Projekt Programistyczny Indywidualny

(2022/2023)

"Wizualizacja Grafów"

autor: Filip Krawczyk

Spis treści:

Wstęp

Celem projektu jest stworzenie aplikacji okienkowej umożliwiającej użytkownikowi wizualne tworzenie grafów skierowanych oraz sprawdzanie czy zawierają one cykl Eulera, ścieżkę Eulera, cykl Hamiltona, ścieżkę Hamiltona oraz znajdowanie najkrótszej (zawierającą najmniejszą ilość krawędzi) trasy między punktami.

Definicje

Ścieżka Eulera – taka ścieżka w grafie, która przechodzi przez każdą krawędź dokładnie raz.

Cykl Eulera – ścieżka Eulera która zaczyna i kończy się na tym samym wierzchołku.

Graf eulerowski – graf zawierający cykl Eulera

Graf pół-eulerowski – graf zawierający ścieżkę, ale nie cykl, Eulera

Ścieżka Hamiltona – taka ścieżka w grafie, która przechodzi przez każdy wierzchołek dokładnie raz

Cykl Hamiltona – ścieżka Hamiltona która zaczyna i kończy się na tym samym wierzchołku

Graf hamiltonowski – graf zawierający cykl Hamiltona

Graf pół-hamiltonowski – graf zawierający ścieżkę, ale nie cykl, Hamiltona

DFS (Depth-first search) – algorytm przeszukiwania grafu, który polega na eksplorowaniu w głąb, czyli przechodzeniu jak najdalej w głąb jednej gałęzi grafu, zanim wróci się do poprzedniego wierzchołka i kontynuuje się eksplorację innej gałęzi

BFS (Breadth-first Search) – algorytm przeszukiwania grafu, który eksploruje strukturę danych poprzez odwiedzanie wszystkich sąsiednich wierzchołków na tym samym poziomie, zanim przejdzie do wierzchołków na kolejnym poziomie

Graf silnie spójny – graf skierowany, w którym każde dwa wierzchołki są połączone drogą

Charakterystyka modelu

Algorytm do grafu eulerowskiego

Do sprawdzania, czy graf jest eulerowski zostały wykorzystane warunki z wykładu 12 z Matematyki Dyskretnej (str. 14) prowadzonego przez Marka A. Kowalskiego. Na jego podstawie został napisany algorytm, a własność spójności grafu jest sprawdzana za pomocą algorytmu DFS.

> Multigraf skierowany ma ścieżkę Eulera wtedy i tylko wtedy, gdy jest spójny i dla każdego wierzchołka v mamy $\deg^+(v) = \deg^-(v)$, z możliwym jedynym wyjątkiem dwóch wierzchołków a, b takich, że $\deg^+(a) = \deg^-(a) + 1$ oraz $\deg^-(b) = \deg^+(b) - 1$

> Multigraf skierowany ma cykl Eulera wtedy i tylko wtedy, gdy jest spójny i dla każdego wierzchołka v mamy $\deg^+(v) = \deg^-(v)$.

Algorytm do grafu hamiltonowskiego

Do sprawdzania czy graf jest hamiltonowski został wykorzystany rekurencyjny algorytm z powracaniem, czyli algorytm rozpoczyna przechodzenie po grafie od dowolnego wierzchołka grafu jako potencjalnego początku cyklu, dodaje ten wierzchołek do aktualnej ścieżki. Sprawdza jeśli ścieżka zawiera wszystkie wierzchołki grafu i/albo istnieje krawędź między bieżącym wierzchołkiem a startowym wierzchołkiem, to zwraca informacje, że istnieje cykl/ścieżka Hamiltona. Jeśli aktualny wierzchołek nie spełnia tych warunków to odwiedza nieodwiedzonego sąsiada, albo jeśli nie ma takiego, wraca do poprzedniego wierzchołka, usuwa wierzchołek ze ścieżki i podejmuje próbę jeszcze raz. Po wykonaniu tego wraca do momentu sprawdzania czy ścieżka zawiera wszystkie wierzchołki grafu. Jeśli nie ma już żadnych wierzchołków do odwiedzenia, a ścieżka albo cykl hamiltona nie został znaleziony, to algorytm kończy działanie i zwraca informację, że graf nie jest hamiltonowski.

Algorytm do wyznaczania najkrótszej trasy

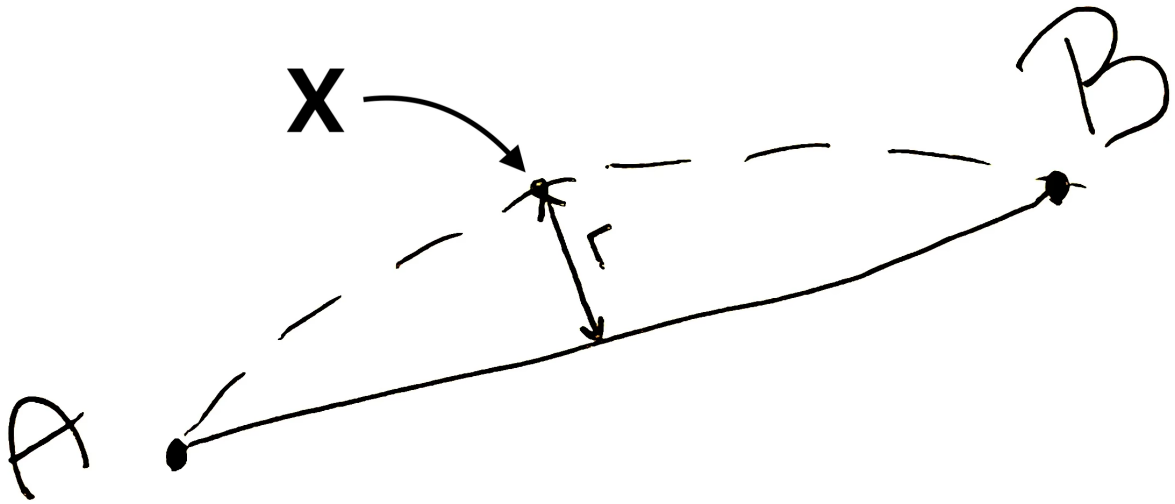
Do wyznaczania trasy między punktami został wykorzystany algorytm BFS, aby znaleźć trasę z najmniejszą ilością krawędzi.

