# ТЕКСТ ПРОГРАМИ

using System;

using CellWorld.Automaton;

namespace CellWorld.Moore

{

public class Area<T>

{

public T[] Neighbors { get; }

public Area(T[] neighbors)

{

if (neighbors.Length != StaticData.MoorAreaSize)

{

throw new ArgumentOutOfRangeException(nameof(neighbors), "Wrong number of neighbors (should be 9)");

}

Neighbors = neighbors;

}

public T this[int idx]

{

get => Neighbors[idx];

set => Neighbors[idx] = value;

}

}

}

namespace CellWorld.Moore

{

public class BoolArea : Area<bool>

{

public BoolArea(bool[] neighbors) : base(neighbors)

{

}

}

}

using System.Collections.Generic;

using System.Linq;

using CellWorld.Automaton;

namespace CellWorld.Moore

{

/// <summary>

/// Represents neighborhood with its states.

/// When used as condition for rule, state -1 equals 'any' (<see cref="StaticData.AnyState"/>).

/// </summary>

public class CellStateArea : Area<sbyte>

{

public CellStateArea(IEnumerable<sbyte> neighbors) : base(neighbors.ToArray())

{

}

}

}

using CellWorld.Moore.Conditions;

namespace CellWorld.Moore

{

public class MooreRule

{

public IMooreCondition Condition { get; set; }

public sbyte Result { get; set; }

public MooreRule(IMooreCondition condition, sbyte result)

{

Condition = condition;

Result = result;

}

/// <summary>

/// Returns <c>Result</c> if <c>Condition</c> is applicable to <param name="cellNeighbors"></param>.

/// If isn't applicable, returns <c>null</c>.

/// </summary>

public sbyte? TryApply(CellStateArea cellNeighbors)

{

return Condition.IsApplicable(cellNeighbors)

? Result

: (sbyte?) null;

}

}

}

using System.Text.Json;

using CellWorld.Automaton;

using CellWorld.Models;

namespace CellWorld.Moore.Conditions

{

internal class DirectCondition : IMooreCondition

{

public CellStateArea ConditionArea { get; }

public DirectCondition(CellStateArea conditionArea)

{

ConditionArea = conditionArea;

}

public DirectCondition(object model)

{

var directModel = JsonSerializer.Deserialize<DirectConditionModel>(model.ToString());

ConditionArea = new CellStateArea(directModel.Condition);

}

public bool IsApplicable(CellStateArea cellNeighbors)

{

for (var i = 0; i < StaticData.MoorAreaSize; i++)

{

if (cellNeighbors[i] != ConditionArea[i]

&& ConditionArea[i] != StaticData.AnyState)

return false;

}

return true;

}

}

}

using System.Text.Json;

using CellWorld.Automaton;

using CellWorld.Models;

namespace CellWorld.Moore.Conditions

{

/// <summary>

/// When applied, this rule will sum cells from cell's neighborhood that are in same position as "true" in Bool.

/// If this sum is equal to required, will return Result state to the cell.

/// </summary>

internal class SumCondition : IMooreCondition

{

public BoolArea CellsToSum { get; }

public int RequiredSum { get; }

public SumCondition(BoolArea cellsToSum, int requiredSum)

{

CellsToSum = cellsToSum;

RequiredSum = requiredSum;

}

public SumCondition(object model)

{

var sumModel = JsonSerializer.Deserialize<SumConditionModel>(model.ToString());

CellsToSum = new BoolArea(sumModel.CellsToSum);

RequiredSum = sumModel.RequiredSum;

}

public bool IsApplicable(CellStateArea cellNeighbors)

{

var sum = 0;

for (var i = 0; i < StaticData.MoorAreaSize; i++)

{

if (CellsToSum[i])

{

sum += cellNeighbors[i];

}

}

return sum == RequiredSum;

}

}

}

using System.Text.Json;

using CellWorld.Models;

namespace CellWorld.Moore.Conditions

{

/// <summary>

/// Combines two conditions with logical operator

/// </summary>

internal class ComplexCondition : IMooreCondition

{

public IMooreCondition LeftCondition { get; }

public IMooreCondition RightCondition { get; }

public string Operator { get; }

public ComplexCondition(IMooreCondition leftCondition, IMooreCondition rightCondition, string op)

{

LeftCondition = leftCondition;

RightCondition = rightCondition;

Operator = op.ToLower();

}

public ComplexCondition(object model)

{

var complexModel = JsonSerializer.Deserialize<ComplexConditionModel>(model.ToString());

LeftCondition = RuleHelper.GetCondition(complexModel.LeftCondition);

RightCondition = RuleHelper.GetCondition(complexModel.RightCondition);

Operator = complexModel.Operator.ToLower();

}

public bool IsApplicable(CellStateArea cellNeighbors)

