Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Використання клітинних автоматів для контурування зображень

Текстова частина до курсової роботи

за спеціальністю «Інженерія програмного забезпечення»

Керівник курсової роботи

Ас. Калітовський Б. В.

_____

(підпис)

“____” _____ 2021 р.

Виконав студент ІПЗ-4:

Липка Є. Є.

“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних

систем,доцент, к.ф.-м.н.

_____О. П. Жежерун

(підпис)

“____”_____202_ р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Липці Єгору

4-го курсу факультету інформатики

ТЕМА: Використання клітинних автоматів для контурування зображень

Вихідні дані:

Застосунок для створення, застосування й оцінки роботи клітинних автоматів для контурування зображень

Зміст ТЧ до курсової роботи:

Анотація

Вступ

1. Дослідження клітинних автоматів
2. Розробка алгоритму
3. Аналіз результатів та порівняння із класичними методами контурування

Висновки

Список джерел

Додатки

Дата видачі “___”_____201_ р.

Керівник_____ Завдання отримано _____

Календарний план-графік підготовки кваліфікаційної роботи на тему:

Використання клітинних автоматів для контурування зображень

студента 4-го курсу факультету інформаткик
Липки Єгора Євгеновича

(прізвище, ім'я, по-батькові)

Науковий керівник Ас. Калітовський Б. В.

(науковий ступінь, вчене звання прізвище, ім'я, по-батькові)

Види і етапи магістерської роботи	Термін виконання етапів роботи	Примітка
1. Затвердження теми і наукового керівника бакалаврської роботи	18.10.2020 – 20.10.2020	
2. Опрацювання першоджерел із темі роботи	20.10.2020 – 31.12.2020	
3. Розробка плану роботи, методики та інструментарію досліджень	01.01.2021 – 14.01.2021	
4. Проведення досліджень та їх аналіз	15.01.2021 – 31.01.2021	
5. Написання вступу та першого розділу роботи	01.02.2021 – 16.02.2021	
6. Написання другого розділу роботи	17.02.2021 – 03.02.2021	
7. Написання третього розділу роботи	04.02.2021 – 28.02.2021	
8. Написання висновків та пропозицій, оформлення роботи	01.03.2021 – 24.03.2021	
9. Подача першого варіанту роботи на кафедру	25.03.2021	
10. Доопрацювання роботи згідно з зауваженнями керівника	26.03.2021 – 07.04.2021	
11. Попередній захист роботи		

Студент (прізвище, ім'я, по батькові)

(підпис)

Зміст

Анотація

Контурування, знаходження границь на зображенні, є надзвичайно важливим аспектом обробки зображень. У цій роботі розглянуто можливість застосування зовнішніх тоталістичних клітинних автоматів для виконання цієї задачі на бінарних зображеннях. Використовуючи візуальні та кількісні вимірювання подібності, було встановлено найкращі тоталістичні автомати та порівняно результати їхньої роботи з відомими класичними методами контурування, а також надано рекомендації для подальшого дослідження даної теми.

Вступ

Зображення – це двовимірна функція f(x, y), де x та y є координатами елементу зображення, пікселя, у двовимірному просторі, а f – значення інтенсивності кольору або відтінку сірого даного пікселя. Обробка зображень є використанням різноманітних математичних алгоритмів на вхідному зображенні для отримання в результаті нового зображення або інформації про початкове. Вона відіграє значну роль у багатьох сферах життя та технологіях, використовується у медицині та біології, системах безпеки та спостереження, технологіях комп’ютерного зору, обробці космічних фотографій з телескопів, системах навігації та картах і багатьох інших.

Одним же із ключових методів, що використовуються в обробці зображень, є контурування. Воно полягає у використанні математичних алгоритмів для виявлення точок (пікселів) із різкою зміною інтенсивності кольору в порівнянні із сусідніми. Контурування є необхідним і використовується у технологіях машинного та комп’ютерного зору, зокрема для виявлення та виділення ознак.

Постановка задачі

Одними з найвідоміших методів виявлення контурів є алгоритми Кенні [1] та Собеля [2], однак загалом методів існує дуже багато. Усі вони мають свої недоліки та переваги, такі як, наприклад, швидкість виконання, стійкість до шуму та артефактів, товщина контурів. У цій роботі буде розглянуто та досліджено можливість використання клітинних автоматів для контурування зображень. Об’єктом дослідження є клітинні автомати, предметом дослідження є алгоритми контурування. Метою цієї роботи буде створення клітинних автоматів, які можуть ефективно використовуватися для виявлення контурів електронних зображень та порівняння результатів їхньої роботи із результатами інших відомих алгоритмів контурування.

Робота складається з трьох розділів.

Перший розділ присвячено дослідженню та аналізу клітинних автоматів в цілому, а також можливістю їх використання для виконання поставленої задачі.

В другому розділі продемонстровано розробку алгоритму для створення та виявлення шуканих клітинних автоматів. Надано використані методи та показано етапи розробки алгоритму від створення та застосування клітинних автоматів до оцінки їхньої роботи та виокремлення найкращих.

Останній третій етап присвячено аналізу ефективності знайдених клітинних автоматів та порівнянню їх із найвідомішими алгоритмами контурування, такими як оператори Кенні, Собеля, Прюітта.