Estetyka renderowania wielu krawędzi między dwoma wierzchołkami

Aby wielokrotne krawędzie między wierzchołkami nie nakładały się na siebie, są one zakrzywiane, jeśli jest ich więcej niż 1. Aby zakrzywić linię silnik graficzny potrzebuje 3 punkty: początkowy, środkowy i końcowy. Do znalezienia punktu środkowego, który musi zostać przesunięty względem tego gdzie normalnie był, została użyta następująca transformacja wektorów:

$$\frac{A+B}{2} + \left(\frac{B-A}{||B-A||_2}\right)^{\mathrm{T}} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \Gamma$$

gdzie A i B to współrzędne punktów początkowego i końcowego (wektory dwuwymiarowe), a Γ to pożądany dystans (skalar) od środka pomiędzy tymi wierzchołkami. Ten algorytm znajduje punkt środkowy pomiędzy A i B, potem przesuwa go prostopadle do linii przecinającej A i B o długość Γ, aby uzyskać punkt X widoczny na poniższej wizualizacji.



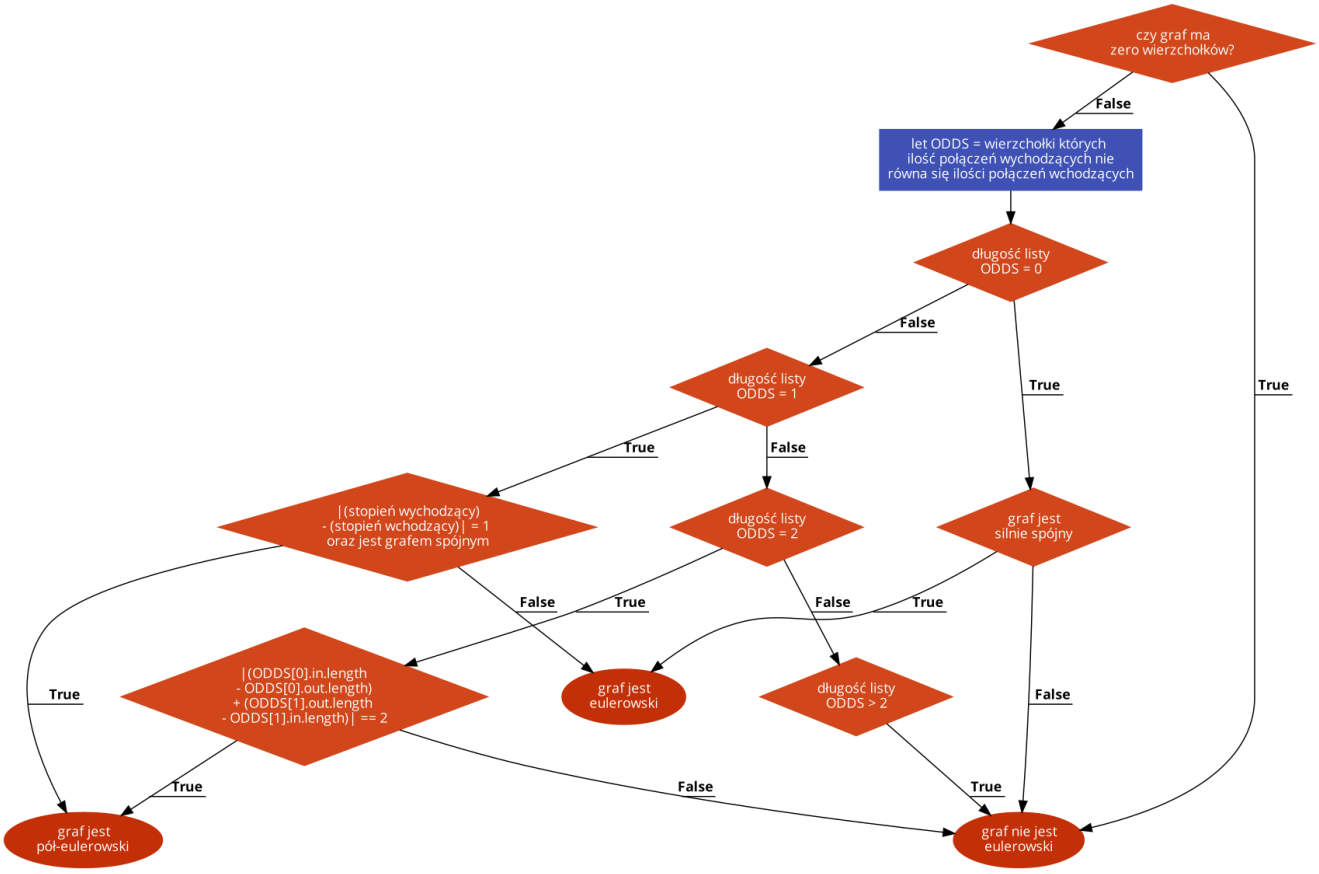
Opis programu

Program został stworzony z użyciem następujących technologi webowych:

- został napisany w języku TypeScript
- do tworzenia interfejsu została użyta biblioteka React
- do wizualizowania grafu została wykorzystana biblioteka Konva
- do stylowania została użyta biblioteka Tailwind.css
- do zarządzania narzędziami został użyty Vite, a projekt był bazowany na gotowym template React+TypeScript
- do zarządzania stanem aplikacji została wykorzystana biblioteka zustand
- do testów jednostkowych została użyta biblioteka vitest

Następnie ten program został przetworzony na aplikację okienkową z użyciem technologii Tauri
Platformy docelowe to macOS i Windows.