{

return Operator switch

{

"not" => !RightCondition.IsApplicable(cellNeighbors),

"and" => LeftCondition.IsApplicable(cellNeighbors) && RightCondition.IsApplicable(cellNeighbors),

"or" => LeftCondition.IsApplicable(cellNeighbors) || RightCondition.IsApplicable(cellNeighbors),

"xor" => LeftCondition.IsApplicable(cellNeighbors) ^ RightCondition.IsApplicable(cellNeighbors),

\_ => false

};

}

}

}

using System;

using CellWorld.Automaton;

namespace CellWorld.Margolus

{

public class Block: IEquatable<Block>

{

public sbyte[] Cells;

public Block(sbyte[] cells)

{

if (cells.Length != StaticData.MargolusAreaSize)

{

throw new ArgumentOutOfRangeException(nameof(cells), "Wrong size of block");

}

Cells = cells;

}

/// <summary>

/// 0 1

/// 2 3

/// </summary>

public sbyte this[int idx]

{

get => Cells[idx];

set => Cells[idx] = value;

}

public bool Equals(Block other)

{

if (ReferenceEquals(null, other)) return false;

if (ReferenceEquals(this, other)) return true;

for (var i = 0; i < StaticData.MargolusAreaSize; i++)

{

if (Cells[i] != other.Cells[i])

return false;

}

return true;

}

public override bool Equals(object obj)

{

if (ReferenceEquals(null, obj)) return false;

if (ReferenceEquals(this, obj)) return true;

if (obj.GetType() != this.GetType()) return false;

return Equals((Block) obj);

}

public override int GetHashCode()

{

return (Cells != null ? Cells.GetHashCode() : 0);

}

}

}

using System.Text.Json;

using CellWorld.Models;

namespace CellWorld.Margolus

{

public class BlockRule

{

/// <summary>

/// 0 or 1 for phase-dependent rules, -1 for phase-independent

/// </summary>

public sbyte Phase { get; set; }

public Block From { get; set; }

public Block To { get; set; }

public BlockRule(sbyte phase, Block from, Block to)

{

Phase = phase;

From = from;

To = to;

}

public BlockRule(BlockRuleModel model)

{

Phase = model.Phase;

From = new Block(model.FromBlock);

To = new Block(model.ToBlock);

}

}

}

using System.Collections.Generic;

using CellWorld.Models;

namespace CellWorld.Moore

{

public class MooreStartConditions

{

public sbyte[][] Matrix { get; set; }

public IEnumerable<MooreRuleModel> RuleSet { get; set; }

public int Steps { get; set; }

public sbyte DefaultValue { get; set; }

}

}

using System.Collections.Generic;

using CellWorld.Models;

namespace CellWorld.Margolus

{

public class BlockStartConditions

{

public sbyte[][] Matrix { get; set; }

public IEnumerable<BlockRuleModel> RuleSet { get; set; }

public int Steps { get; set; }

}

}

namespace CellWorld.Models

{

public class BlockRuleModel

{

public sbyte Phase { get; set; }

public sbyte[] FromBlock { get; set; }

public sbyte[] ToBlock { get; set; }

public BlockRuleModel() { }

}

}

namespace CellWorld.Models

{

public class MooreRuleModel

{

public MooreRuleModel(){}

public ConditionModel ConditionModel { get; set; }

public sbyte Result { get; set; }

}

}

namespace CellWorld.Models

{

public class DirectConditionModel

{

public sbyte[] Condition { get; set; }

public DirectConditionModel(){}

}

}

namespace CellWorld.Models

{

public class SumConditionModel

{

public bool[] CellsToSum { get; set; }

public int RequiredSum { get; set; }

public SumConditionModel(){}

}

}

namespace CellWorld.Models

{

public class ComplexConditionModel

{

public ConditionModel RightCondition { get; set; }

public ConditionModel LeftCondition { get; set; }

public string Operator { get; set; }

public ComplexConditionModel(){}

}

}

namespace CellWorld.Models

{

public class ConditionModel

{

public string ConditionType { get; set; }

public object Condition { get; set; }

}

}

using System;

using CellWorld.Moore;

using CellWorld.Moore.Conditions;

namespace CellWorld.Models

{

internal static class RuleHelper

{

public static MooreRule GetMooreRule(MooreRuleModel model)

{

var condition = GetCondition(model.ConditionModel);

return new MooreRule(condition, model.Result);

}

public static IMooreCondition GetCondition(ConditionModel model)

{

return model.ConditionType.ToLower() switch

{

"direct" => new DirectCondition(model.Condition),

"sum" => new SumCondition(model.Condition),

"complex" => new ComplexCondition(model.Condition),

\_ => throw new ArgumentException($"Unknown rule type {model.ConditionType}")

