Michał Jagoda

Wizualizacja algorytmów szukania drogi

Opis tematu

Cel projektu

- Wizualizacja różnych algorytmów szukania drogi.
- Porównanie ich działania .
- Porównanie czasów wyszukiwania.

Realizacja

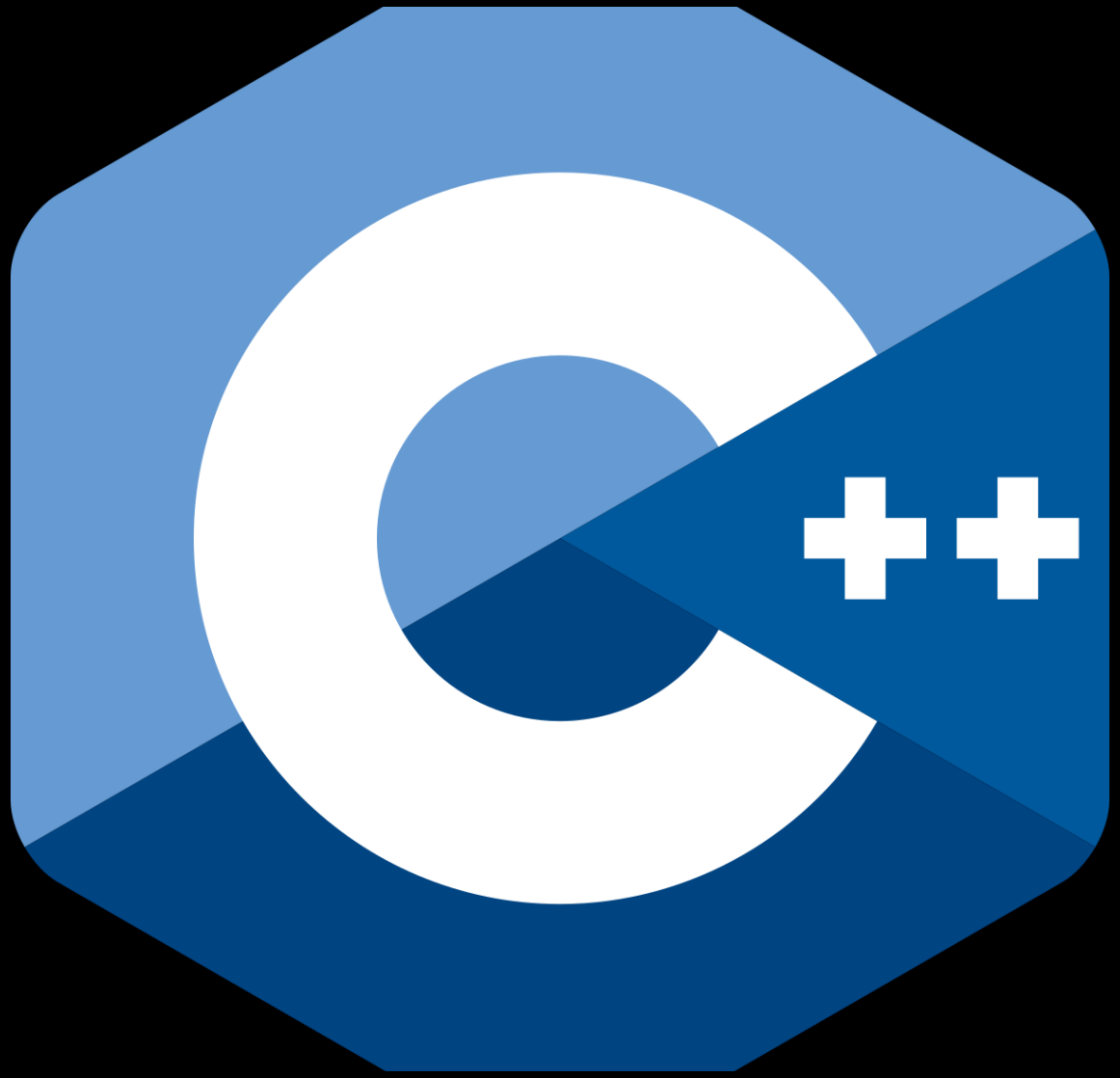
- HTML
- CSS
- JS
- BOOTSTRAP
- REACT

- C++
- WASM

Użyte technologie

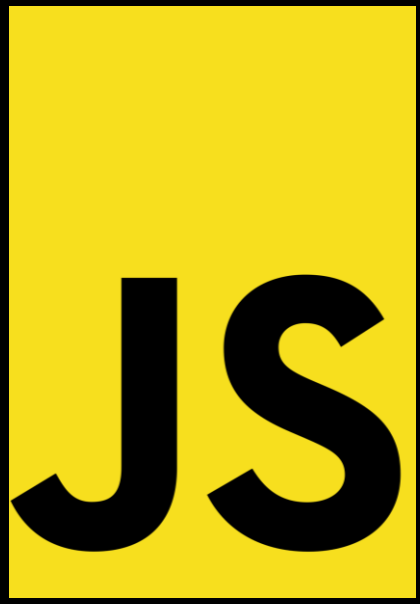