Розділ 1. Дослідження клітинних автоматів

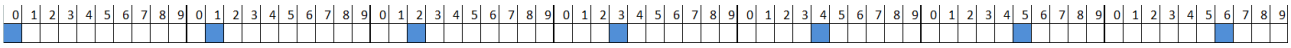
1.1 Клітинні автомати

Клітинні автомати, також відомі як ітеративні масиви, є дискретними математичними моделями, що описуються набором клітин (комірок), які утворюють сітку, та правилами переходу клітини із поточного стану в наступний [3]. Кожний клітинний автомат має сітку, яка може мати будь-яку скінченну кількість вимірів. Сітка складається з клітин, кожна з яких знаходиться в одному зі скінченної кількості визначених станів. Також кожна клітина має сусідів – набір комірок, що розташовуються на певній відстані від даної. У більшості клітинних автоматів лише ці сусіди можуть впливати на стан клітини, з клітинами поза межами цього сусідства вона ніяк не взаємодіє. На початку кожна комірка знаходиться в певному стані, на кожній ітерації створюється нове покоління комірок, застосовуючи правила переходу до кожної з них. При цьому правила застосовуються одночасно, лише після чого створюється нове покоління і стани перезаписуються, таким чином уся сітка оновлюється одночасно. У більшості клітинних автоматів правила є статичними і не змінюються протягом виконання.

Для прикладу розглянемо простий одновимірний клітинний автомат розмірності 10. Набір станів складається лише з двох станів: 0, продемонстрований білим кольором, і 1, продемонстрований синім кольором. Сусідство визначимо з відстанню 1, тобто ті клітини, які розташовуються безпосередньо зліва та справа він поточної. Також встановимо два правила переходу:

1) Клітина в стані 0 переходить у стан 1, якщо має сусіда зліва в стані 1
2) Клітина в стані 1 переходить у стан 0, якщо не має сусіда зліва в стані 1

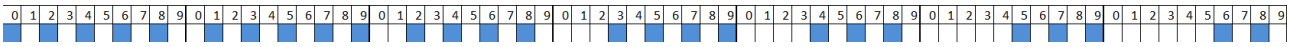
Залишилося лише задати початковий стан сітки. Нехай нульова клітина знаходиться в стані 1, а всі інші – в стані 0. Продемонстровано початковий стан та перші шість ітерацій.



Малюнок 1, Приклад роботи одновимірного клітинного автомату

На першій ітерації перша клітина переходить в стан 1, оскільки до неї застосовується перше правило, а нульова клітина переходить в стан 0, бо до неї підходить правило 2. До решти комірок жодне правило застосувати не можна, тож їхні стани не змінюються. Аналогічно на другій ітерації правило 1 застосовується до другої комірки, правило два – до першої, і так далі. Таким чином наша зафарбована комірка “рухається” зліва направо. На десятій ітерації всі клітини перейдуть в стан 0, і далі жодних змін відбуватися не буде – це значить, що клітинний автомат перейшов у стабільний стан.

В залежності від початкового стану, переданого цьому самому автомату, можна спостерігати дещо відмінний процес. Якщо в початковому стані сітки не тільки нульова, а всі парні клітини знаходяться в стані 1, то матимемо п’ять клітинок, що “рухаються” праворуч на кожній ітерації. Однак результат в даному випадку буде тим самим – на десятій ітерації наш клітинний автомат перейде у стабільний стан.



Малюнок 2, Приклад роботи одновимірного клітинного автомату

Те, що ми щойно розглянули, називається елементарним клітинним автоматом, тобто найпростішим можливим варіантом [3]. Характеризується він тим, що є одновимірним, має всього два можливих стани клітин, і сусідами кожної комірки є лише два її безпосередні сусіди ліворуч та праворуч. Якщо, наприклад, клітина знаходиться в стані 1, клітина ліворуч від неї – в стані 0, а праворуч – в стані 1, то це можна просто записати як "011". Тоді ми можемо перетворити наше перше правило переходу із текстового вигляду на щось, що можна значно простіше змоделювати. Маємо:

1) 100 -> 1
2) 101 -> 1

Однак очевидно, що існує ще два варіанти, коли клітина знаходиться у стані 0, які не були явно вказані в нашому правилі, оскільки вона переходила в той самий стан. Тож повне перше правило можна перекласти на

1) 000 -> 0
2) 001 -> 0
3) 100 -> 1
4) 101 -> 1

Так само друге правило перетворюється на:

1) 010 -> 0
2) 011 -> 0
3) 110 -> 1
4) 111 -> 1

Для спрощення це записують у вигляді таблиці

111	110	101	100	011	010	001	000
1	1	1	1	0	0	0	0

Малюнок 3, Таблиця правил переходу

У випадку елементарних клітинних автоматів, оскільки для кожного правила розглядається три клітини – центральна і два її сусіди, і кожна клітина може бути в одному з двох станів, всього маємо $2^3 = 8$ можливих варіантів сусідства, а отже і правил переходу, як видно з таблиці вище. Знову ж, через те, що існує два стани, в які комірка може перейти за будь-яким із правил, то загалом маємо $2^8 = 256$ можливих наборів правил і елементарних клітинних автоматів. Зазвичай їх називають за допомогою коду Вольфрама запропонованого Стівеном Вольфрамом у 1983 [3]. У наведеній вище таблиці верхній рядок є цілими числами від 7 до 0 у двійковій системі числення впорядкованими зліва направо за спаданням. Якщо ж нижній рядок об’єднати і перевести із двійкової системи числення в десяткову, то це і буде номер елементарного клітинного автомату в коді Вольфрама. Таким чином наш

приклад є правилом 240, адже 11110000 є двійковим представленням числа 240, а всі інші правила можна створити за їхнім порядковим номером: правило 0 – 00000000, правило 43 – 00101011, з тією лише різницею, що насправді в елементарних клітинних автоматах розмір сітки (стрічки в цьому випадку) є нескінченним.