};

}

}

}

using System;

using System.Linq;

using CellWorld.Margolus;

using CellWorld.Moore;

namespace CellWorld.Automaton

{

public class Matrix

{

public sbyte[][] M { get; }

public int Height { get; }

public int Width { get; }

public Matrix(sbyte[][] m)

{

M = m;

Height = m.Length;

Width = m[0].Length;

if (m.Any(row => row.Length != Width))

{

throw new ArgumentException("All rows in matrix doesn't have same length");

}

}

public Matrix(int height, int width)

{

M = new sbyte[height][];

for (var i = 0; i < height; i++)

{

M[i] = new sbyte[width];

}

Height = height;

Width = width;

}

public sbyte this[int i, int j]

{

get => M[GetCycledI(i)][GetCycledJ(j)];

set => M[GetCycledI(i)][GetCycledJ(j)] = value;

}

/// <summary>

/// Previous cell value (X), then clockwise from N:

/// X N NE E SE S SW W NW

/// </summary>

public CellStateArea GetMooreNeighborhood(int i, int j)

{

var x = this[i, j];

var n = this[i - 1, j];

var ne = this[i - 1, j + 1];

var e = this[i, j + 1];

var se = this[i + 1, j + 1];

var s = this[i + 1, j];

var sw = this[i + 1, j - 1];

var w = this[i, j - 1];

var nw = this[i - 1, j - 1];

return new CellStateArea(new[] { ne, n, nw, e, x, w, se, s, sw});

}

public Block GetMargolusBlock((int, int) blockCorner)

{

var (i, j) = blockCorner;

var ul = this[i, j];

var ur = this[i, j + 1];

var dr = this[i + 1, j + 1];

var dl = this[i + 1, j];

return new Block(new[] { ul, ur, dl, dr});

}

public void UpdateMargolusBlock((int i, int j) blockCorner, Block blockToInsert)

{

var (i, j) = blockCorner;

this[i, j] = blockToInsert[0];

this[i, j + 1] = blockToInsert[1];

this[i + 1, j] = blockToInsert[2];

this[i + 1, j + 1] = blockToInsert[3];

}

private int GetCycledI(int i)

{

if (i == -1)

return Height - 1;

if (i == Height)

return 0;

return i;

}

private int GetCycledJ(int j)

{

if (j == -1)

return Width - 1;

if (j == Width)

return 0;

return j;

}

}

}

namespace CellWorld.Automaton

{

internal static class StaticData

{

public const int MoorAreaSize = 9;

public const sbyte AnyState = -1;

public const int MargolusAreaSize = 4;

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using CellWorld.Margolus;

using CellWorld.Moore;

namespace CellWorld.Automaton

{

internal class CaHelper

{

public static sbyte[][][] SimulateMoore(Matrix start, IEnumerable<MooreRule> ruleSet, int steps, sbyte defaultValue)

{

var h = start.Height;

var w = start.Width;

var simulation = new List<Matrix> { start };

for (var step = 0; step < steps; step++)

{

var prev = simulation[step];

var next = new Matrix(h, w);

var iRange = Enumerable.Range(0, h);

var jRange = Enumerable.Range(0, w);

var points = iRange.SelectMany(i => jRange.Select(j => (i, j)));

Parallel.ForEach(points, point =>

{

var (i, j) = point;

var neighbors = prev.GetMooreNeighborhood(i, j);

var newValue = ApplyMooreRules(neighbors, ruleSet);

if (newValue != null)

{

next[i, j] = newValue.Value;

}

else if (defaultValue != -1)

{

next[i, j] = defaultValue;

}

});

simulation.Add(next);

}

return simulation.Select(m => m.M).ToArray();

}

public static sbyte[][][] SimulateBlock(Matrix start, IEnumerable<BlockRule> ruleSet, int steps)

{

var h = start.Height;

var w = start.Width;

var simulation = new List<Matrix> { start };

for (var step = 0; step < steps; step++)

{

var prev = simulation[step];

var next = new Matrix(h, w);

var phase = step % 2;

var iRange = Enumerable.Range(0, h).Where(i => i % 2 == phase);

var jRange = Enumerable.Range(0, w).Where(j => j % 2 == phase);

var blocksCorners = iRange.SelectMany(i => jRange.Select(j => (i, j)));

Parallel.ForEach(blocksCorners, blockCorner =>

{

var prevBlock = prev.GetMargolusBlock(blockCorner);

var resultBlock = prevBlock;

foreach (var blockRule in ruleSet.Where(r => r.Phase == phase || r.Phase == -1))

{

if (blockRule.From.Equals(prevBlock))

{

resultBlock = blockRule.To;

break;

}

}

next.UpdateMargolusBlock(blockCorner, resultBlock);

});

simulation.Add(next);