C++

- Bardzo duże możliwości
- Duża wydajność



WASM

- Możliwość uruchomienia niskopoziomowego kodu (C / C++ / Rust)
- Duża wydajność
- Gry, przetwarzanie obrazu i dźwięku
- Emscripten



HTML, CSS, JS

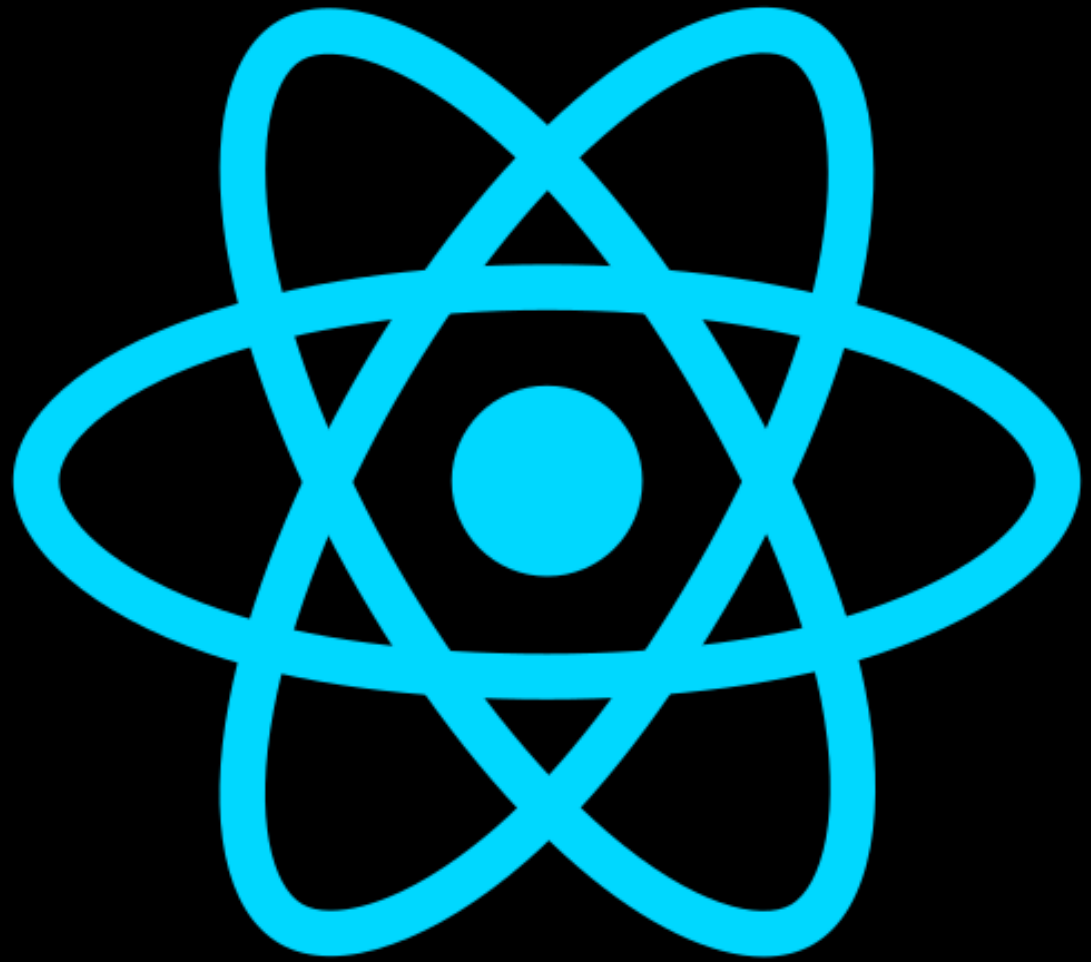
- HTML → struktura strony
- CSS → wygląd strony
- JS → interaktywność strony