Такі автомати, хоч і називаються елементарними, можуть демонструвати доволі складну поведінку. Так Метью Куком, асистентом Стівена Вольфрама, було доведено його твердження, зроблене у 1985, що правило 110 є повним за Тьюрингом [4], тобто будь-яке обчислення або програма можуть бути симульовані за допомогою цього правила. Втім, це правило є єдиним серед усіх елементарних автоматів, для якого доведено Тьюринг-повноту.

Проте для поставленого завдання нас не цікавлять елементарні клітинні автомати, адже зображення є двовимірними, а не одновимірними. Однак, як було зазначено вище, клітинні автомати можуть мати будь-яку скінченну кількість вимірів.

1.2 Гра “Життя” Джона Конвея

Мабуть, найвідомішим двовимірним клітинним автоматом є Гра Життя Конвея [5], яка симулює справжнє життя. Існує теж лише два стани: клітина може бути або живою, або мертвою, а за сусідство береться окіл Мура з радіусом 1, тобто квадрат розміру 3x3 навколо центральної клітини. Також є чотири правила переходу:

1) Жива клітина, яка має менше двох живих сусідів, помирає
2) Жива клітина, яка має два чи три живих сусіди, продовжує жити
3) Жива клітина, яка має більше трьох живих сусідів, помирає від перенаселення
4) Мертва клітина, яка має три живих сусіди, оживає

Як можна одразу помітити, ці правила дещо відрізняються від правил в елементарних клітинних автоматах. Тут ми не розглядаємо точне розташування

усіх сусідів, а лише їхню загальну кількість в одному зі станів. Звісно, цей клітинний автомат теоретично можна звести до вигляду, який було показано вище, із правилами вигляду "010001010 -> 1" вказуючи, скажімо, клітини в порядку починаючи від лівого верхнього кута і йдучи праворуч і вниз до правого нижнього. Але проблема полягає в тому, що в елементарних клітинних автоматах сусідство визначається трьома комірками з двома можливими станами, тобто $2^3 = 8$ можливих початкових станів, із кожного з яким можна перейти знову в 0 або 1, тобто $2^8 = 256$ можливих наборів правил. Тут же кількість станів лишається тією ж, але сусідство уже визначається дев'ятьма комірками, що дає $2^9 = 512$ початкових станів і 2^{512} можливих наборів правил. Тож щоби задати Гру Життя у вигляді точних правил переходу знадобилося би ввести 512 правил, що, очевидно, не є практичним. Використовуючи ж загальну кількість живих сусідів можна спростити це всього до чотирьох правил. Такі клітинні автомати виділяються окремо і мають два підтипи: тоталістичні – на значення клітини в наступному поколінні впливає загальна кількість сусідів в певному стані, включно із центральною клітиною, і зовнішні тоталістичні – на значення клітини в наступному поколінні впливає її поточне значення і загальна кількість сусідів в певному стані. Оскільки в Грі Життя Конвея розглядається сума восьми сусідів і значення центральної комірки, то цей клітинний автомат належить до зовнішніх тоталістичних.

1.3 Види клітинних автоматів

1.3.1 Тоталістичні клітинні автомати

В тоталістичних клітинних автоматах стан кожної комірки представлений невід'ємним цілим числом. На кожній ітерації для усіх клітин обраховується значення – сума станів усіх сусідів комірки, залежно від того, як саме визначено сусідство, включно з її власним станом. Правила переходу беруть до уваги лише це значення загальної суми при переведенні клітини в наступний стан, тобто дві клітини в різних початкових станах із різною конфігурацією сусідніх комірок можуть перейти в однаковий стан за одним і тим же правилом,

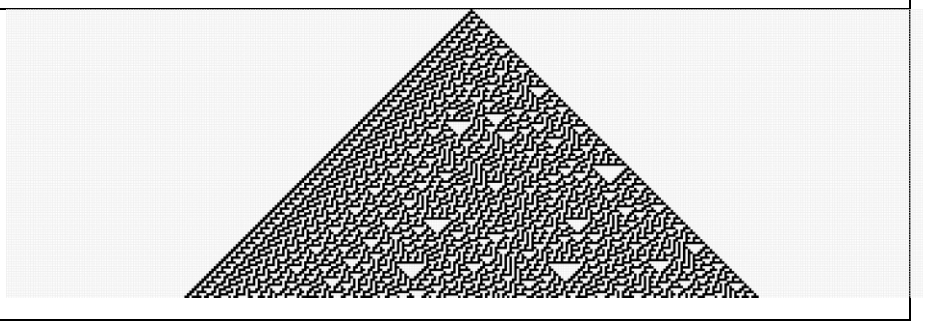
достатньо мати однакову суму сусідів. Ден Гордон у 1987 [6] продемонстрував, що тоталістичні клітинні автомати можуть симулювати роботу будь-якої машини Тьюринга за лінійний час, це стосується навіть простих одновимірних автоматів із сусідством з відстанню в одиницю, тобто до уваги беруться лише три комірки.

1.3.2 Зовнішні тоталістичні клітинні автомати

У випадку ж зовнішніх тоталістичних автоматів правила переходу є функціями, що визначають наступний стан клітини не за одним параметром, а за двома: поточним станом комірки та сумою станів решти комірок у сусідстві. Як було сказано вище, Гра Життя Конвея – це зовнішній тоталістичний автомат. Попри простоту правил і, здавалося б, хаотичність, Гра Життя може демонструвати складність та впорядкованість в своїй поведінці. Наприклад, існують такі конструкції (набори живих клітин), що можуть “рухатися” сіткою в певному напрямку, вони називаються глайдерами. Також існують конструкції, що можуть нескінченно створювати нові глайдери, їх можна розташувати таким чином, що глайдери взаємодіятимуть один із одним. Це і було зроблено Полом Чапманом, який за допомогою цього у Грі Життя створив універсальну машину Тьюринга [7].