}

return simulation.Select(m => m.M).ToArray();

}

private static sbyte? ApplyMooreRules(CellStateArea neighbors, IEnumerable<MooreRule> rules)

{

foreach (var rule in rules)

{

var ruleResult = rule.TryApply(neighbors);

if (ruleResult != null)

{

return ruleResult.Value;

}

}

return null;

}

}

}

using System.Linq;

using Microsoft.AspNetCore.Mvc;

using CellWorld.Automaton;

using CellWorld.Margolus;

using CellWorld.Models;

using CellWorld.Moore;

namespace CellWorld.Controllers

{

[Route("api/")]

[ApiController]

public class ApiController : ControllerBase

{

[HttpPost]

[Route("simulateMoore")]

public sbyte[][][] SimulateMoore([FromBody] MooreStartConditions mooreStartConditions)

{

var startMatrix = new Matrix(mooreStartConditions.Matrix);

var rules = mooreStartConditions.RuleSet.Select(RuleHelper.GetMooreRule).ToList();

var res = CaHelper.SimulateMoore(startMatrix, rules, mooreStartConditions.Steps, mooreStartConditions.DefaultValue);

return res;

}

[HttpPost]

[Route("simulateBlock")]

public sbyte[][][] SimulateBlock([FromBody] BlockStartConditions blockStartConditions)

{

var startMatrix = new Matrix(blockStartConditions.Matrix);

var rules = blockStartConditions.RuleSet.Select(m => new BlockRule(m)).ToList();

var res = CaHelper.SimulateBlock(startMatrix, rules, blockStartConditions.Steps);

return res;

}

}

}

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Hosting;

namespace CellWorld

{

public class Program

{

public static void Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

}

}

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using Microsoft.AspNetCore.SpaServices.AngularCli;

namespace CellWorld

{

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

services.AddSpaStaticFiles(configuration =>

{

configuration.RootPath = "client/dist";

});

services.AddSwaggerGen();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

app.UseSwagger();

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");

});

app.UseStaticFiles();

if (!env.IsDevelopment())

{

app.UseSpaStaticFiles();

}

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

app.UseSpa(spa =>

{

spa.Options.SourcePath = "client";

if (env.IsDevelopment())

{

spa.UseAngularCliServer(npmScript: "start");

}

});

}

}

}

import { BlockRulesSet } from "./../rules/block-rule/block-rules-set";

import { RulesService } from "./../services/rules.service";

import { Component, OnInit } from "@angular/core";

import { SimulationType } from "../constants/simulation-type";

import { MooreRulesSet } from "../rules/moore-rule/moore-rules-set";

@Component({

selector: "rules-editor",

templateUrl: "./rules-editor.component.html",

styleUrls: ["./rules-editor.component.css"],

})

export class RulesEditorComponent implements OnInit {

public ruleNames: string[];

public currentRuleName: string;

public currentMooreRule: MooreRulesSet;

public currentBlockRule: BlockRulesSet;

public isMooreMode: boolean;

constructor(private rulesService: RulesService) {

this.ruleNames = rulesService.getRuleSetsNames();

this.selectRule(this.ruleNames[1]);

}

ngOnInit(): void {}

public selectRule(ruleName: string) {

this.currentRuleName = ruleName;

const ruleType = this.rulesService.getRuleSetType(ruleName);

if (ruleType == SimulationType.Moore) {

this.currentMooreRule = this.rulesService.getMooreRulesSet(ruleName);

this.currentBlockRule = null;

this.isMooreMode = true;

} else {

this.currentBlockRule = this.rulesService.getBlockRulesSet(ruleName);

this.currentMooreRule = null;

this.isMooreMode = false;

}

}

public saveBlockRule() {

this.rulesService.setBlockRuleSet(this.currentRuleName, this.currentBlockRule);

}

}

import { MooreRuleModel } from './../../rules/moore-rule/moore-rule-model';

import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

import { MooreRulesSet } from 'src/app/rules/moore-rule/moore-rules-set';

import { ColorMap } from 'src/app/colors/color-map';

@Component({

selector: 'moore-editor',

templateUrl: './moore-editor.component.html',

styleUrls: ['./moore-editor.component.css']

})

export class MooreEditorComponent implements OnInit {

@Input() RulesSet: MooreRulesSet;

@Output() saveRule = new EventEmitter<boolean>();

public ColorMap: ColorMap;

public RuleModels: MooreRuleModel[];

constructor() { }

ngOnInit(): void {

this.ColorMap = this.RulesSet.ColorMap;

this.RuleModels = this.RulesSet.Rules;

}

public changeState() {

this.ColorMap.toggleState();

}

public save(){

this.saveRule.emit(true);

}

public updateRule(idx: number, rule: MooreRuleModel){

this.RuleModels[idx] = rule;

