木幅アルゴリズムの学習システムの構築

正答判定結果保持とグラフ画像生成の修正

B4 小林紹子

November 14, 2025

目次

前回までの課題

1. ◎ × 問題は正答判定可能だが, 数値の入力問題でも正答判定が不可能.
2. それぞれの問題の正解・不正解が保存されない.
3. React Flow で思ったようにグラフが描けない.
4. 問題文と画像生成が連携されていない.
5. 問題がページ分割されずに一覧で出てくる.

目次

Problem テーブルの修正

修正前:

```
1       model Problem{
2       id Int @id @default(
            autoincrement())
3       text String
4       image String?
5       answer Boolean
6       }
```

変更後：

```
7       model Problem{
8       id Int @id @default(
            autoincrement())
9       text String
10      image String?
11      answer Int
12      }
```

→answer の型を Boolean 型から Int 型に変更.

⇒ まだ問題ページから入力する欄は作れていない.

⇒ ◎ × 問題か入力問題かカテゴリ分けする必要あり.

目次

前回までの課題点

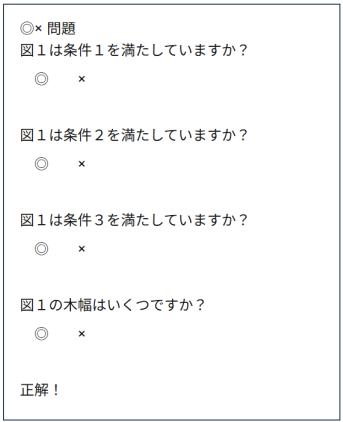


Figure: 修正前

- 正答判定結果が問題一覧の最後にある.
- 他の問題を解くと正答判定結果が変化.
- 問いた問題１問１問の結果が保存できていない.

$\Longrightarrow$ 問題ごとに判定結果表示欄を作り、正答判定を保存できるようにする.

1問ずつ正答判定欄を作成（problems/page.tsx）

修正前：

```
1     {problems.map((p) => (
2       <div key={p.id} style={{marginBottom: 30}}>
3         <p>問題{p.id}</p>
4         <p>{p.text}</p>
5         {p.image && <img src={p.image} alt="問題画像"
                width={200}></img>}
6         <div>
7           <button ...>×</button>
8         </div>
9   </div>
10        )
11  )}
12    <div>{messages[p.id] ? "正解！ " : "不正解"}</div>
```

修正後:

```
1     {problems.map((p) => (
2       <div key={p.id} style={{marginBottom: 30}}>
3         <p>問題{p.id}</p>
4         <p>{p.text}</p>
5         ...
6         <div>
7           <button ...>×</button>
8         </div>
9         {/*mapの中に正答判定結果欄を作成*/}
10        <h3>{messages[p.id] ? "正解！ " : "不正解"}</
              h3>
11      </div>
12    )
13  )}
```

$\Longrightarrow$ 正答判定結果欄を map の中に入れる.

$\Longrightarrow$ しかし, 正答判定結果が問題ごとに保存されていない.

問題別の正答結果を維持させる（problems/page.tsx）I

1. React Hook と呼ばれる状態管理機能（useState）を用いる [1].
2. それぞれの問題の正解を保持できる map（messages）と更新関数（setMessages）を作成.
3. サーバーから用いた正答判定を messages に保存させる.

```
1     export default function ProblemsPage(){
2   //問題の配列を状態管理
3   const [problems, setProblems] = useState<Problem[]>([]);
4   //正答判定をProblem.idをキーとするmapでbooleanまたはundefinedを管理
5   const [messages,setMessages] = useState<{[id:number]:boolean |
        undefined}>([]);
6
7   //GETリクエストで問題一覧を取得.
8   useEffect(() => {
9     fetch("/api/problems")
```