Bootstrap

- Biblioteka HTML, CSS, JS
- Zestaw narzędzi ułatwiających tworzenie interfejsu graficznego

React

- Biblioteka JS służąca do tworzenia interfejsów użytkownika
- Ułatwia manipulację DOM

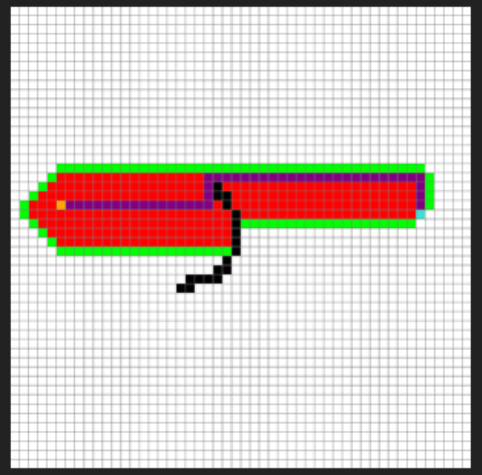


Interfejs oraz funkcje

Interfejs

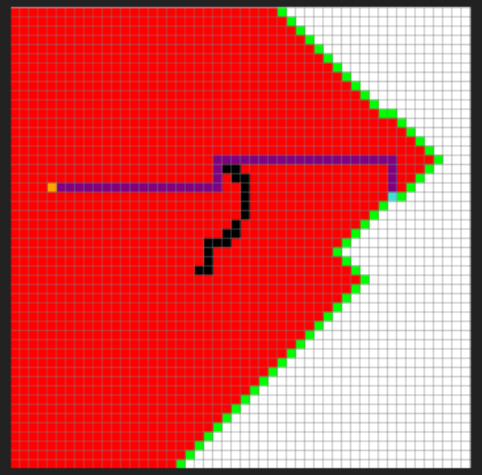
- 4 plansze z siatką
- Przyciski start, clear
- Wyświetlanie czasu wyszukiwania drogi algorytmem

A star



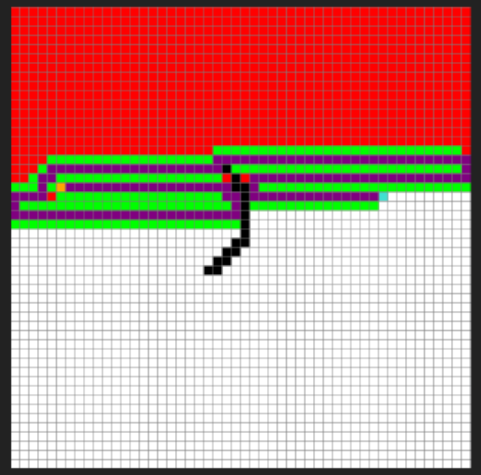
Liczba iteracji: 240
Czas wyszukiwania: 04:35 s