1.4 Практичні застосування клітинних автоматів

Клітинні автомати, окрім того, що можуть виступати машинами Тьюринга, можуть виявляти надзвичайно складну поведінку, особливо, при збільшенні кількості станів, мати цілу низку практичних застосувань, а також навіть зустрічатися в природі. Мабуть, одним із найвідоміших прикладів цього є молюск конус текстильний (Conus textile) із сімейства Conidae, візерунок на мушлі якого нагадує вольфрамівське правило 30 [8]. Також у роботі [9] було продемонстровано, що за певних умов рослини у процесі фотосинтезу можуть використовувати системи, поведінка яких дуже схожа на поведінку клітинних автоматів.

Конус Текстильний	Правило 30
	

Малюнок 4, Порівняння мушлі молюска із правилом 30

Реакція Бєлоусова-Жаботинського – це клас хімічних реакцій, які протікають у коливальному режимі, тобто можуть довгий час відбуватися хаотично, дуже повільно наближаючись до термодинамічної рівноваги. Проводячи таку реакцію у чашці Петрі, на поверхні утворюються цятки, які згодом переростають у концентричні кола або спіралі. Дослідження [10] показали, що поведінка цих реакцій може симулюватися за допомогою простих клітинних автоматів.

Найбільше практичних застосувань мають двовимірні клітинні автомати, бо часто використовуються саме для обробки зображень або вилучення інформації з них. Наприклад, можуть застосовуватися для скелетонізації об’єктів на зображеннях. Наскельні малюнки фігур людей, зроблені за допомогою простих ліній, є першими прикладами так званих скелетів об’єктів. Сьогодні ми називаємо їх ендоскелетами, тобто внутрішніми скелетами, вони є базовим представленням форми певного об’єкта. Ця технологія може використовуватися для виявлення ознак зображення або для розпізнавання тексту [11, с. 117].

Малюнок 5, використання скелетонізації для розпізнавання тексту

Також клітинні автомати мають застосування в узгодженому фільтруванні [11, с. 137] – технології в обробці сигналів, за допомогою якої знаходяться подібності між заданим базовим сигналом, шаблоном, і сигналом, що подається на вхід. Це може використовуватися, зокрема, в розпізнаванні образів, виявленні контурів та кутів на зображеннях, в обробці радіосигналів радарів. Клітинні автомати також знаходять використання в сегментації зображень [11, с. 113] – процесі виділення одного об’єкту на зображенні, сегменту, і відкидання решти задля отримання простішого, легшого для подальшого аналізу результуючого зображення.

Цікаві дослідження в сфері криптографії [12] демонструють, що програмовані клітинні автомати, побудовані за допомогою елементарних автоматів, можуть бути використаними для створення блочних шифрів, а високоякісні генератори псевдовипадкових образів, побудовані за допомогою правил 90 та 150, можуть виступати генераторами ключів для потокових шифрів. Говорячи про псевдовипадкові генератори, в роботі [13] показано, яким чином за допомогою програмованого клітинного еволюційного алгоритму створюються двовимірні клітинні автомати, що виступають генераторами випадкових чисел. Обширне тестування показало, що ці генератори можуть швидко створювати якісні послідовності випадкових чисел, при цьому часто перевершуючи раніше створені високоякісні генератори.

Клітинні автомати знаходять використання у відео іграх. Для багатьох ігор процедурна генерація рівнів є важливим аспектом ігрового дизайну. По-перше, це зменшує витрати на створення рівнів вручну і збільшує кількість контенту в грі, адже кожен рівень є унікальним і непередбачуваним. По-друге, для деяких ігор і навіть жанрів ігор основна концепція ігрового дизайну вимагає процедурної генерації рівнів в реальному часі. У роботі 2010 року було продемонстровано, як прості алгоритми, основані на клітинних автоматах, генерують готові до використання, добре продумані ігрові рівні, що складаються з печер та тунелів. Алгоритм має дуже низьку обчислювальну вартість, дозволяючи використовувати його в реальному часі, що у багатьох випадках є дуже важливим[14].

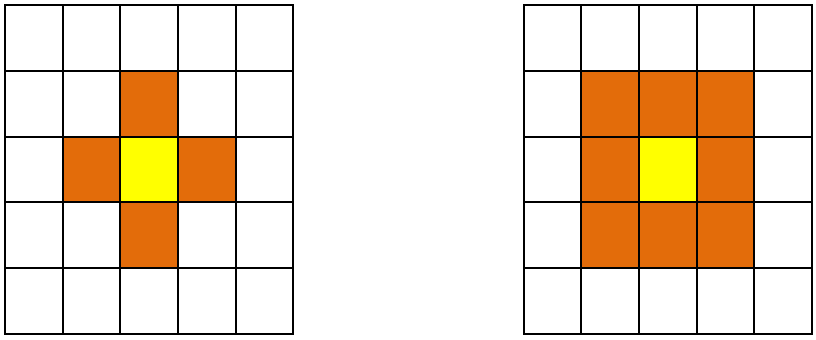
1.5 Висновки до розділу 1

Отже, було проведено аналіз предметної області та досліджено загальні характеристики та принципи роботи клітинних автоматів. Тепер можна перейти до наступного етапу, а саме – до розробки алгоритму для їх створення та застосування до зображень.

Розділ 2. Розробка алгоритму

2.1 Аналіз вимог

Першим етапом є визначення того, із якими саме клітинними автоматами ми будемо працювати. Очевидно, що нас цікавлять двовимірні автомати, як і було зазначено вище, адже зображення задаються двовимірними масивами значень, так само як і сітка двовимірного ітеративного масиву, де значення кожного елементу є значенням відповідної комірки. Та це лише перший елемент, як було вказано, для клітинного автомату необхідно іще три: набір можливих станів, визначення сусідства та правила переходу.