問題別の正答結果を維持させる（problems/page.tsx）II

```
10      //fetchはレスポンスオブジェクトを返すのでJSON型に変換.
11      .then((res) => res.json())
12      .then((data) => setProblems(data));
13    },[]);
14    //回答を送信し、正答判定を取得
15    async function handleAnswer(id: number,answer: number) {
16      const res = await fetch("/api/problems",{
17        method: "POST",
18        headers: {"Content-Type": "application/json"},
19        body: JSON.stringify({id,answer}),
20      });
21      const data = await res.json();
22      setMessages((prev) => ({
23        //全プロパティをコピー（既存の回答状況をコピー）
24        ...prev,
25        //新たに正答判定を追加
26        [id]:data.correct,
```

問題別の正答結果を維持させる（problems/page.tsx）III

```
27        }))
28      }
29      return(
30        ...
31        {/*回答判定されている場合のみ判定結果を出力 */}
32            {messages[p.id] !== undefined && (
33            <h3>{messages[p.id] ? "正解！ " : "不正解"}</h3>
34            )}
35      )
36    }
```

実行結果

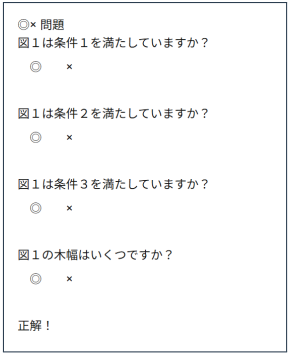


Figure: 修正前

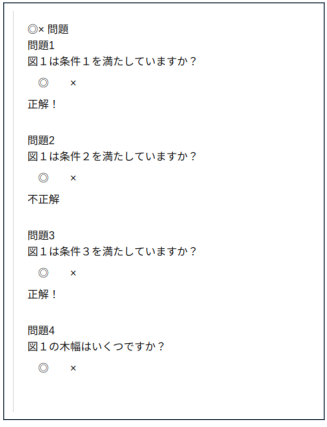


Figure: 修正後

→ 個別に正答判定を出力・保持できるようになった.

目次

WSL 環境でのエラー

- なぜか Windows の VScode（WSL）で Web アプリケーションにエラーが出た.
- データベースと Next.js の統合がうまく行かない.
- .env ファイルにある環境変数が読み込まれないことが原因.
- prisma.config.ts に `import 'dotenv/config'` を追加することで解消できた.

エラーの原因候補

1. Windows 上の VSCode + WSL 環境では，環境変数の伝播や改行コードの違いにより，`.env` が正しく読まれないことがある．
2. Prisma CLI は内部で `dotenv` を呼び出すが，WSL 経由では `process.env` に値が渡らない場合がある．
3. Ubuntu ネイティブ環境では，シェル経由で正しく環境変数が設定されるため問題が起きなかった．

prisma.config.ts と環境変数の自動読み込み

- Prisma 5 まででは、CLI が自動的に `.env` を読み込んでいた.
- Prisma 6 以降では，設定を TypeScript モジュールとして記述する `prisma.config.ts` が導入された.
- 設定ファイルが TypeScript/ESM モジュールとして扱われる.
- 仕様変更：「TypeScript モジュールとして読み込まれる設定ファイルからは、.env を自動で読み込まない」
- 理由：ESM モジュール読み込み時に環境変数を暗黙的に扱うと挙動が不明確になるため.
- そのため，明示的に `import "dotenv/config";` を追加して環境変数を読み込む必要がある.

Prisma CLI（Prisma command line interface）とは

- Prisma の各種操作をターミナルから実行するためのコマンド群 [2].
- データベーススキーマの管理やクライアント生成を行う.
- 例：
  - `npx prisma init`：初期設定
  - `npx prisma generate`：Prisma Client 生成
  - `npx prisma migrate dev`：DB マイグレーション
  - `npx prisma studio`：GUI でデータ確認
- CLI は内部で `.env` を読み込んで環境変数を設定する.
- ただし WSL 環境ではこの自動読み込みが失敗する場合がある.