Dijkstra



Liczba iteracji: 211
Czas wyszukiwania: 04:06 s

DFS



Liczba iteracji: 475
Czas wyszukiwania: 10:56 s

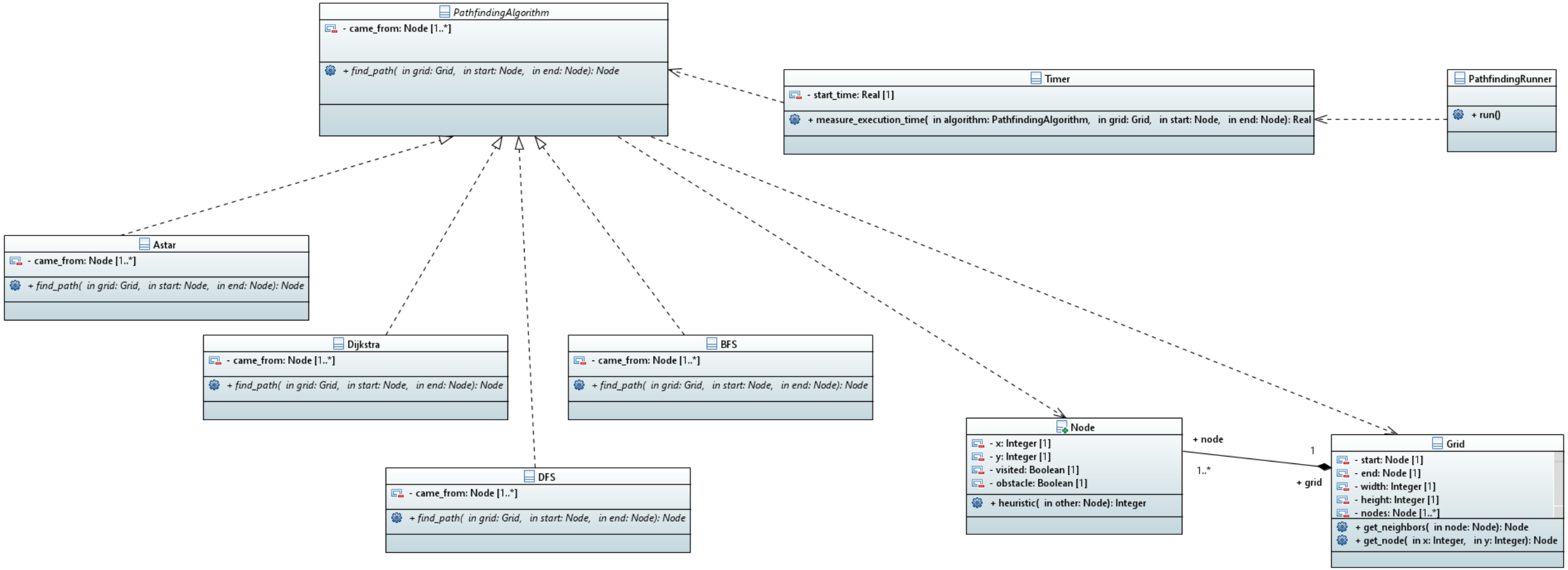
Funkcje

- Rysowanie przeszkód
- Usuwanie przeszkód
- Czyszczenie całej planszy
- Wybieranie wielkości planszy