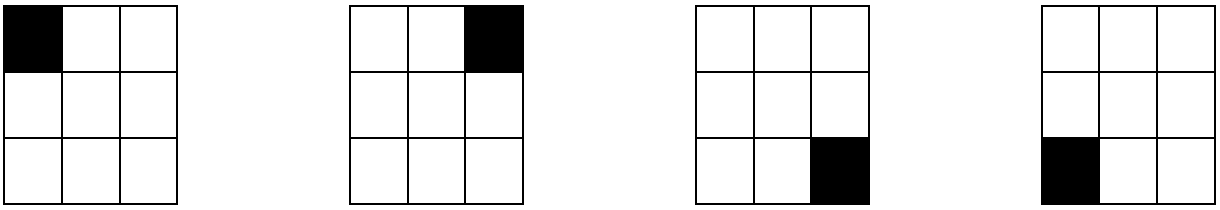


Малюнок 6, окіл фон Ноймана (ліворуч) та окіл Мура (праворуч)

Спершу визначимось із сусідством. Найчастіше у двовимірних автоматах використовують окіл Мура або окіл фон Ноймана. Завдання полягає у знаходженні контурів на зображенні, тобто точок із різкою зміною інтенсивності кольору. Оскільки товщина контурів теж відіграє важливу роль, то найбільше нас цікавить зміна у безпосередньо сусідніх пікселях (комірках), а не тих, що розташовуються далі, тож радіус сусідства був узятий за одиницю. В той же час комірки, які мають спільну вершину, а не грань, теж можуть вплинути на визначення границь, і аналізуючи вісім сусідів, а не тільки чотири, ми можемо отримати більш точні результати. Тож було вирішено взяти окіл Мура із радіусом один за визначення сусідства.

Перед тим, як визначитися із набором станів та правилами переходу, слід встановити, як саме будуть створюватися автомати. Для цього слід повернутися до розрахунків про кількість можливих автоматів, зроблених у першому

розділі. Як було показано, найпростіших клітинних автоматів з одним виміром, двома станами і сусідством, визначеним радіусом в одиницю, існує $2^8 = 256$. Збільшивши ж тільки виміри до потрібних нам двох отримуємо вже 2^{512} можливих наборів правил. Щоб же гарантовано визначити ті, які підходять для поставленої задачі, потрібно створити і перевірити кожен із них. Авжеж, можна виключити деякі правила, які очевидно не підходять, наприклад, якщо усі сусіди знаходяться в одному стані, тобто усі пікселі в даному околі однакові, то це однозначно не є контуром, тож цей піксель можна завжди переводити в стан 0. Також можна об'єднати симетричні правила, скажімо, якщо маємо правило 100000000 -> 0, то можна ввести і ще три правила: 001000000 -> 0, 000000001 -> 0, 000000100 -> 0, адже вони є обертаннями попереднього на 90°, втім число можливих наборів правил є занадто великим, щоби перевірити усі з них.



Малюнок 7, Симетричні правила переходу

При роботі ж із чорно-білими зображеннями кількість станів збільшується у сто двадцять вісім разів, тож очевидно, що перебрати усі правила таким же способом, як і у випадку елементарних автоматів, не вдасться. З огляду на це було прийнято рішення працювати із бінарними зображеннями що мають лише два кольори: чорний та білий і, відповідно, два стани. Також було вирішено використовувати тоталістичні клітинні автомати, тобто в правилах враховувати загальну кількість сусідніх білих пікселів, а не їхнє точне розташування. Таким чином кількість правил значно зменшиться, що дасть змогу перевірити усі з них за прийнятний час. Тут є дві альтернативи: тоталістичні та зовнішні тоталістичні автомати. Другий варіант має значно більше правил, однак через це повинен дати кращі результати, адже правила будуть більш детальними.

У випадку тоталістичих автоматів кожен піксель (клітина) може мати від нуля до десяти білих сусідів, тобто десять початкових розташувань. Існує два стани, в які клітина може перейти, отож маємо $2^{10} = 1024$ можливих правил. Пронумерувавши їх за аналогією до коду Вольфрама, у таблиці показано правило 180

Кількість сусідів	9	8	7	6	5	4	3	2	1	0
Наступний стан	0	0	1	0	1	1	0	1	0	0

Малюнок 8, Таблиця правил переходу для тоталістичного автомату 180

Якщо ж використовувати зовнішні тоталістичні автомати, в яких важливу роль відіграє поточний стан центральної комірки, маємо лише дев'ять варіантів сусідства: від нуля до восьми, однак для кожного з них розглядаються дві альтернативи для двох станів центрального пікселя. Тому загальна кількість початкових розташувань дорівнює вісімнадцяти, і маємо $2^{18} = 262144$ можливих правил. Аналогічно в таблиці показано правило 37655

Кількість сусідів	8		7		6		5		4		3		2		1		0	
Поточний стан	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
Наступний стан	0	0	1	0	0	1	0	0	1	1	0	0	0	1	0	1	1	1

Малюнок 9, Таблиця правил переходу для зовнішнього тоталістичного автомату 37655

Було вирішено спершу перевірити тоталістичні автомати, оскільки їхня кількість значно менша. Спершу вхідне зображення перетворюється на чорно-біле, якщо воно було кольоровим, а далі за допомогою алгоритму автоматичного порогування перетворюється на бінарне зображення, яке і буде далі оброблюватися. Тепер необхідно створити і застосувати усі клітинні автомати до вхідного зображення та зберегти результати.