Implementacja algorytmu do sprawdzania czy graf jest grafem eulerowskim



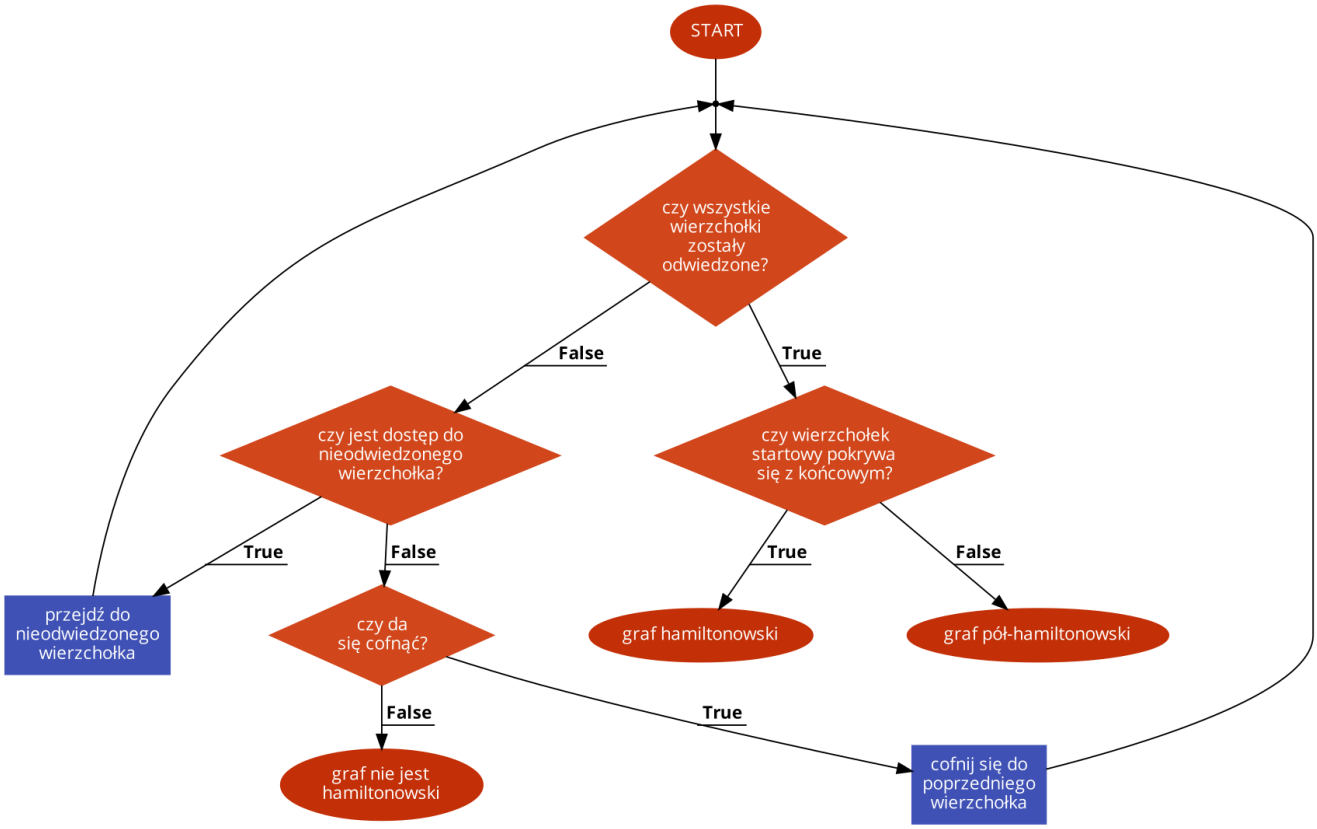
.out to lista krawędzi wychodzących, .in to lista krawędzi wchodzących do danego wierzchołka, .length to długość danej listy

```
// src/math/eulerian.ts

export enum Eulerian {
        /** not eulerian */
        not,
        /** semi-eulerian graph */
        semi,
        /** eulerian graph */
        fully,
}

/** Checks if given graph is eulerian */
export function isEulerian(vertices: Vertex[], edges: Edge[]): Eulerian
{
        let graph = toConnectedGraph(vertices, edges)

        if (graph.length === 0) return Eulerian.not

        const odds: number[] = []
        for (let i = 0; i < graph.length; i++) {
                if (graph[i].in.length !== graph[i].out.length) {
                        odds.push(i)
                }
```

```
        }

        switch (odds.length) {
        case 0: // ALL EVEN
                if (isStronglyConnected(graph)) {
                        return Eulerian.fully
                } else {
                        return Eulerian.not
                }

        case 1: // 1 ODD
                const odd = graph[odds[0]]

                if (
                        Math.abs(odd.in.length - odd.out.length) === 1
&&
                        isConnected(graph, odds[0] === 0 ? 1 : 0)
                ) {
                        return Eulerian.semi
                } else {
                        return Eulerian.not
                }

        case 2: // 2 ODDS
                const first = graph[odds[0]]
                const second = graph[odds[1]]

                if (Math.abs(first.in.length - first.out.length +
                        (second.out.length - second.in.length)) === 2)
{
                        return Eulerian.semi
                } else {
                        return Eulerian.not
                }

        default: // >2 ODDS
                return Eulerian.not
        }
}

function isStronglyConnected(graph: ConnectedGraph, index: number = 0):
boolean {
        let visited: boolean[] = Array(graph.length).fill(false)

        // traverse the graph with DFS
        function dfs(index: number, reverse: boolean = false) {
```

```
                if (visited[index]) return
                        visited[index] = true

                if (reverse) {
                        graph[index].in.map(e => dfs(e, true))
                } else {
                        graph[index].out.map(e => dfs(e))
                }
        }

        dfs(index)

        if (visited.includes(false)) return false

        // check the inverse of the graph
        visited = visited.fill(false)
        dfs(index, true)

        if (visited.includes(false)) return false

        return true
}

function isConnected(graph: ConnectedGraph, index: number): boolean {
        let visited: boolean[] = Array(graph.length).fill(false)

        // traverse the graph with DFS
        function dfs(index: number) {
                if (visited[index]) return
                visited[index] = true

                graph[index].out.map(dfs)
        }

        dfs(index)

        return !visited.includes(false)
}
```