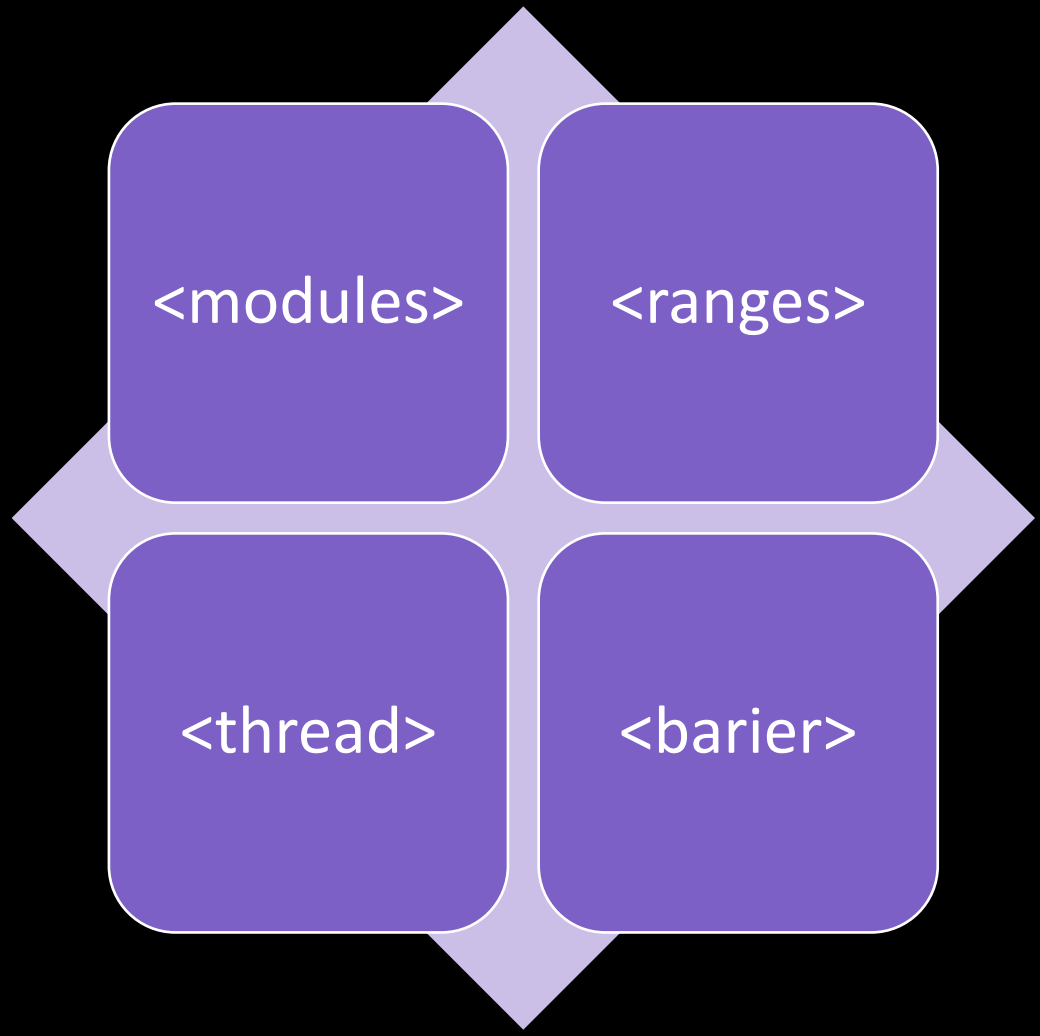
Klasy

- *PathfindingAlgorithm*
  - Astar
  - Dijkstra
  - DFS
  - BFS

- Node
- Grid
- Timer
- PathfindingRunner


PathfindingAlgorithm
- came_from: Node [1..*]
+ find_path(in grid: Grid, in start: Node, in end: Node): Node
Timer
- start_time: Real [1]
+ measure_execution_time(in algorithm: PathfindingAlgorithm, in grid: Grid, in start: Node, in end: Node): Real
PathfindingRunner
+ run()
Astar
- came_from: Node [1..*]
+ find_path(in grid: Grid, in start: Node, in end: Node): Node
Dijkstra
- came_from: Node [1..*]
+ find_path(in grid: Grid, in start: Node, in end: Node): Node
BFS
- came_from: Node [1..*]
+ find_path(in grid: Grid, in start: Node, in end: Node): Node
DFS
- came_from: Node [1..*]
+ find_path(in grid: Grid, in start: Node, in end: Node): Node
Node
- x: Integer [1]
- y: Integer [1]
- visited: Boolean [1]
- obstacle: Boolean [1]
+ heuristic(in other: Node): Integer
+ node
1..*
1
+ grid
Grid
- start: Node [1]
- end: Node [1]
- width: Integer [1]
- height: Integer [1]
- nodes: Node [1..*]
+ get_neighbors(in node: Node): Node
+ get_node(in x: Integer, in y: Integer): Node