2.2 Опис та реалізація алгоритмів

2.2.1 Алгоритм визначення границь на зображенні (контурування)

Набір правил для тоталістичного автомату можна задати одновимірним масивом розмірності десять, в якому індекс дорівнюватиме кількості сусідів, а значення на цьому індексі – наступному станові пікселя. Таким чином правило

180 із таблиці вище можна задати масивом A:[0,0,1,0,1,1,0,1,0,0]. Тобто якщо піксель має двох сусідів, то наступне його значення дорівнюватиме A[2]=1, якщо трьох – A[3]=0. Щоб же застосувати його, необхідно лише перебрати кожен піксель, порахувати кількість його сусідів і використати відповідне правило, після цього наш клітинний автомат перейде на наступну ітерацію. Єдина проблема полягає в тому, що, циклічно перебираючи кожен піксель, правило застосовується до цього одного пікселя, що може вплинути на всі ще не перевірені сусідні пікселі, а, як було зазначено вище, в клітинних автоматах правила мають застосовуватися синхронно одразу до всієї сітки. Однак це вирішується дуже просто – необхідно всього лише спершу створити копію вхідного автомату, яка не буде змінюватись, і рахувати кількість сусідів на ній. Це дозволить вільно змінювати значення пікселів зображення, з яким ми, власне, працюємо, не переймаючись за те, що це вплине на наступні пікселі.

Було створено метод для обрахунку кількості сусідів та клас, що містить зображення і масив правил переходів та метод, що застосовує друге до першого. Єдине, що відрізнятиметься в клітинних автоматах, – це саме масив правил переходу, тож для створення усіх можливих автоматів необхідно створити усі можливі правила. Оскільки для нумерації використовується аналог коду Вольфрама, і загальна кількість автоматів відома, то необхідно лише створити таку ж кількість екземплярів класу і передати параметром порядковий номер автомату. В конструкторі ж цей порядковий номер переводиться у двійкове представлення, за необхідності дописуючи на початок нулі, яке посимвольно записується в масив правил переходу.

Тепер потрібно лише застосувати кожне правило до вхідного зображення достатню кількість разів та зберегти результат. Задано максимальну дозволену кількість ітерацій, після якої закінчується виконання, і результат зберігається, а також запам’ятовується стан сітки перед кожною ітерацією. Якщо після кроку стан сітки не змінився, то це означає, що клітинний автомат перейшов у

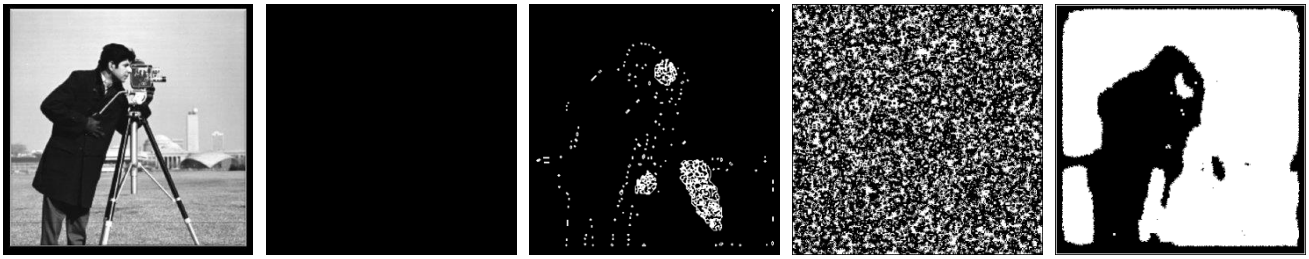
стабільний стан, і далі виконувати кроки не має сенсу, тож його робота теж припиняється.

Отож, схематично алгоритм виглядає наступним чином.

Малюнок 10, перший етап алгоритму

Таким чином було створено та застосовано усі тисяча двадцять чотири правила для тоталістичного клітинного автомату. Для спрощення аналізу результати, в яких не було жодного білого пікселя, або в яких кількість білих пікселів складала більше сорока відсотків від усіх, не зберігалися, адже вони очевидно не можуть бути шуканими контурами. Так як кількість вихідних зображень невелика, особливо із відкиданням непотрібних, то в даному випадку достатньо було візуального аналізу результатів. Він, однак, показав, що жоден із тоталістичних автоматів не дав бажаного результату. Всі вихідні зображення підпали під одну із чотирьох категорій:

1) Одноколірні зображення
2) Чорні зображення із деякими білими пікселями, були найближчими до бажаного результату
3) Зображення із приблизно рівною кількістю хаотично розташованих чорних та білих пікселів
4) Мало змінені від вхідного зображення зі згладженими контурами



Малюнок 11, Результати виконання тоталістичних автоматів до зображення фотографа

Таким чином було визначено, що тоталістичні автомати не підходять для вирішення поставленої задачі. Отже, тепер необхідно перевірити зовнішні тоталістичні. Ідея точно така ж, існує лише три відмінності. Перша полягає у значно більшій загальній кількості автоматів, друга – в тому, що при підрахунку кількості сусідів клітини ми не включаємо саму клітину. Третя полягає у представленні та застосуванні правил переходу. Тут наш масив A правил переходу вже матиме розмірність вісімнадцять, а відповідне правило застосовуватиметься не за формулою A[n], а вже A[n*2+o], де n – кількість сусідів, а o – зсув, який дорівнює поточному значенню клітини, оскільки ми розглядаємо два варіанти для кожної кількості сусідів. Після створення нового класу і методів для зовнішнього тоталістичного автомату, що імплементують ці відмінності, можна починати виконання.

