Cyberpunk 2077 Skill Tree Planner

Estructuras de Datos y Algoritmos

Geor Sebastián Gómez Correa
Tomás Camilo García López
Eduardo Castellanos Márquez

UNIVERSIDAD NACIONAL DE COLOMBIA

Arquitectura Técnica del Skill Tree

El sistema no es simplemente una colección de imágenes; es una implementación completa de una estructura de datos de **Grafo Dirigido Acíclico (DAG)**, gestionada mediante una máquina de estados finitos para cada nodo.

---

1. Estructura de Datos del Grafo (GLOBAL_STATE)

La columna vertebral de la aplicación es el objeto GLOBAL_STATE. Este objeto es un **Grafo Representado por Listas de Adyacencia**, cargado inicialmente desde skills_data.js (generado por Python) y clonado en tiempo de ejecución para permitir la mutabilidad.

Jerarquía del Objeto

La estructura sigue este esquema anidado:

Árbol (Category) -> Nodo (Perk ID) -> Propiedades

```javascript
const GLOBAL_STATE = {
    "Reflexes": {
        "21": { // ID del Nodo
            id: "21",
            title: "Dash",
            state: "state-blocked", // Estado actual (blocked, available, selected)
            currentLevel: 0,        // Puntos invertidos actualmente
```

```
            maxLevel: 2,            // Puntos máximos
permitidos
            levelReq: 9,            // Nivel de atributo
requerido (Tiers: 4, 9, 15, 20)

            // --- CONEXIONES DEL GRAFO ---
            parents: ["8"],         // Lista de IDs de nodos
prerrequisito (Arcos entrantes)
            children: ["34", "15"], // Lista de IDs de nodos
dependientes (Arcos salientes)

            // --- METADATA VISUAL ---
            icon: "assets/Reflexes/Dash.webp",
            descriptions: ["...", "..."]
        },
        // ... más nodos
    },
    "Body": { ... }
};
```

¿Por qué esta estructura?

1. **Acceso O(1):** Podemos acceder a cualquier nodo instantáneamente usando su ID (treeData["21"]) sin tener que iterar arrays.
2. **Bidireccionalidad:** Al almacenar explícitamente tanto parents como children, podemos recorrer el grafo en ambas direcciones:
    - **Hacia abajo (Children):** Para desbloquear nuevas habilidades cuando compramos una.
    - **Hacia arriba (Parents):** Para validar requisitos antes de permitir una compra.

---

2. Algoritmos de Lógica de Juego

La interacción del usuario dispara algoritmos de validación y propagación de estados a través del grafo.

A. Algoritmo de Validación de Dependencias (AND Logic)

A diferencia de un árbol simple, en este grafo un nodo puede tener múltiples padres. La lógica implementada es una compuerta **AND**.

- **Función:** checkUnlock(nodeId)

- **Disparador:** Se ejecuta sobre los *hijos* de un nodo que acaba de completarse (alcanzar maxLevel).
- **Lógica:**

```
JavaScript

// Pseudocódigo
PARA CADA padre EN node.parents:
    SI (padre.state != SELECTED) O (padre.currentLevel < padre.maxLevel):
        RETORNAR Falso (Bloqueado)
RETORNAR Verdadero (Desbloquear -> state-available)
```

- *Esto asegura que si una habilidad requiere dos ramas previas, ambas deben estar completas.*

B. Algoritmo de Protección de Integridad (Anti-Refund)

Para evitar estados inválidos (tener una habilidad de nivel 20 activa sin tener la de nivel 15), implementamos una verificación recursiva inversa.

- **Función:** handleRightClick() (Validación previa).
- **Lógica:** Antes de devolver un punto de un nodo:
    1. Verificamos si el nodo está actuando como "puente" (currentLevel == maxLevel).
    2. Si es así, escaneamos sus children.
    3. Si algún hijo tiene currentLevel > 0, se bloquea la acción y se lanza una alerta.
       Esto mantiene la coherencia topológica del grafo.

---

3. Algoritmos Visuales y de Renderizado

Aquí es donde las matemáticas se encuentran con el diseño para lograr la estética "Cyberpunk".

A. Trazado de Conexiones ("Manhattan Chamfer")

En lugar de usar líneas Bézier (curvas suaves) o líneas rectas directas (Euclidianas), creamos un algoritmo personalizado para dibujar cables estilo circuito impreso (PCB).

- **Función:** getCircuitPath(x1, y1, x2, y2)

- **Objetivo:** Conectar el borde inferior del padre con el borde superior del hijo evitando cruces diagonales sobre otros nodos.
- **Pasos del Algoritmo:**
    1. **Punto Medio ($midY$):** Calculamos la mitad vertical entre los dos nodos: $y_1 + \frac{y_2 - y_1}{2}$.
    2. **Cálculo de Chaflán ($corner$):** Definimos un recorte de 20px para las esquinas.
    3. **Construcción del Path SVG:**
        - L Vertical hacia abajo hasta $midY - corner$.
        - L Diagonal de 45° ($x + corner, y + corner$).
        - L Horizontal hasta alinearse con el hijo.
        - L Diagonal de 45° inversa.
        - L Vertical final hacia el hijo.

Este algoritmo garantiza simetría y limpieza visual, creando "autopistas de datos" horizontales entre las filas de niveles.

B. Renderizado en Capas (Layering)

Para lograr el efecto de "Cable Plano", no dibujamos una sola línea. Cada conexión genera un grupo SVG (<g>) con 3 trazos superpuestos calculados sobre la misma ruta matemática:

1. **Capa Base (Background):** Línea gruesa (10px). Color del estado (Azul/Amarillo). Opacidad baja.
2. **Capa Separadora (Mask):** Línea media (6px). Color negro (#050505). Simula la separación física de dos cables.
3. **Capa Núcleo (Core):** Línea fina (2px). Color brillante sólido. Si está activa, recibe una animación CSS stroke-dasharray para simular flujo de datos.

---

4. Persistencia y Serialización

El sistema maneja tres tipos de persistencia de datos:

1. **Estática (Python -> JSON):**
    - El script process_data.py realiza la **Inferencia de Padres**. Dado que el Excel original solo tenía "Next Perk ID" (Hijos), el script recorre los nodos y puebla inversamente la lista de parents. Esto transforma una lista simple en un grafo navegable bidireccionalmente antes de llegar al navegador.
2. **De Sesión (Variable Global):**
    - GLOBAL_STATE mantiene el estado mientras el usuario cambia de pestañas (Body <-> Reflexes). Al cambiar de pestaña, no se recarga el archivo, sino que se repinta el DOM basándose en este objeto en memoria.
3. **Local/Exportable (Presets System):**
    - Para guardar y cargar builds, usamos un algoritmo de **Serialización Diferencial**.

- **Guardar:** No guardamos todo el árbol. Solo guardamos un mapa de { ID: Nivel } de los nodos que tienen currentLevel > 0. Esto hace que los strings de guardado sean muy ligeros.
- **Cargar:**
    1. resetInternalState() (Limpia todo a 0).
    2. Inyecta los niveles del preset.
    3. **Recálculo Masivo (recalculateAllStates):** Como inyectamos niveles arbitrariamente, el grafo puede quedar inconsistente visualmente. Ejecutamos un barrido de 3 pasadas sobre todo el grafo para actualizar automáticamente qué nodos deben aparecer como "Disponibles" (Azules) basándose en los nuevos niveles de sus padres.