}

public deleteRule(idx: number){

this.RuleModels.splice(idx, 1);

}

}

import { ColorMap } from './../../colors/color-map';

import { BlockRuleModel } from './../../rules/block-rule/block-rule-model';

import { Component, Input, OnInit, Output, EventEmitter } from '@angular/core';

import { BlockRulesSet } from 'src/app/rules/block-rule/block-rules-set';

@Component({

selector: 'block-editor',

templateUrl: './block-editor.component.html',

styleUrls: ['./block-editor.component.css']

})

export class BlockEditorComponent implements OnInit {

@Input() rulesSet: BlockRulesSet;

@Output() saveRule = new EventEmitter<boolean>();

public ColorMap: ColorMap;

public RuleModels: BlockRuleModel[];

constructor() { }

ngOnInit(): void {

this.ColorMap = this.rulesSet.ColorMap;

this.RuleModels = this.rulesSet.Rules;

}

public changeState() {

this.ColorMap.toggleState();

}

public save(){

this.saveRule.emit(true);

}

public updateRule(idx: number, rule: BlockRuleModel){

this.RuleModels[idx] = rule;

}

public deleteRule(idx: number){

this.RuleModels.splice(idx, 1);

}

public addTransition(){

this.RuleModels.push(new BlockRuleModel())

}

}

export class BlockRuleModel {

public Phase: number;

public FromBlock: Array<number>;

public ToBlock: Array<number>;

constructor();

constructor(phase: number, fromBlock: Array<number>, toBlock: Array<number>);

constructor(phase?: number, fromBlock?: Array<number>, toBlock?: Array<number>) {

this.Phase = phase || 0;

this.FromBlock = fromBlock || [0, 0, 0, 0];

this.ToBlock = toBlock || [0, 0, 0, 0];

}

public getFrom(i: number, j: number): number {

return this.FromBlock[i \* 2 + j];

}

public getTo(i: number, j: number): number {

return this.ToBlock[i \* 2 + j];

}

public setFrom(i: number, j: number, val: number) {

this.FromBlock[i \* 2 + j] = val;

}

public setTo(i: number, j: number, val: number) {

this.ToBlock[i \* 2 + j] = val;

}

}

import { BlockRuleModel } from './block-rule-model';

import { Component, Input, OnInit, Output, EventEmitter } from '@angular/core';

import { ColorMap } from 'src/app/colors/color-map';

@Component({

selector: 'block-rule',

templateUrl: './block-rule.component.html',

styleUrls: ['./block-rule.component.css']

})

export class BlockRuleComponent implements OnInit {

@Input() rule: BlockRuleModel;

@Input() colorMap: ColorMap;

@Output() ruleOutput = new EventEmitter<BlockRuleModel>();

@Output() toDelete = new EventEmitter<boolean>();

public get FromBlock(): number[][] {

return [

[this.rule.FromBlock[0], this.rule.FromBlock[1]],

[this.rule.FromBlock[2], this.rule.FromBlock[3]]

]

}

public get ToBlock(): number[][] {

return [

[this.rule.ToBlock[0], this.rule.ToBlock[1]],

[this.rule.ToBlock[2], this.rule.ToBlock[3]]

]

}

constructor() {

}

ngOnInit(): void {

}

public onChangeToBlock(i: number, j: number) {

this.rule.setTo(i, j, this.colorMap.currentState);

this.ruleOutput.emit(this.rule)

return false;

}

public onChangeFromBlock(i: number, j: number) {

this.rule.setFrom(i, j, this.colorMap.currentState);

this.ruleOutput.emit(this.rule)

return false;

}

public getColor(state: number) {

return this.colorMap.getColor(state)

}

public onDelete(){

this.toDelete.emit(true);

}

}

import { ColorMap } from "./../../colors/color-map";

import { BlockRuleModel } from "./block-rule-model";

export class BlockRulesSet {

constructor(public Rules: BlockRuleModel[], public ColorMap: ColorMap) {}

}

import { ConditionModel } from './../condition-model';

export class ComplexConditionModel {

constructor (

public LeftCondition: ConditionModel,

public RightCondition: ConditionModel,

public Operator: string

){}

}

export class DirectConditionModel {

constructor (

public Condition: Array<number>

){}

public get(i: number, j: number): number {

return this.Condition[i \* 3 + j];

}

public set(i: number, j: number, val: number) {

this.Condition[i \* 3 + j] = val;

}

}

export class SumConditionModel {

constructor (

public CellsToSum: Array<boolean>,

public RequiredSum: number

){}

}

import { MooreRuleModel } from './../moore-rule-model';

import { DirectConditionModel } from './../condition-models/direct-condition-model';

import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

import { ColorMap } from 'src/app/colors/color-map';