Однак у цьому випадку ми стикаємося з двома проблемами. По-перше, застосування понад чверті мільйона можливих автоматів зайняло би занадто довгий час. По-друге, візуальний аналіз в даному випадку не спрацює з двох причин. Перша очевидна причина в тому, що занадто велика кількість результатів для аналізу. Друга ж в тому, що такий аналіз може лише відкинути непотрібні результати, але серед результатів правил, які таки виконали функцію контурування, неможливо лише візуально встановити, яке це зробило краще за інші. Отже потрібно виконати дві проміжні задачі: зменшити кількість правил та створити спосіб автоматичної оцінки роботи цих правил.

Вирішення першої є очевидним та дуже простим. Потрібно лише відкинути правила переходу, які точно не можуть бути присутні в шуканих автоматах. Це правила для нуля та восьми сусідів. Оскільки ми шукаємо

контури, то якщо піксель оточений лише чорними або лише білими пікселями, то він однозначно не є частиною контуру, тож його слід завжди переводити в стан нуль. Таким чином ми відкидаємо два лівих правила для восьми і два правих правила для нуля сусідів. Це лишає тільки чотирнадцять можливих початкових розташувань і зменшує кількість правил у шістнадцять разів із 2^{18} до 2^{14}=16384. Така кількість автоматів уже є цілком прийнятною для роботи, тож можна запускати їх на виконання та зберігати результати.

2.2.2 Алгоритм оцінки результатів

Тепер потрібно визначитися із тим, як саме ці результати оцінювати. Одним із найпоширеніших методів, що використовуються для оцінки роботи алгоритмів контурування, є оцінка якості Претта (Pratt's Figure of Merit) [15], яку й було обрано через її розповсюдженість та простоту застосування.

Для того, щоб оцінити роботу алгоритму контурування на якомусь зображенні, необхідно вже мати зображення зі знайденими контурами, із яким його можна буде порівняти. Такі зображення називаються базовими істинами. Ними виступають зображення, до яких були застосовані уже відомі алгоритми контурування. В цій роботі за базові істини були взяті результати операторів Собеля, Кенні та Прюіта. Суть будь-якого методу оцінки ефективності алгоритму контурування полягає у порівнянні того, наскільки контури на вихідному зображенні збігаються з контурами базової істини. Оцінка якості Претта бере до уваги кількість неправильно знайдених точок контурів, а також відстань від знайдених точок до контурів базової істини. Записується наступною формулою

$$FM = \frac{1}{\max\{I_I, I_A\}} \sum_{i=1}^{I_A} \frac{1}{1 + \alpha d_i^2} \qquad (1)$$

де I_I – кількість ідеальних граничних точок, тобто кількість білих пікселів на зображенні базової істини; I_A – кількість дійсних граничних точок, тобто знайдених алгоритмом, що перевіряється; d_i – це відстань від дійсної точки границі до істинної точки границі; α – це масштабуюча константа, має значення

$\frac{1}{9}$, як і було запропоновано в [15]. Інакше кажучи, це функція, що приймає два зображення як параметри – базову істину та зображення, що перевіряється. Спершу на кожному з них необхідно обрахувати загальну кількість контурних пікселів для першого множника $\frac{1}{\max\{I_I, I_A\}}$. Після цього для кожного граничного пікселя другого зображення необхідно знайти відстань d до найближчого граничного пікселя базової істини для другого множника $\sum_{i=1}^{I_A} \frac{1}{1+\alpha d_i^2}$.

Результатом є значення від нуля до одиниці включно, що відповідає тому, наскільки схоже вхідне зображення до базової істини. Що більше значення, то краще працює алгоритм, а значення одиниця означає, що вхідне зображення та базова істина є ідентичними.

Для імплементації оцінки якості Претта було створено клас, що містить два зображення: базову істину та зображення, яке із нею порівнюється, а також метод, що обраховує FM за формулою вище.