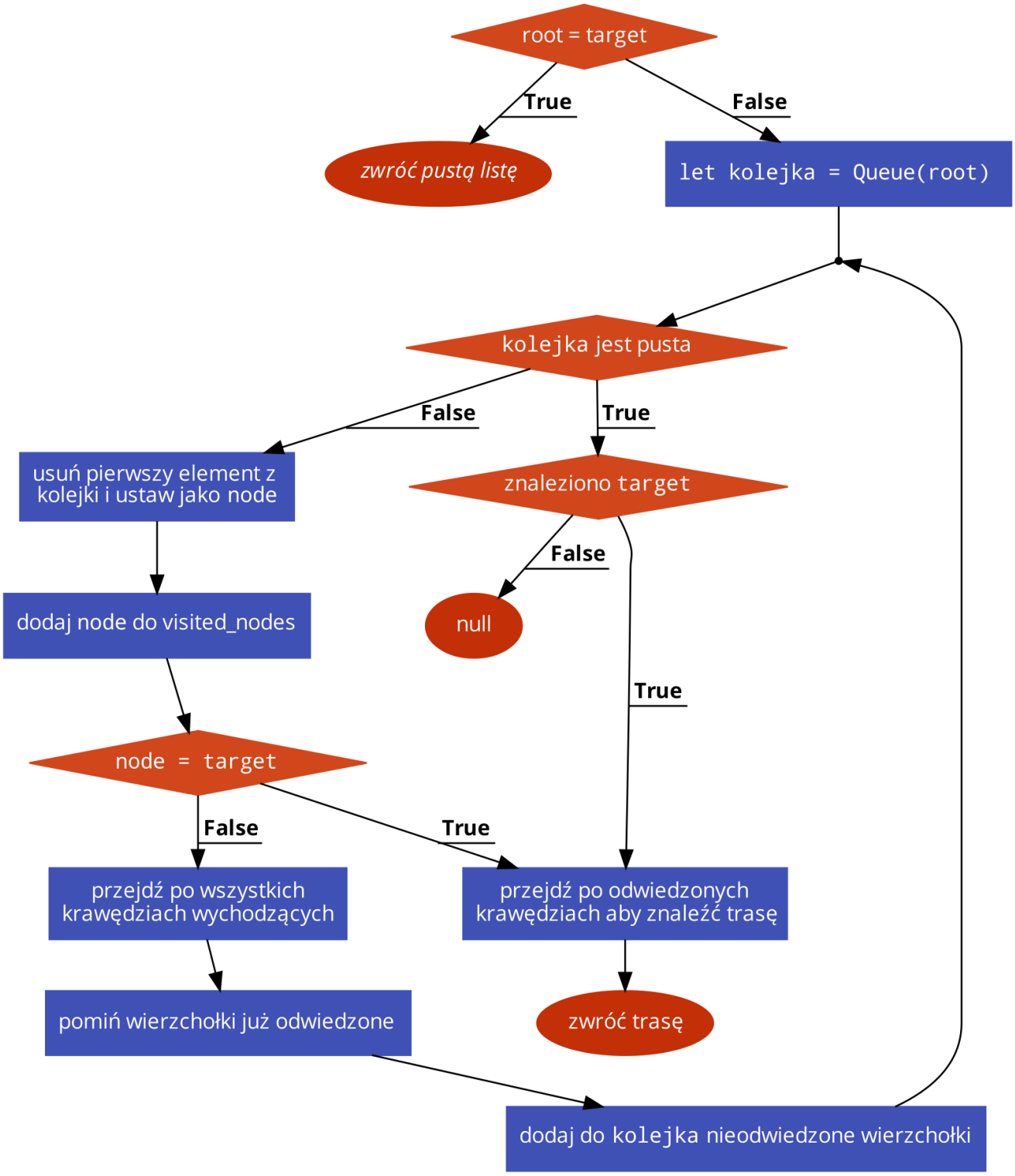
Implementacja algorytmu do sprawdzenia czy graf jest grafem hamiltonowskim




```
// src/math/hamiltonian.ts

export enum Hamiltonian {
        /** not hamiltonian graph */
        not,
        /** semi-hamiltonian graph */
        semi,
        /** hamiltonian graph */
        fully,
}

/**
 * Check if graph is hamiltonian or semi-hamiltonian using backtracking
method
 */
export function isHamiltonian(verts: Vertex[], edges: Edge[]):
Hamiltonian {
        const graph = toConnectedGraph(verts, edges)
        if (graph.length === 0) return Hamiltonian.not

        let path: number[] = []
        let visited: boolean[] = Array(graph.length).fill(false)

        function dfs(index: number): Hamiltonian {
                visited[index] = true
```

```
                if (path.includes(index)) {
                        if (path.length === graph.length) {
                                if (path[0] === index) {
                                        return Hamiltonian.fully
                                }
                                return Hamiltonian.semi
                        }
                        return Hamiltonian.not
                }
                path.push(index)

                for (const vert of graph[index].out) {
                        const result = dfs(vert)
                        if (result !== Hamiltonian.not) return result
                }
                if (path.length === graph.length) {
                        return Hamiltonian.semi
                }
                path.pop()

                return Hamiltonian.not
        }

        let startingIndex = 0
        while (startingIndex != -1) {
                const result = dfs(startingIndex)
                if (result != Hamiltonian.not ||
!visited.includes(false))
                        return result
                startingIndex = visited.findIndex(e => !e)
        }

        return Hamiltonian.not
}
```

Implementacja algorytmu do znajdywania najkrótszej trasy




```
// src/math/pathfinding.ts

export function findPath(
        vertices: Vertex[],
        edges: Edge[],
        root: number,
        target: number
): number[] | null {
        if (root === target)
                return []
        if (root < 0 || target < 0)
```

```
                return null
        if (root >= vertices.length || target >= vertices.length)
                return null

        const graph = toConnectedGraph(vertices, edges)

        let visitedNodes: number[] = []
        let visited: boolean[] = new Array(graph.length).fill(false)

        let q = new Queue<number>([root])
        visited[root] = true

        let found = false
        outsideLoop: while (!q.empty()) {
                const check = q.dequeue()!

                visitedNodes.push(check)

                for (const node of graph[check].out) {
                        if (visited[node]) continue

                        if (node === target) {
                                found = true
                                break outsideLoop
                        }
                        q.enqueue(node)
                        visited[node] = true
                }
        }
        if (!found) return null

        let path: number[] = [target]
        let current = target
        while (current !== root) {
                current = visitedNodes
                        .find(e => graph[current].in.includes(e))!
                path.push(current)
        }

        return path.reverse()
}
```

