Mesas para Jogos de Tabuleiro com Grafos

Andressa Alves Martins
Giovany Oliveira de Moraes
Luana Ferreira Vasconcelos

Instituto de Informática
Universidade Federal de Goiás

2025

INSTITUTO DE INFORMÁTICA
UFG

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

Bancos de Dados

Bancos de Dados

1. Board Games Recommendation: Graph Drawing Contest 2023

Contém as fichas de 100 jogos contendo as seguintes informações de cada um:

- Id
- Título
- Ano de lançamento
- Rank
- Números mínimo e máximo de jogadores
- Tempo mínimo e máximo de jogo
- Idade mínima
- Rating
- Número de Reviews
- Recomendações (fãs também gostaram)
- Categorias
- Mecânicas
- Créditos

Bancos de Dados

2. Banco original

Contém as fichas de 600 usuários contendo as seguintes informações de cada um:

- Id
- Nome
- Idade
- Categorias favoritas
- Mecânicas favoritas
- Jogos favoritos (com avaliação de 1 a 5)

Os valores possíveis que cada categoria pode assumir foram designados manualmente (como limites de idade inferior e superior).
Porém, os valores propriamente ditos foram determinados randomicamente.

Problema

Problema

- Como filtrar, em uma base de usuários, quem gosta do jogo X?
- Como conectar essas pessoas de forma que a "tensão social" seja mínima (máxima afinidade)?

Solução Proposta

Conceito
Uma aplicação desktop (C++ e Qt) inspirada no visual do Letterboxd (catálogo e avaliações), mas com foco social.

Funcionalidade Principal
O usuário escolhe um jogo e o sistema sugere a "Mesa Perfeita".

Diferencial
Não é apenas "quem joga".
É "quem joga bem junto" baseado em histórico de notas e perfil.

Conceitos utilizados

Grafo ponderado:

Cada aresta possui um peso (custo, distância, tempo etc.).

Usado para modelar problemas onde conexões têm valores diferentes.

Busca em Largura (BFS):

Explora o grafo nível por nível, a partir de um nó inicial.

Garante encontrar o caminho com menor número de arestas.

Algoritmo de Prim

Árvore Geradora Máxima (MST Max):

Subconjunto de arestas que conecta todos os vértices sem ciclos.

Maximiza a soma dos pesos das arestas escolhidas.

Modelagem

Vértices (Nós): Usuários

Atributos: Idade, Categorias Favoritas, Histórico de Avaliações (Ratings).

Arestas (Conexões): Relacionamento de Afinidade

- Grafo Não-Direcionado.
- Grafo Ponderado (Peso = Nível de Compatibilidade).

Topologia: Malha

Inicialmente, calculamos a afinidade de todos contra todos para estabelecer a rede social latente.

Matemática da afinidade

Objetivo: Transformar dados subjetivos em um peso numérico (int weight).

- Fórmula da Aresta
- Mínimo de 25 de afinidade

```
// 1. Afinidade por Jogos e Ratings (peso 40)
QList<int> jogosA = a.jogosAvaliados.keys();
double scoreTemp = 0;
int jogosComum = 0;
for (int gId : jogosA) {
    if (b.jogosAvaliados.contains(gId)) {
        jogosComum++;
        double notaA = a.getAvaliacao(gId);
        double notaB = b.getAvaliacao(gId);

        scoreTemp += (5 - qAbs(notaA - notaB));
    }
}
if (jogosComum >0) scoreTemp /= jogosComum;
if (jogosComum > 10) jogosComum = 10;
score += (scoreTemp * jogosComum) * 0.8;
```

```
// 2. Afinidade por Categorias (peso 20)
QSet<int> intersecaoC = a.categoriasFavoritas;
intersecaoC.intersect(b.categoriasFavoritas);
score += intersecaoC.size() * 20;

// 2. Afinidade por Mecanicas (peso 20)
QSet<int> intersecaoM = a.mecanicasFavoritas;
intersecaoM.intersect(b.mecanicasFavoritas);
score += intersecaoM.size() * 20;

// 3. Afinidade por Idade (peso 20)
int diff = qAbs(a.getIdade() - b.getIdade());
if (diff > 60) diff = 60;
score += 20 - (diff/3);

if (score == 100) score = 99;
```

Algoritmo de Busca (Filtro)

Problema -> O grafo total tem 600 usuários. Para jogar Gloomhaven, não precisamos analisar todos.

Solução -> Busca/Filtro Linear.

Critério -> Selecionar apenas vértices onde user.getAvaliacao(jogoId) >= 3.

Resultado -> Redução do espaço de busca para um subgrafo de candidatos (ex: 10 pessoas).