解決策

明示的に .env を読み込む（prisma.config.ts）

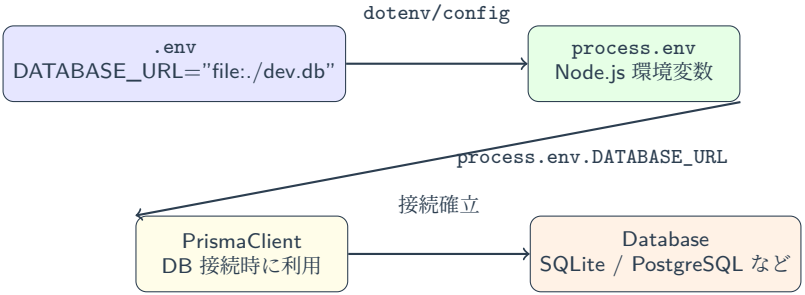
```ts
import "dotenv/config";
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();
export default prisma;
```

- この 1 行により、Node.js 起動時に .env が読み込まれる.
- PrismaClient が process.env.DATABASE_URL を利用可能に.

環境変数と process.env

- Node.js にはグローバルオブジェクト process が存在する.
- その中の process.env は, OS や .env ファイルに定義された環境変数を保持.
- 例: process.env.DATABASE_URL はデータベース接続 URL を表す.
- dotenv/config を読み込むことで, .env の内容が process.env に展開される.

環境変数の流れ（Prisma 利用時）



- `dotenv/config` が `.env` の内容を `process.env` に展開.
- PrismaClient が `process.env.DATABASE_URL` を利用して DB に接続.

目次

前回までの課題点と解決策

課題点:

- 頂点から出る辺の位置が固定されている.
- 辺同士がぶつかり, 綺麗なグラフになっていない.

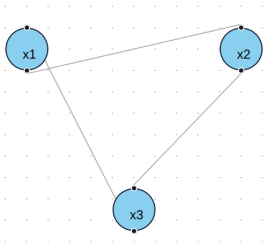


Figure: 今までの生成図形

解決策:

⇒ **辺を出す位置を頂点の中心の裏側から出るようにする.**

React Flow の全体構造

React Flow とは

React Flow は「Node（頂点）」と「Edge（線）」で構成されたグラフを, ブラウザ上で直感的に表示・操作できるライブラリ.

- 主な構成要素：
  - **Node（ノード）**：丸や四角などの図形.
  - **Edge（エッジ）**：ノード同士をつなぐ線.
  - **Handle（ハンドル）**：ノード上の「線の接続点」.
  - **ReactFlowProvider**：全体の状態（ノード・エッジ・選択状態など）を管理.
- ノードやエッジの配置・ズームなどは React Flow が自動的に処理.
- 独自の見た目を作るときは `CustomNode` のようにコンポーネントを定義.

各構成要素のプロパティ I

- **Node**：
  - id：一意な識別子.
  - type：表示に使うコンポーネントの種類（例：CustomNode）.
  - position：座標（x, y）.
  - data：ノードに渡すカスタムデータ（例：ラベル名など）.
- **Edge**：
  - source：出発ノードの id.
  - target：到着ノードの id.
  - type：エッジの種類（直線・曲線など）.

各構成要素のプロパティ II

- **Handle**：
  - type："source" または "target"（エッジの出発点／到達点を指定）.
  - id：ハンドルの識別子（ノード内で複数ハンドルを区別）.
  - position：ハンドルの位置（上・下・左・右）を `Position.Top / Bottom / Left / Right` で指定.
  - style：CSS で見た目をカスタマイズ（例：透明にして中心に配置）.
  - isConnectable：接続を許可するかどうか（true/false）.

- **ReactFlow コンポーネント**がこれらを統合し, 描画とイベント処理を行う.

Node Componentの作成 I

Node Component（ノードコンポーネント）：
「ノード単体の見た目と挙動（色・形・サイズ・ラベル・Handle の配置など）」を定義する場所.

```
1  export default function CustomNode({ data, selected }: NodeProps)
       {
2    return (
3      <div
4        style={{
5          background: selected ? "#FFD700" : "#89CFF0",
6          borderRadius: "50%",
7          width: 40,
8          height: 40,
9          display: "flex",
10         justifyContent: "center",
```

Node Componentの作成 II

```
11          alignItems: "center",
12          fontWeight: "bold",
13          color: "#000",
14          position: "relative",
15        }}
16      >
17        {data.label}
18
19        <Handle
20          type="source"
21          id="center"
22          position={Position.Right}
23          style={{ ... }}
24        />
25
26        <Handle
```