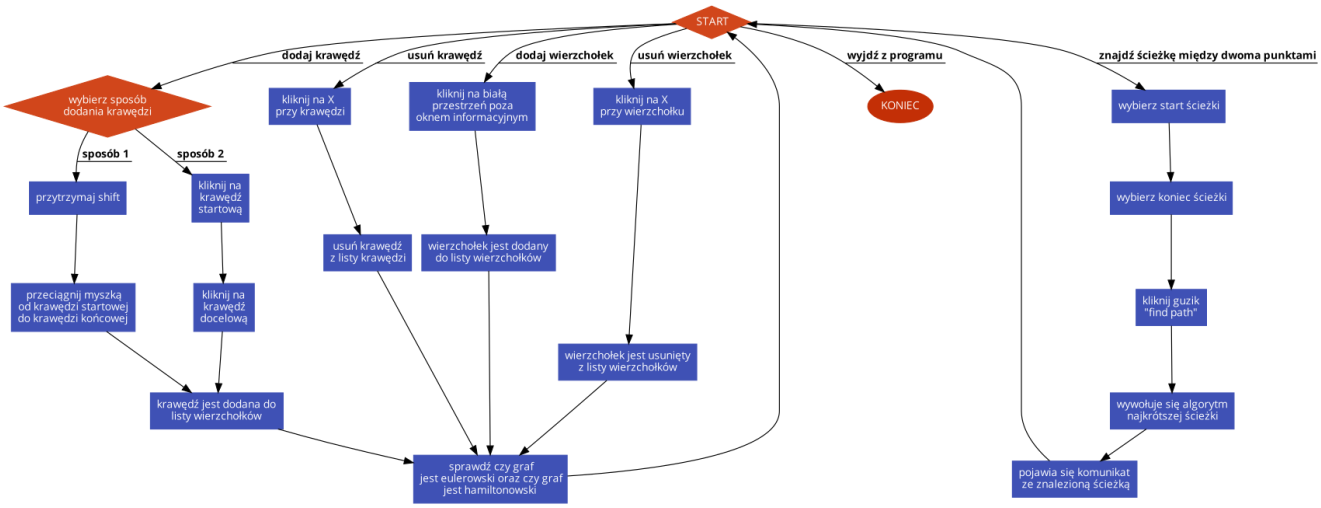
Renderowanie wierzchołków

Przy renderowaniu wierzchołki są reprezentowane za pomocą kształtu koła. Aby krawędź, reprezentowana za pomocą strzałki, wskazywała na koło, a nie do środka

koła, współrzędne punktu na krawędzi tego koła wyznaczono za pomocą funkcji trygonometrycznych.

```
// src/math/util.ts

export function calcAngle(a: Point, b: Point) {
        return Math.atan((b.y - a.y) / (b.x - a.x))
}

export function arrowStickingPoint(a: Point, b: Point): [Point, Point]
{
        const angle = (() => {
                if (a.x > b.x) {
                        return Math.PI + calcAngle(a, b)
                } else {
                        return calcAngle(b, a)
                }
        })()

        const x = CIRCLE_RADIUS * Math.cos(angle)
        const y = CIRCLE_RADIUS * Math.sin(angle)

        return [
                { x: a.x + x, y: a.y + y },
                { x: b.x - x, y: b.y - y },
        ]
}
```

Schemat działania aplikacji



Struktura programu

- `public/` - pliki statyczne importowane z pliku html

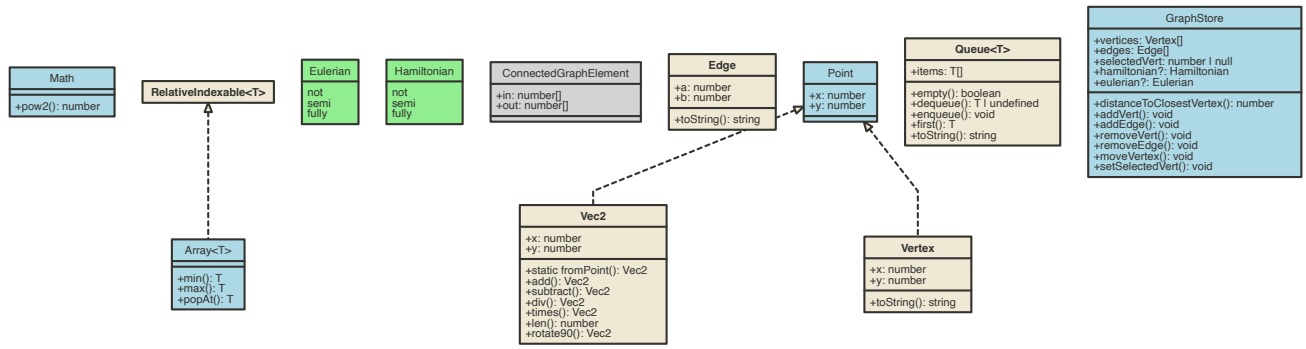
- `src/` - pliki kodu źródłowego
  - `components/` - komponenty uniwersalne
    - `Button.tsx` - guzik z dodanymi stylami
  - `hooks/` - hooki reactowe
    - `useDraggingLine.tsx` - hook zawierający logikę zarządzającą łączeniem wierzchołków
    - `useKeyHold.tsx` - hook zawierający logikę do obsługi przytrzymywania klawiszy
  - `math/` - funkcje wykonujące obliczenia matematyczne
    - `eulerian.ts` - zawiera funkcje do sprawdzania, czy graf jest eulerowski
    - `eulerian.test.ts` - testy jednostkowe do `eulerian.ts`
    - `hamiltonian.ts` - zawiera funkcje do sprawdzania, czy graf jest hamiltonowski
    - `hamiltonian.test.ts` - testy jednostkowe do `hamiltonian.ts`
    - `pathfinding.ts` - zawiera funkcję do znalezienia najkrótszej ścieżki między dwoma punktami
    - `utils.ts` - dodatkowe funkcje z obliczeniami matematycznymi
  - `models/` - struktury potrzebne w innych częściach programu
    - `edge.ts` - krawędzie do grafu
    - `point.ts` - punkt z koordynatami x i y
    - `vec2.ts` - wektor 2-wymiarowy (używany do zakrzywiania krawędzi)
    - `vertex.ts` - wierzchołki do grafu
  - `sections/` - komponenty reprezentujące sekcje aplikacji
    - `App.tsx` - komponent trzonowy aplikacji
    - `ControlPanel.tsx` - komponent panelu kontrolnego
    - `GraphCanvas.tsx` - komponent powierzchni na której jest wyświetlany graf
  - `state/` - struktury do zarządzania stanem aplikacji
    - `graph.ts` - stan aplikacji
  - `constants.ts` - stałe
- `src-tauri/` - folder z projektem tauri

pliki konfiguracyjne bibliotek zostały pominięte

Główny punkt startowy programu jest w `src/sections/App.tsx` który zawiera komponent trzonowy aplikacji. Wskazuje on na dwa najważniejsze komponenty w tej aplikacji: `ControlPanel.tsx` oraz `GraphCanvas.tsx` gdzie pierwszy z nich zawiera panel z którego użytkownik może edytować graf a drugi jest komponentem wizualizującym ten graf. Jedyne dalsze komponenty interfejsu używane to HTML i

`src/components/Button.tsx`. Do obsługi stanu globalnego aplikacji jest używany globalny store (`GraphStore`) generowany przez bibliotekę `zustand` w pliku `src/state/graph.ts`. Wszelkiego rodzaju implementacje algorytmów oraz inne funkcje zawierające obliczenia znajdują się w folderze `src/math/`. `eulerian.ts` zawiera funkcje do sprawdzania, czy graf jest grafem Eulera, `hamiltonian.ts` zawiera funkcje do sprawdzania, czy graf jest grafem Hamiltona, a `pathfinding.ts` zawiera funkcje do znajdywania najkrótszej trasy między wierzchołkami. Wewnątrz `utils.ts` znajdują się inne obliczenia potrzebne do samego renderowania.