Algoritmo de Busca (Filtro)

```
QVector<Usuario> Grafo::buscarCandidatos(int gameId) {
    QVector<Usuario> candidatos;
    // Itera sobre todos os usuários do banco
    for (auto user : bancoUsuarios) {
        // Só queremos pessoas que avaliaram o jogo com 3 estrelas ou mais.
        if (user.getAvaliacao(gameId) >= 3) {
            candidatos.push_back(user);
        }
    }

    return candidatos;
}
```

Algoritmo de Prim - MST

Problema -> Temos 10 candidatos compatíveis com o jogo. Quem convidamos para preencher 4 vagas? E quem deve convidar quem?

Solução -> Adaptação do Algoritmo de Prim (Árvore Geradora Mínima/Máxima).

Aplicação -> O Host (usuário logado) é a raiz e o algoritmo busca a aresta de maior peso (maior afinidade) saindo do grupo atual para os não-visitados. Isso evita ciclos e redundâncias.

Resultado Visual -> Uma "Cadeia de Convite" otimizada:

Ana (Host) convida Bruno (90%) -> Bruno convida Carlos (85%)...

Algoritmo de Prim - MST

```
// Adaptado para buscar a MAIOR afinidade (Maximum Spanning Tree)
QString Grafo::gerarMST(const QVector<Usuario>& grupo) {
    if (grupo.size() < 2) return "Grupo muito pequeno para conexões.";

    int n = grupo.size();
    QString resultado;

    // Estruturas auxiliares do Prim
    QVector<bool> visitado(n, false);
    QVector<int> maxPeso(n, -1);       // Peso máximo para chegar ao nó
    QVector<int> pai(n, -1);           // Quem conectou nesse nó

    // Começa pelo primeiro usuário do grupo (índice 0)
    maxPeso[0] = 999999; // Infinito (para ser escolhido primeiro)

    for (int i = 0; i < n; ++i) {

        // 1. Encontrar o vértice não visitado com maior peso de conexão
        int u = -1;
        int maxVal = -1;

        for (int v = 0; v < n; ++v) {
            if (!visitado[v] && maxPeso[v] > maxVal) {
                maxVal = maxPeso[v];
                u = v;
            }
        }

        if (u == -1) break; // Não há mais conexões possíveis (grupo desconexo)

        visitado[u] = true;
```

```
        // Se tem pai, adiciona ao texto (formato da aresta)
        if (pai[u] != -1) {
            int pesoReal = calcularAfinidade(grupo[pai[u]], grupo[u]);
            if (pesoReal >= 25) {
                resultado += QString("%1 --(%2)--> %3\n")
                                .arg(grupo[pai[u]].getNome())
                                .arg(pesoReal)
                                .arg(grupo[u].getNome());
            }
        }

        // 2. Atualizar os vizinhos do vértice escolhido 'u'
        for (int v = 0; v < n; ++v) {
            if (!visitado[v]) {
                // Calcula afinidade na hora (Subgrafo implícito)
                int peso = calcularAfinidade(grupo[u], grupo[v]);

                // Se essa conexão é melhor do que a que eu tinha antes, atualiza
                if (peso > maxPeso[v]) {
                    maxPeso[v] = peso;
                    pai[v] = u;
                }
            }
        }
    }

    return resultado;
}
```

Arquitetura e implementação

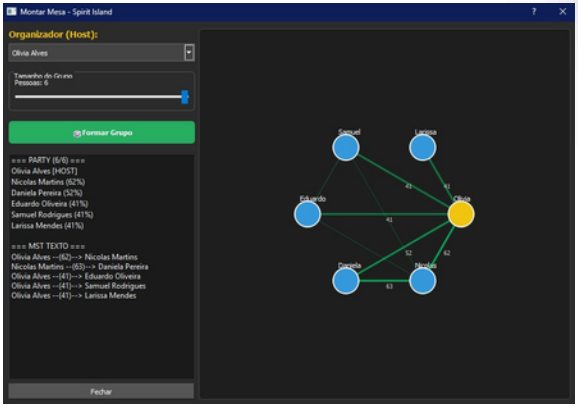
- Linguagem: C++ (Padrão C++11).
- Framework Gráfico: Qt 5 (Widgets + Graphics View).
- Persistência: JSON (Parsing manual com QJsonDocument). - grafo.json e users.json
- Versionamento: github

Estrutura de Classes (TADs):

- Usuario.h/.cpp: Gestão de perfil e avaliações.
- Jogo.h/.cpp: Metadados do jogo.
- Grafo.h/.cpp: Lógica de Matriz de Adjacência, Busca e Prim.
- MainWindow.h/.cpp: Camada de Apresentação.

Visualização do grafo

- Funcionalidade Extra: Implementamos um visualizador gráfico usando QGraphicsScene.
- Visual:
  - Layout circular dos nós.
  - Arestas desenhadas dinamicamente.
  - Espessura da linha proporcional ao peso calculado.



Resultado