Node Componentの作成 III

```
27            type="target"
28            id="center"
29            position={Position.Left}
30            style={{ ... }}
31          />
32        </div>
33      )
34    }
```

CustomNode の基本構造

```tsx
export default function CustomNode(
  { data, selected }: NodeProps
) {
  return (
    <div>
      {data.label}
      <Handle type="source" id="center"
          />
      <Handle type="target" id="center"
          />
    </div>
  )
}
```

- **CustomNode**:
  React Flow における「ノード 1 つの描画定義」を担うコンポーネント.
- **NodeProps**: React Flow が各ノードに渡す情報の型.
- 引数 **data**, **selected**:
  - **data.label**: ノードのラベル.
  - **selected**: ユーザーが選択中かどうか.
- **<Handle>**：
  エッジ接続点. 1 つのノード内に複数設置できる.

ノード本体のスタイル

```
<div
  style={{
    background: selected ? "#FFD700" :
        "#89CFF0",
    borderRadius: "50%",
    width: 40,
    height: 40,
    display: "flex",
    justifyContent: "center",
    alignItems: "center",
    fontWeight: "bold",
    color: "#000",
    position: "relative",
  }}
>
  {data.label}
</div>
```

- background：
  - 選択時：金色（#FFD700）.
  - 非選択時：水色（#89CFF0）.
- `borderRadius: "50%"`：円形ノード.
- width × height：40px × 40px.
- `display: flex` と中央揃えでラベルを中央配置.
- `position: "relative"`：内部のハンドルを相対的に配置できるようにする.

Handle の役割

```
<Handle
  type="source"
  id="center"
  position={Position.Right}
  style={{
    top: "50%",
    left: "50%",
    transform: "translate(-50%, -50%)",
    background: "transparent",
    border: "none",
  }}
/>
```

- type:
  ‘”source”（線を出す）or ”target”（線を受ける）.
- position:
  接続位置（例：Position.Left, Position.Right）.
- id:
  エッジの sourceHandle / targetHandle と対応.
- style: CSS で位置や透明化を指定.
- この例ではどちらの Handle もノード中央に配置.
- 背景を透明にし, 見えないが線を接続できる.

実行結果

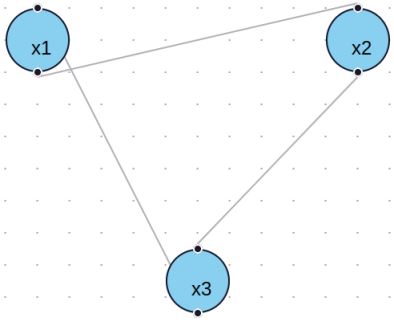


Figure: 修正前

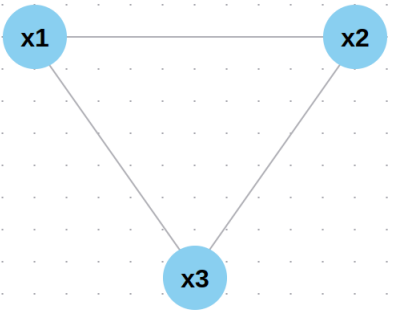


Figure: 修正後

→ グラフらしくすることに成功.

目次

今回の進捗状況

1. ~~◎ × 問題は正答判定可能だが, 数値の入力問題でも正答判定が不可能.~~
2. ~~それぞれの問題の正解・不正解が保存されない.~~
3. ~~React Flow で思ったようにグラフが描けない.~~
4. 問題文と画像生成が連携されていない.
5. 問題がページ分割されずに一覧で出てくる.

→残りの 2 つと◎ × 問題以外にも対応可にさせることに取り組む.

参考文献 I

[1] 高林佳稀 手島拓也, 吉田健人.
TypeScript と React/Next.js でつくる実践 Web アプリケーション開発.
株式会社技術評論社, 2022.

[2] Prisma. Prisma CLI.
`https://www.prisma.io/docs/orm/tools/prisma-cli`. アクセス日: 2025-11-14.