Diagram klas



`Math.pow2` jest rozszerzeniem globalnego obiektu `Math`. Dodaje metodę `pow2(x: number)` która podnosi `x` do potęgi 2.

```
// src/util.ts
Math.pow2 = (num: number) => num * num
```

`Array.min`, `Array.max`, `Array.popAt` są metodami rozszerzającymi globalny obiekt `Array` w celu ułatwienia znajdowania najmniejszych i największych wartości w listach, oraz w celu ułatwienia usuwania elementów z konkretnych miejsc w listach.

```
// src/util.ts

Array.prototype.min = function () {
        return Math.min(...this)
}

Array.prototype.max = function () {
        return Math.max(...this)
}

Array.prototype.popAt = function (index) {
        return this.splice(index, 1)[0]
}
```

`Eulerian` to enum reprezentujący możliwe stany grafu w relacji do bycia eulerowym.

```
// src/math/eulerian.ts

export enum Eulerian {
        /** not eulerian */
        not,
        /** semi-eulerian graph */
        semi,
        /** eulerian graph */
        fully,
}
```

`Hamiltonian` to enum reprezentujący możliwe stany grafu w relacji do bycia hamiltonowym.

```
// src/math/hamiltonian.ts

export enum Hamiltonian {
        /** not hamiltonian graph */
        not,
        /** semi-hamiltonian graph */
        semi,
        /** hamiltonian graph */
        fully,
}
```

`ConnectedGraphElement` jest strukturą reprezentującą element grafu skierowanego, w którym dla danej krawędzi można przechodzić w obydwu kierunkach.

```
// src/math/utils.ts
export type ConnectedGraphElement = { in: number[]; out: number[] }
export type ConnectedGraph = Array<ConnectedGraphElement>
```

`Edge` jest strukturą reprezentującą krawędź grafu

```
// src/models/edge.ts

export class Edge {
        /** source */
        public a: number
        /** target */
        public b: number
```

```
        constructor(a: number, b: number) { /* ... */ }

        toString(): string { ... }
}
```

`Vec2` jest strukturą reprezentującą macierz 2-wymiarową

```
// src/models/vec2.ts

/**
 * 2-dimensional vector
 */
export class Vec2 implements Point {
        static fromPoint(p: Point): Vec2 { /* ... */ }

        constructor(public x: number, public y: number) { /* ... */ }

        add(v: Vec2): Vec2 { /* ... */ }
        div(n: number): Vec2 { /* ... */ }
        len(): number { /* ... */ }
        rotate90(): Vec2 { /* ... */ }
        subtract(v: Vec2): Vec2 { /* ... */ }
        times(n: number): Vec2 { /* ... */ }
}
```

`Vertex` jest strukturą reprezentującą wierzchołek grafu

```
// src/models/vertex.ts

export class Vertex implements Point {
        /** visual X */
        public x: number
        /** visual Y */
        public y: number

        constructor(x: number, y: number) { /* ... */ }

        toString(): string { /* ... */ }
}
```

`Vec2` i `Vertex` implementują interfejs `Point` który reprezentuje punkt na 2-wymiarowej płaszczyźnie

```
// src/models/point.ts
```

```
export interface Point {
        x: number
        y: number
}
```

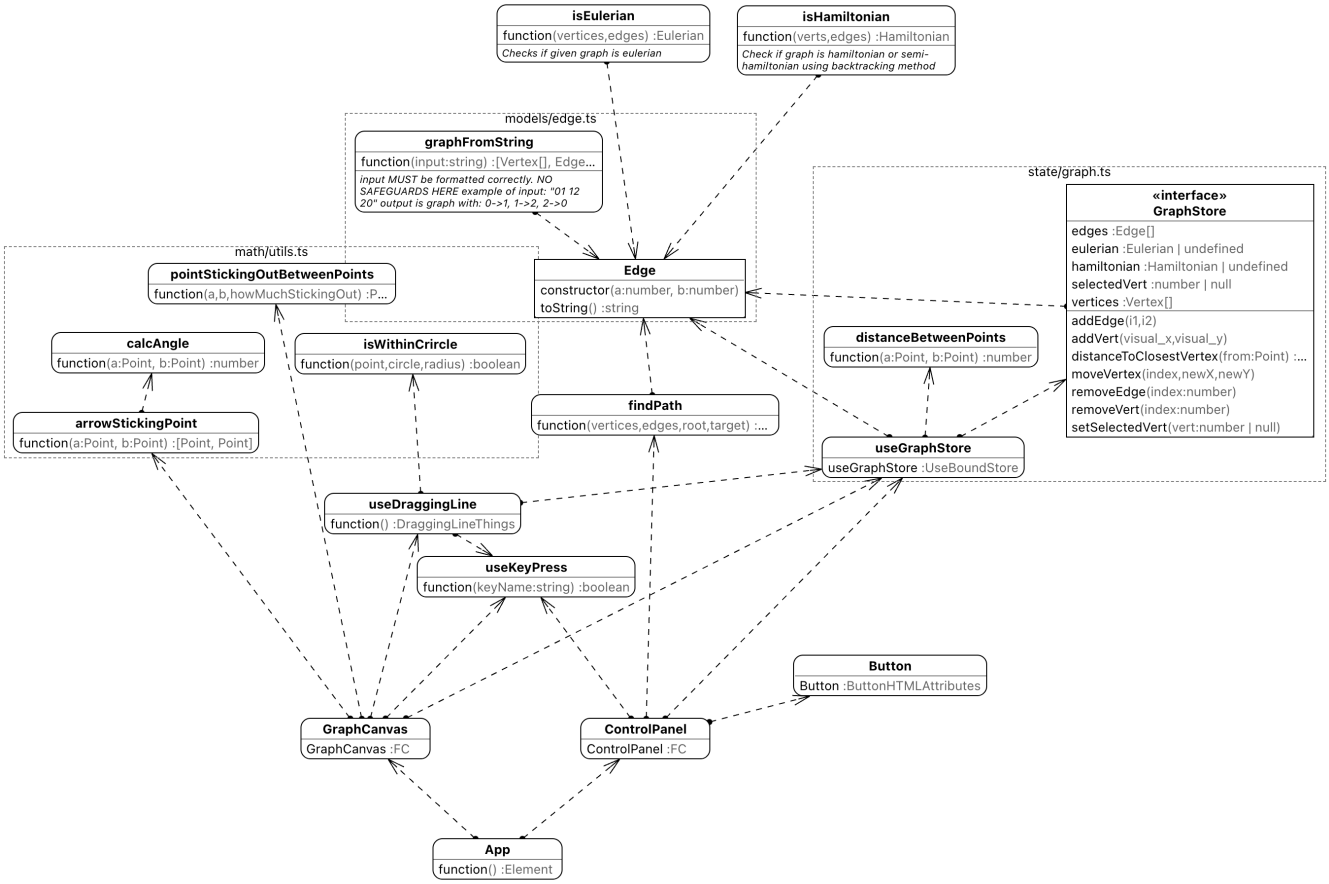
`Queue` jest implementacją kolejki

```
// src/models/queue.ts

export class Queue<T> {
        items: T[] = []

        constructor(initial: T[] = []) { /* ... */ }

        empty() { /* ... */ }
        dequeue() { /* ... */ }
        enqueue(item: T) { /* ... */ }
        first() { /* ... */ }

        toString() { /* ... */ }
}
```