Wykorzystane tematy laboratoryjne



Implementacja

```cpp
export class AStar : public PathfindingAlgorithm {
private:
    // Keep track of the parent node for each node
    std::unordered_map<Node*, Node*> came_from;

public:
    // Find a path from start to end in the given grid
    std::vector<Node*> find_path(Grid& grid, Node* start, Node* end) {
        std::vector<Node*> path;

        // Initialize the open and closed sets
        std::vector<Node*> open_set;
        open_set.push_back(start);

        std::vector<Node*> closed_set;

        // Keep track of the cost of getting to each node
        std::unordered_map<Node*, int> g_score;
        g_score[start] = 0;

        // Keep track of the estimated cost of getting from each node to the end
        std::unordered_map<Node*, int> f_score;
        f_score[start] = start->heuristic(end);

        // Loop until the open set is empty (i.e., we've searched all reachable nodes)
        while (!open_set.empty()) {
            // Find the node in the open set with the lowest f-score
            auto current = *std::min_element(open_set.begin(), open_set.end(), [&](Node* a, Node* b) {
                return f_score[a] < f_score[b];
                });

            // If we've reached the goal, we're done
            if (current == end) {
                // Reconstruct the path from the end node to the start node
                path.push_back(current);
                while (current != start) {
                    current = came_from[current];
                    path.push_back(current);
                }
                std::reverse(path.begin(), path.end());
                break;
            }

            // Remove the current node from the open set
            open_set.erase(std::remove(open_set.begin(), open_set.end(), current), open_set.end());

            // Add the current node to the closed set
            closed_set.push_back(current);

            // Explore the neighbors of the current node
            for (auto neighbor : grid.get_neighbors(current)) {
                // If the neighbor is already in the closed set, skip it
                if (std::find(closed_set.begin(), closed_set.end(), neighbor) != closed_set.end()) {
                    continue;
                }

                // Compute the tentative g-score for this neighbor
                int tentative_g_score = g_score[current] + 1;

                // If the neighbor is not in the open set, add it
                if (std::find(open_set.begin(), open_set.end(), neighbor) == open_set.end()) {
                    open_set.push_back(neighbor);
                }
                // If the tentative g-score is higher than the current g-score for the neighbor, skip it
                else if (tentative_g_score >= g_score[neighbor]) {
                    continue;
                }

                // We've found a better path to the neighbor, so update its g-score and f-score
                came_from[neighbor] = current;
                g_score[neighbor] = tentative_g_score;
                f_score[neighbor] = tentative_g_score + neighbor->heuristic(end);
            }
        }

        return path;
    }
};
```

```cpp
export class Node {
public:
    int x;
    int y;
    bool visited;
    bool obstacle;

    Node() : x(0), y(0), visited(false), obstacle(false) {}

    Node(int x, int y) : x(x), y(y), visited(false), obstacle(false) {}

    // A* search heuristic function
    int heuristic(Node* other) {
        int dx = std::abs(x - other->x);
        int dy = std::abs(y - other->y);
        return (dx + dy);
    }

    void make_obstacle() {
        obstacle = true;
    }

    bool operator==(const Node* other){
        return (x == other->x) && (y == other->y);
    }
};

export class Grid {
private:
    Node* start;
    Node* end;
    int width;
    int height;
    std::vector<Node> nodes;

public:
    Grid(int w, int h) : width(w), height(h), nodes(w* h) {
        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                nodes[x * height + y] = Node(x, y);