@Component({

selector: 'direct-rule',

templateUrl: './direct-rule.component.html',

styleUrls: ['./direct-rule.component.css']

})

export class DirectRuleComponent implements OnInit {

@Input() conditionModel: DirectConditionModel;

@Input() result: number;

@Input() ColorMap: ColorMap;

@Output() deleteModel = new EventEmitter<boolean>();

public get Condition(): number[][] {

return [

[this.conditionModel.Condition[0], this.conditionModel.Condition[1], this.conditionModel.Condition[2]],

[this.conditionModel.Condition[3], this.conditionModel.Condition[4], this.conditionModel.Condition[5]],

[this.conditionModel.Condition[6], this.conditionModel.Condition[7], this.conditionModel.Condition[8]],

]

}

constructor() { }

ngOnInit(): void {

}

public onChangeCondition(i: number, j: number) {

this.conditionModel.set(i, j, this.ColorMap.currentState);

return false;

}

public getColor(state: number) {

return this.ColorMap.getColor(state)

}

public onDelete(){

this.deleteModel.emit(true);

}

}

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'sum-rule',

templateUrl: './sum-rule.component.html',

styleUrls: ['./sum-rule.component.css']

})

export class SumRuleComponent implements OnInit {

constructor() { }

ngOnInit(): void {

}

}

export class ConditionModel {

constructor(

public ConditionType: string,

public Condition: any

){}

}

import { DirectConditionModel } from './condition-models/direct-condition-model';

import { ConditionModel } from './condition-model';

export class MooreRuleModel{

constructor(

public ConditionModel: ConditionModel,

public Result: number

){}

public isDirect(): boolean {

return this.ConditionModel.ConditionType == "direct";

}

public isSum(): boolean {

return this.ConditionModel.ConditionType == "sum";

}

}

import { MooreRuleModel } from "./moore-rule-model";

import { ColorMap } from "./../../colors/color-map";

export class MooreRulesSet {

constructor(public Rules: MooreRuleModel[], public ColorMap: ColorMap) {}

}

import { RulesService } from './rules.service';

import { SimulationType } from '../constants/simulation-type';

import { MooreStartConditions, BlockStartConditions } from './start-conditions';

import { Injectable } from '@angular/core';

import { HttpClient} from '@angular/common/http';

@Injectable({

providedIn: 'root',

})

export class DataService {

private url = "/api/";

private savedState: Array<Array<number>> = null;

constructor(private http: HttpClient, private rulesService: RulesService) {}

public fetchSimulationResults(startMatrix: Array<Array<number>>, ruleName: string, steps: number) {

const simType = this.rulesService.getRuleSetType(ruleName);

if (simType == SimulationType.Block) {

const rulesSet = this.rulesService.getBlockRulesSet(ruleName);

return this.http.post(this.url + "simulateBlock", new BlockStartConditions(startMatrix, rulesSet.Rules, steps));

}

const rulesSet = this.rulesService.getMooreRulesSet(ruleName);

return this.http.post(this.url + "simulateMoore", new MooreStartConditions(startMatrix, rulesSet.Rules, steps, 0));

}

public saveState(matrix: Array<Array<number>>){

this.savedState = matrix;

}

public getSavedState() : Array<Array<number>> {

return this.savedState;

}

}

import { BlockRulesSet } from './../rules/block-rule/block-rules-set';

import { SimulationType } from './../constants/simulation-type';

import { ConstantRules } from './../constants/constant-rules';

import { Injectable } from '@angular/core';

import { MooreRulesSet } from '../rules/moore-rule/moore-rules-set';

@Injectable({

providedIn: 'root'

})

export class RulesService {

private mooreRuleSets: Map<string, MooreRulesSet>;

private blockRuleSets: Map<string, BlockRulesSet>;

constructor() {

this.mooreRuleSets = ConstantRules.MooreRules;

this.blockRuleSets = ConstantRules.BlockRules;

}

public getRuleSetsNames(): string[] {

return Array.from(this.mooreRuleSets.keys()).concat(Array.from(this.blockRuleSets.keys()))

}

public getRuleSetType(ruleSetName: string): SimulationType {

if (this.blockRuleSets.has(ruleSetName)) {

return SimulationType.Block;

}

return SimulationType.Moore;

}

public getMooreRulesSet(ruleSetName: string): MooreRulesSet{

return this.mooreRuleSets.get(ruleSetName);

}

public getBlockRulesSet(ruleSetName: string): BlockRulesSet{

return this.blockRuleSets.get(ruleSetName);

}

public setBlockRuleSet(ruleSetName: string, rule: BlockRulesSet) {

this.blockRuleSets.set(ruleSetName, rule);

}

public setMooreRuleSet(ruleSetName: string, rule: MooreRulesSet) {

this.mooreRuleSets.set(ruleSetName, rule);