`GraphStore` jest obiektem reprezentującym stan globalny aplikacji. Zawiera on zmienne reprezentujące wierzchołki, krawędzie, aktualnie zaznaczoną krawędź, oraz czy dany graf na ekranie jest eulerowy, albo hamiltonowy. Posiada on także metody modyfikujące powyższe zmienne, oraz metodę zwracającą dystans od punktu do najbliższego wierzchołka.

```
// src/state/graph.ts

export interface GraphStore {
        vertices: Vertex[]
        edges: Edge[]
        distanceToClosestVertex(from: Point): number
        addVert(visual_x: number, visual_y: number): void
        addEdge(i1: number, i2: number): void
        removeVert(index: number): void
        removeEdge(index: number): void
        moveVertex(index: number, newX: number, newY: number): void
        selectedVert: number | null
        setSelectedVert(vert: number | null): void
        hamiltonian?: Hamiltonian
        eulerian?: Eulerian
}
```

Diagram komponentów



Instrukcja obsługi

Instalacja

Program wymaga instalacji, na macOS należy uruchomić plik płyty wirtualnej `Graph Stuff_0.1.0_aarch64.dmg`, a następnie przenieść ręcznie `Graph Stuff.app` do folderu `Aplikacje`. Na systemie Windows należy uruchomić plik instalacyjny `Graph Stuff_0.1.0_x64.msi`, a następnie klikać kilkukrotnie guzik "Dalej", oraz na samym końcu "Zakończ". Uwaga: program na systemie Windows wymaga uprawnień administratora do poprawnego zainstalowania.

Działanie programu

Po otwarciu programu widać przed użytkownikiem białą powierzchnię z panelem kontrolnym w prawym górnym rogu. Kliknięcie w jakimkolwiek punkcie na białej powierzchni tworzy wierzchołki reprezentowane przez różowe kulki.

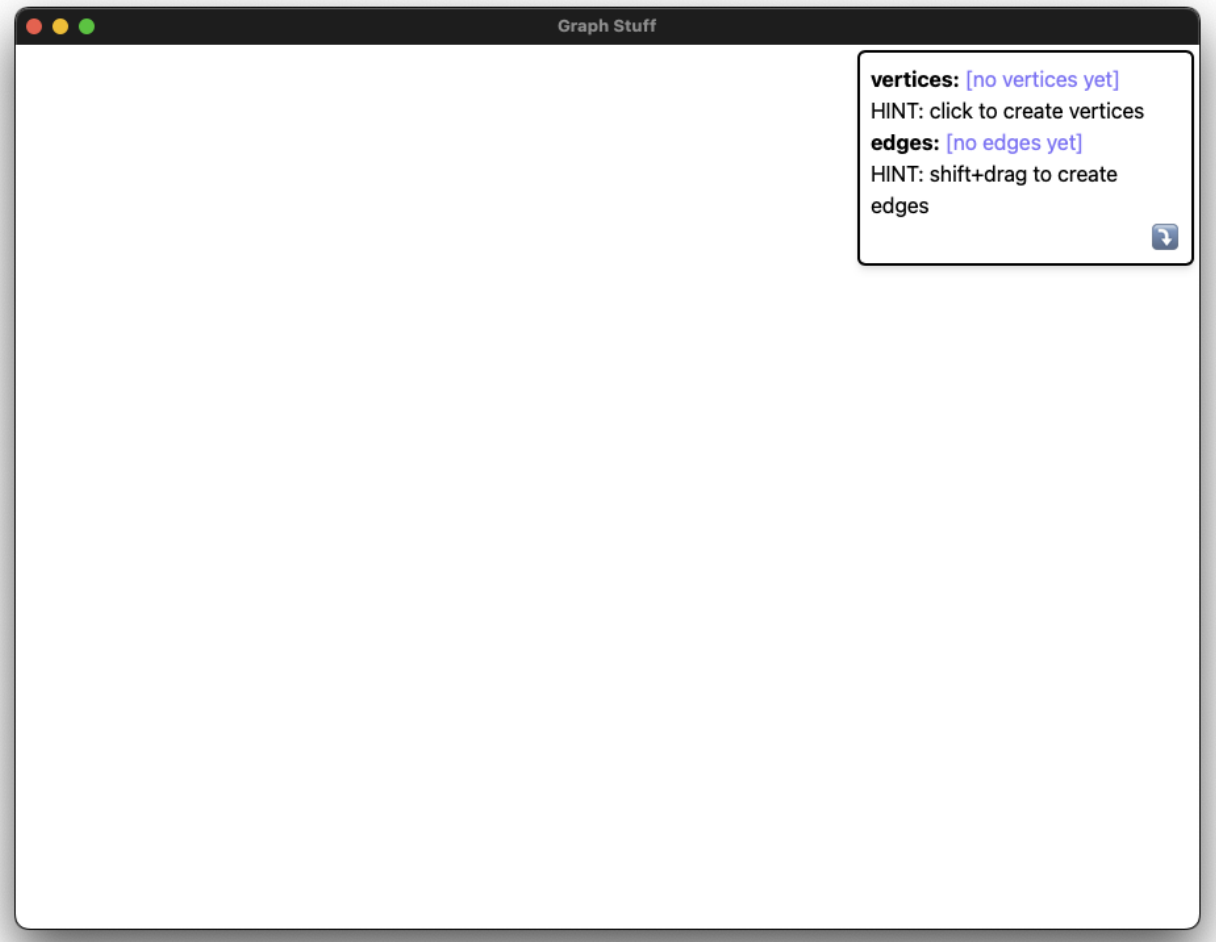
Aby tworzyć krawędzie między nimi wystarczy przytrzymać shift i lewy przycisk myszy przeciągnąć myszką od jednego wierzchołka do drugiego. W celu połączenia wierzchołka z samym sobą wystarczy kliknąć raz na niego i wybrać opcję "connect to itself" pojawiającą się na panelu kontrolnym.