            }
        }
    }

    Node* get_node(int x, int y) {
        if (x < 0 || x >= width || y < 0 || y >= height) {
            return nullptr;
        }
        return &nodes[x * height + y];
    }

    std::vector<Node*> get_neighbors(Node* node) {
        std::vector<Node*> neighbors;
        int x = node->x;
        int y = node->y;

        // Check 4 neighboring nodes
        for (int i = -1; i <= 1; i += 2) {
            int nx = x + i;
            int ny = y;
            Node* neighbor = get_node(nx, ny);
            if (neighbor != nullptr) {
                neighbors.push_back(neighbor);
            }
        }
        for (int j = -1; j <= 1; j += 2) {
            int nx = x;
            int ny = y + j;
            Node* neighbor = get_node(nx, ny);
            if (neighbor != nullptr) {
                neighbors.push_back(neighbor);
            }
        }
        return neighbors;
    }

    double get_edge_cost(Node* node1, Node* node2) {
        // Compute the Euclidean distance between the nodes
        double dx = node2->x - node1->x;
        double dy = node2->y - node1->y;
        double distance = sqrt(dx * dx + dy * dy);

        // Return the distance as the edge cost
        return distance;
    }
};
```

```cpp
export class Timer {
private:
    std::chrono::high_resolution_clock::time_point start_time_;

public:
    double measure_execution_time(PathfindingAlgorithm& algorithm, Grid& grid, Node* start, Node* end) {
        start_time_ = std::chrono::high_resolution_clock::now();

        std::vector<Node*> path = algorithm.find_path(grid, start, end);

        auto end_time = std::chrono::high_resolution_clock::now();
        auto elapsed_time = std::chrono::duration_cast<std::chrono::microseconds>(end_time - start_time_);

        std::cout << "Path found in " << elapsed_time.count() / 1000000.0 << " seconds" << std::endl;

        double measured_time = elapsed_time.count() / 1000000.0;

        return measured_time;
    }
};

export class PathfindingRunner {
public:
    void run() {
        // Set up the grid and start/end nodes
        Grid grid(10, 10);
        Node* start = grid.get_node(0, 0);
        Node* end = grid.get_node(9, 9);

        // Create instances of the four pathfinding algorithms
        AStar a_star;
        Dijkstra dijkstra;
        BFS bfs;
        DFS dfs;

        // Create a vector to hold the results
        std::vector<double> results;

        // Create four threads, one for each algorithm
        std::vector<std::thread> threads;
        threads.push_back(std::thread([&]() {
            Timer timer;
            double elapsed_time = timer.measure_execution_time(a_star, grid, start, end);
            results.push_back(elapsed_time);
            }));
        threads.push_back(std::thread([&]() {
            Timer timer;
            double elapsed_time = timer.measure_execution_time(dijkstra, grid, start, end);
            results.push_back(elapsed_time);
            }));
        threads.push_back(std::thread([&]() {
            Timer timer;
            double elapsed_time = timer.measure_execution_time(bfs, grid, start, end);
            results.push_back(elapsed_time);
            }));
        threads.push_back(std::thread([&]() {
            Timer timer;
            double elapsed_time = timer.measure_execution_time(dfs, grid, start, end);
            results.push_back(elapsed_time);
            }));

        // Wait for all threads to complete
        for (auto& thread : threads) {
            thread.join();
        }

        // Print the results
        std::cout << "A* time: " << results[0] << " seconds" << std::endl;
        std::cout << "Dijkstra time: " << results[1] << " seconds" << std::endl;
        std::cout << "BFS time: " << results[2] << " seconds" << std::endl;
        std::cout << "DFS time: " << results[3] << " seconds" << std::endl;
    }
};
```

Dziękuję za uwagę

Bibliografia

- https://webassembly.org/
- https://isocpp.org/
- https://pl.legacy.reactjs.org/
- https://getbootstrap.com/