}

}

import { BlockRuleModel } from '../rules/block-rule/block-rule-model';

import { MooreRuleModel } from '../rules/moore-rule/moore-rule-model';

export class MooreStartConditions {

constructor (

public matrix: Array<Array<number>>,

public ruleSet: Array<MooreRuleModel>,

public steps: number,

public defaultValue: number

) {

}

}

export class BlockStartConditions {

constructor (

public matrix: Array<Array<number>>,

public ruleSet: Array<BlockRuleModel>,

public steps: number

) {

}

}

import \* as p5 from "p5";

export class CellGrid extends p5 {

public id: string;

public colorMap = [

'white',

'black'

]

public currentState = 1;

public cellSize = 5;

public gridWidth: number;

public gridHeight: number;

public currentLayer: Array<Array<number>>;

constructor(sketch = p=>{ }) {

super(sketch);

}

setup(){

this.id = "sketch-holder";

this.currentState = 1;

this.cellSize = 5;

this.colorMap = [

'white',

'black'

];

let width = this.windowWidth - 300;

let height = this.windowHeight - 120;

this.gridWidth = Math.floor(width / this.cellSize / 2) \* 2;

this.gridHeight = Math.floor(height / this.cellSize / 2) \* 2;

this.gridWidth = this.gridHeight;

this.currentLayer = this.getEmptyMatrix(this.gridWidth, this.gridHeight);

let canvas = this.createCanvas(this.gridWidth \* this.cellSize, this.gridHeight \* this.cellSize);

canvas.parent(this.id);

}

draw(){

this.drawBackground()

if (!!this.currentLayer) {

for (var i = 0; i < this.gridHeight; i++) {

for (var j = 0; j < this.gridWidth; j++) {

this.drawCell(i, j)

}

}

}

}

mouseClicked(){

if (this.mouseX <= this.width && this.mouseX >= 0 && this.mouseY <= this.height && this.mouseY >= 0) {

this.setCell(this.mouseX, this.mouseY)

}

return false;

}

mouseDragged(){

if (this.mouseX <= this.width && this.mouseX >= 0 && this.mouseY <= this.height && this.mouseY >= 0) {

this.setCell(this.mouseX, this.mouseY)

}

return false;

}

public clearGrid(){

for (var i = 0; i < this.gridHeight; i++) {

for (var j = 0; j < this.gridWidth; j++) {

this.currentLayer[i][j] = 0;

}

}

}

public fillWithStatic(){

for (var i = 0; i < this.gridHeight; i++) {

for (var j = 0; j < this.gridWidth; j++) {

this.currentLayer[i][j] = this.floor(this.random() \* 2);

}

}

}

public getEmptyMatrix(width: number, height: number): Array<Array<number>> {

const res = [];

for (var i = 0; i < height; i++) {

res[i] = [];

for (var j = 0; j < width; j++) {

res[i][j] = 0;

}

}

return res;

}

private drawBackground(){

this.background(255);

this.stroke('black');

this.noFill();

this.stroke('#b3b3b3');

for (var i = 0; i < this.gridHeight; i++) {

this.line(0, i\*this.cellSize, this.gridWidth \* this.cellSize, i\*this.cellSize)

}

for (var j = 0; j < this.gridWidth; j++) {

this.line(j\*this.cellSize, 0, j\*this.cellSize, this.gridHeight \*this.cellSize)

}

}

private setCell(mouseX: number, mouseY: number) {

const x = Math.floor(mouseX / this.cellSize);

const y = Math.floor(mouseY / this.cellSize);

this.currentLayer[y][x] = this.currentState;

}

private drawCell(i: number, j: number){

const val = this.currentLayer[i][j]

if (val != 0) {

const cellColor = this.colorMap[this.currentLayer[i][j]]

this.fill(cellColor)

const x = j \* this.cellSize;

const y = i \* this.cellSize;

this.rect(x, y, this.cellSize, this.cellSize);

}

}

}

import { RulesService } from './../services/rules.service';

import { Component, OnDestroy, OnInit } from '@angular/core';

import { Subject, timer, Observable } from 'rxjs';

import { takeUntil } from 'rxjs/operators';

import { CellGrid } from './cell-grid';

import { DataService } from '../services/data.service';

import { ColorMap } from '../colors/color-map';

@Component({

selector: 'simulation',

templateUrl: './simulation.component.html',

styleUrls: ['./simulation.component.css']

})

export class SimulationComponent implements OnInit, OnDestroy{

public static GridId = "sketch-holder";

public get GridId() {

return SimulationComponent.GridId;