Wierzchołki można też przenosić myszką w dowolne miejsca na ekranie. W przypadku niefortunnego przeciągnięcia ich za panel kontrolny można kliknąć na guzik w jego prawym dolnym rogu aby zmienić jego lokalizacje na prawy dolny róg ekranu.

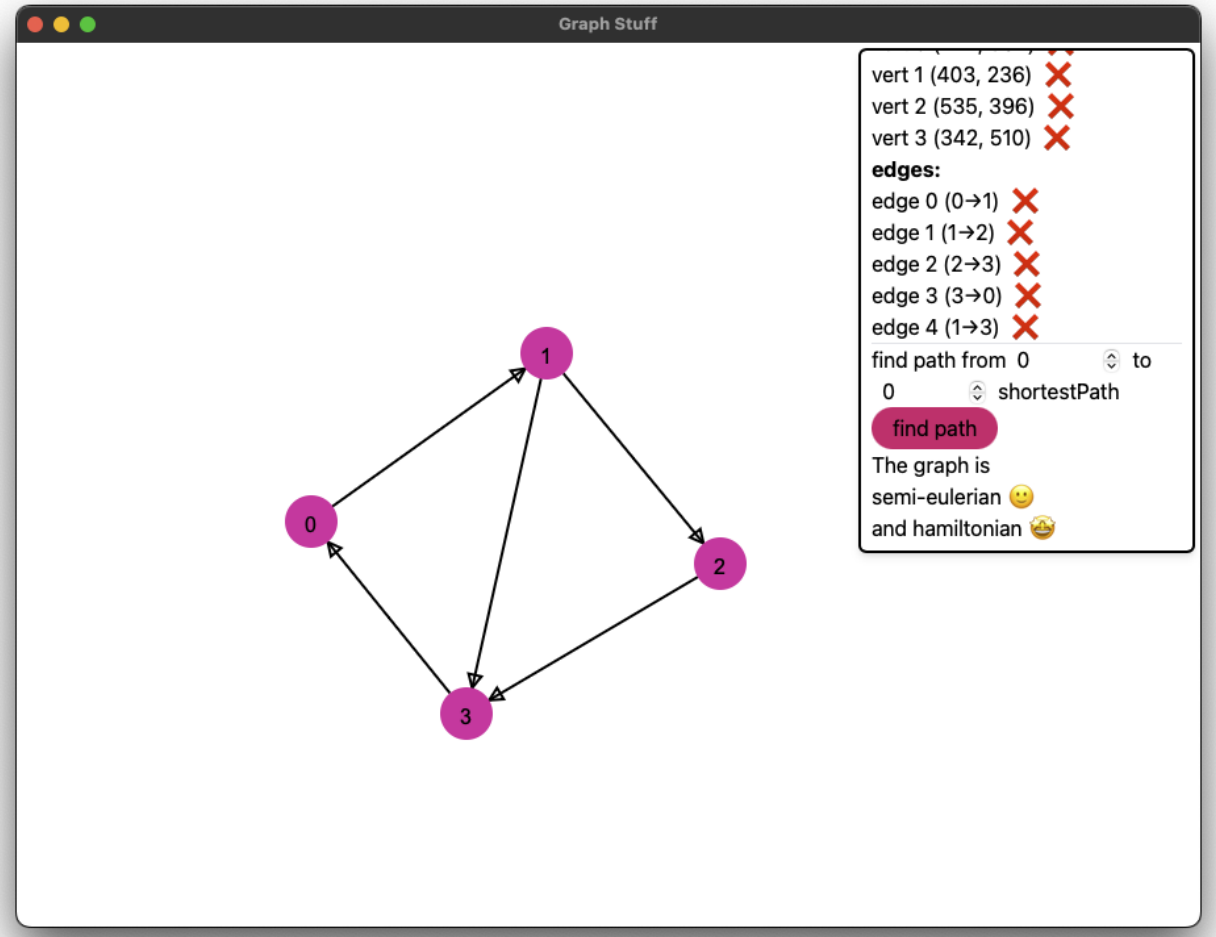
Działania widoczne na białej powierzchni są opisywane w panelu kontrolnym, który wyświetla listę wierzchołków i krawędzi oraz połączone nimi punkty. Dowolny wierzchołek lub krawędź można usunąć poprzez kliknięcie na guzik "❌".

znajdujący się przy każdym z nich.
Po utworzeniu przynajmniej jednego wierzchołka na panelu kontrolnym ukazują się dodatkowe funkcje i informacje: użytkownik ma możliwość sprawdzić najkrótszą trasę między dwoma punktami: wystarczy wpisać numer dwóch wierzchołków i kliknąć guzik "find path". Pod tym znajduje się sekcja gdzie wyświetlana jest użytkownikowi informacja czy graf jest hamiltonowski, pół-hamiltonowski, eulerowski lub pół-eulerowski. Wszystkie informacje na panelu kontrolnym aktualizują się na bieżąco w momencie tworzenia wierzchołków i krawędzi.

Widok po włączeniu aplikacji



Widok z utworzonym grafem



Bibliografia

- https://en.wikipedia.org/wiki/Eulerian_path#Properties
- https://en.wikipedia.org/wiki/Hamiltonian_path
- https://en.wikipedia.org/wiki/Strongly_connected_component
- https://mathspace.co/textbooks/syllabuses/Syllabus-1030/topics/Topic-20297/subtopics/Subtopic-266708/
- https://www.geeksforgeeks.org/hamiltonian-cycle/
- https://www.typescriptlang.org/
- https://react.dev/
- https://konvajs.org/
- https://tailwindcss.com/
- https://vitejs.dev/
- https://github.com/vitejs/vite/tree/main/packages/create-vite/template-react-ts
- https://github.com/pmndrs/zustand
- https://vitest.dev/
- https://tauri.app/

Zawartość płyty

- `Graph Stuff_0.1.0_aarch64.dmg`
- `Graph Stuff_0.1.0_x64.msi`