}

public cellGrid: CellGrid;

public needsToStop = new Subject<true>();

public needsToSimulate = new Subject<true>();

public startLayer: Array<Array<number>>;

public isSimulating = false;

public rulesNames: string[];

public ruleToSimulate: string;

simulation: Array<Array<Array<number>>> = [];

speed: number = 40;

stepsPerRequest = 50;

constructor(private dataService: DataService, private rulesService: RulesService) {

this.rulesNames = rulesService.getRuleSetsNames();

this.ruleToSimulate = this.rulesNames[0];

this.needsToSimulate.subscribe(() => {

this.simulate(

this.dataService.fetchSimulationResults(

this.startLayer,

this.ruleToSimulate,

this.stepsPerRequest

)

);

});

}

public ngOnInit() {

if (!this.cellGrid) {

setTimeout(() => {

this.cellGrid = new CellGrid();

const savedState = this.dataService.getSavedState();

if (savedState != null){

this.cellGrid.currentLayer = savedState;

}

}, 10); //to prevent filling before init

}

}

ngOnDestroy(): void {

this.dataService.saveState(this.cellGrid.currentLayer);

this.cellGrid.remove();

}

public changeState() {

this.cellGrid.currentState = this.cellGrid.currentState == 0 ? 1 : 0;

}

public stopSimulation() {

this.isSimulating = false;

this.needsToStop.next(true);

}

public startSimulation() {

this.simulation = [];

this.isSimulating = true;

timer(0, this.speed)

.pipe(takeUntil(this.needsToStop))

.subscribe((x) => {

if (!!this.simulation[0]) {

this.cellGrid.currentLayer = this.simulation.shift();

if (this.simulation.length == 30) {

this.startLayer = this.simulation.pop();

this.needsToSimulate.next(true);

}

}

});

this.startLayer = this.cellGrid.currentLayer;

this.needsToSimulate.next(true);

}

private simulate(result: Observable<Object>) {

result.subscribe((simulation: number[][][]) => {

this.simulation = this.simulation.concat(simulation);

});

}

}

import { RulesEditorComponent } from './rules-editor/rules-editor.component';

import { SimulationComponent } from './simulation/simulation.component';

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [

{ path: 'simulation', component: SimulationComponent },

{ path: 'rules-editor', component: RulesEditorComponent },

{ path: '', redirectTo: '/simulation', pathMatch: 'full' }

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

import { ConstantColorMaps } from './../constants/constant-color-maps';

export class ColorMap{

public statesToColors: Map<number, string>;

public currentState: number;

public get States(): number[] {

return Array.from(this.statesToColors.keys());

}

constructor();

constructor(map: Map<number, string>)

constructor(map?: Map<number, string>){

this.statesToColors = map || ConstantColorMaps.WhiteBlack;

if (this.States.length > 0){

this.currentState = this.States[0]

} else {

this.currentState = 0;

}

}

public getColor(state: number): string {

return this.statesToColors.get(state);

}

public toggleState(){

const idx = this.States.indexOf(this.currentState);

if (idx < this.States.length - 1){

this.currentState = this.States[idx + 1]

} else {

this.currentState = this.States[0]

}

}

}

import { DataService } from "./services/data.service";

import { Component } from "@angular/core";

@Component({

selector: "app",

styleUrls: ["./app.component.css"],

templateUrl: "./app.component.html",

providers: [DataService],

})

export class AppComponent {

}

import { NgModule } from "@angular/core";

import { BrowserModule } from "@angular/platform-browser";

import { FormsModule } from "@angular/forms";

import { AppComponent } from "./app.component";

import { MatButtonModule } from "@angular/material/button";

import { HttpClientModule } from "@angular/common/http";

import { SimulationComponent } from "./simulation/simulation.component";

import { AppRoutingModule } from "./app-routing.module";

import { RulesEditorComponent } from "./rules-editor/rules-editor.component";

import { BlockRuleComponent } from "./rules/block-rule/block-rule.component";

import { BlockEditorComponent } from "./rules-editor/block-editor/block-editor.component";

import { MooreEditorComponent } from "./rules-editor/moore-editor/moore-editor.component";

import { DirectRuleComponent } from "./rules/moore-rule/direct-rule/direct-rule.component";

import { SumRuleComponent } from "./rules/moore-rule/sum-rule/sum-rule.component";

@NgModule({

imports: [

BrowserModule,

FormsModule,

MatButtonModule,

HttpClientModule,

AppRoutingModule,

],

declarations: [

AppComponent,

SimulationComponent,

RulesEditorComponent,

BlockRuleComponent,

BlockEditorComponent,

MooreEditorComponent,

DirectRuleComponent,

SumRuleComponent,

],

bootstrap: [AppComponent],

})

export class AppModule {}

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { enableProdMode } from '@angular/core';

import { AppModule } from './app/app.module';

enableProdMode();

const platform = platformBrowserDynamic();

platform.bootstrapModule(AppModule);