2.3 Висновки до розділу 2

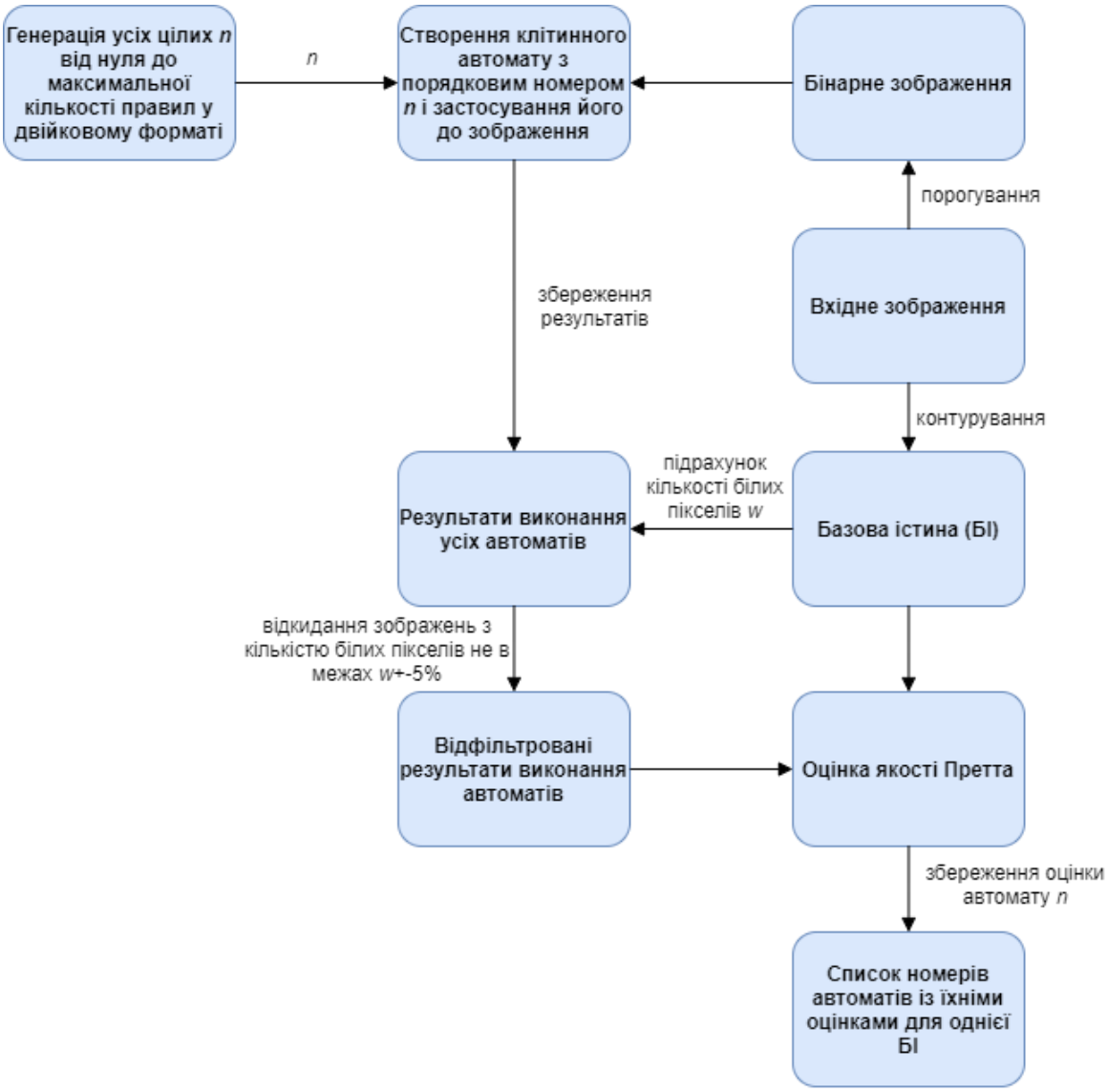
У цьому розділі було реалізовано алгоритм створення й застосування клітинних автоматів до зображень, а також алгоритм перевірки якості роботи методу знаходження границь. Тож тепер, маючи результуючі зображення від роботи усіх потрібних клітинних автоматів і спосіб оцінки контурування, можна перейти до аналізу результатів і визначення найкращих автоматів для вирішення поставленої задачі. Необхідно лише взяти набір тестових зображень та створити базові істини. Аналіз результатів та порівняння створених клітинних автоматів із поширеними методами контурування буде розглянуто в наступному розділі.

Розділ 3. Аналіз результатів та порівняння із класичними методами контурування

3.1 Аналіз роботи клітинних автоматів

Попри те, що кількість автоматів, яку ми перевіряємо, було зменшено з чверті мільйона до трохи понад шістнадцяти тисяч, перевіряти їх усі за допомогою оцінки якості Претта може зайняти досить багато часу, оскільки пошук відстані від кожної знайденої граничної точки до найближчої дійсної граничної точки виконується відносно повільно. Тому добре було би знову зменшити кількість зображень, відкинувши явно непотрібні. Швидкий візуальний аналіз показав, що тут, так само як і у випадку тоталістичних клітинних автоматів, більшість результатів є або одноколірними, або хаотичним розташуванням приблизно однакової кількості чорних та білих пікселів. Обидва типи зображень перевіряти оцінкою якості Претта не має сенсу, тож їх можна ігнорувати, потрібно лише встановити, за якою саме характеристикою такі зображення виявляти. Як було зазначено вище, оцінка якості Претта бере до уваги відстань між ідеальними та знайденими пікселями, а також їхню загальну кількість – тобто, якщо поки що опустити відстань, найкращими зображеннями є ті, в яких кількість білих пікселів найближча до їхньої кількості в базовій істині. Оскільки загальна кількість граничних точок рахується значно швидше, то саме це і було зроблено. Так, наприклад, для базової істини, заданої оператором Собеля на зображенні фотографа, що було продемонстроване вище, відсоток граничних точок складає 3,8% від загальної кількості пікселів. Знаючи приблизну потрібну кількість білих пікселів, ми можемо відкинути всі зображення, в яких вона значно відрізняється. Так, ігноруючи одноколірні результати, а також ті, де відсоток точок контурів відрізняється від базової істини більш ніж на п’ять відсотків, кількість зображень, що потрібно перевірити, зменшилась із шістнадцяти тисяч триста вісімдесяти чотирьох до всього лише сімсот шістдесяти п’яти.

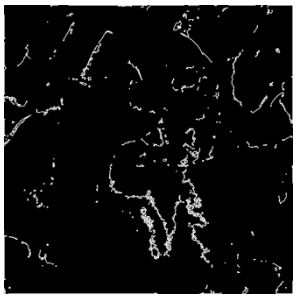
Маючи тепер досить невеликий набір даних, можна порівнювати їх із базовими істинами. Результат кожного порівняння зберігається у вигляді пар ключ-значення, де ключем є номер правила, а значенням – результат оцінки якості Претта. Після перевірки усіх зображень ці пари зберігаються у файлі відсортовані за значенням, тобто від правила, що дає результат, найбільш наближений до базової істини, до найбільш віддаленого. Тепер наш розширений алгоритм має вигляд:



Малюнок 12, другий етап алгоритму

Отже, правило, записане найпершим, є найкращим, однак лише для конкретної базової істини, яку ми перевіряли. Навіть якщо порівнювати дві базові істини, які є результатами одного і того ж алгоритму контурування до

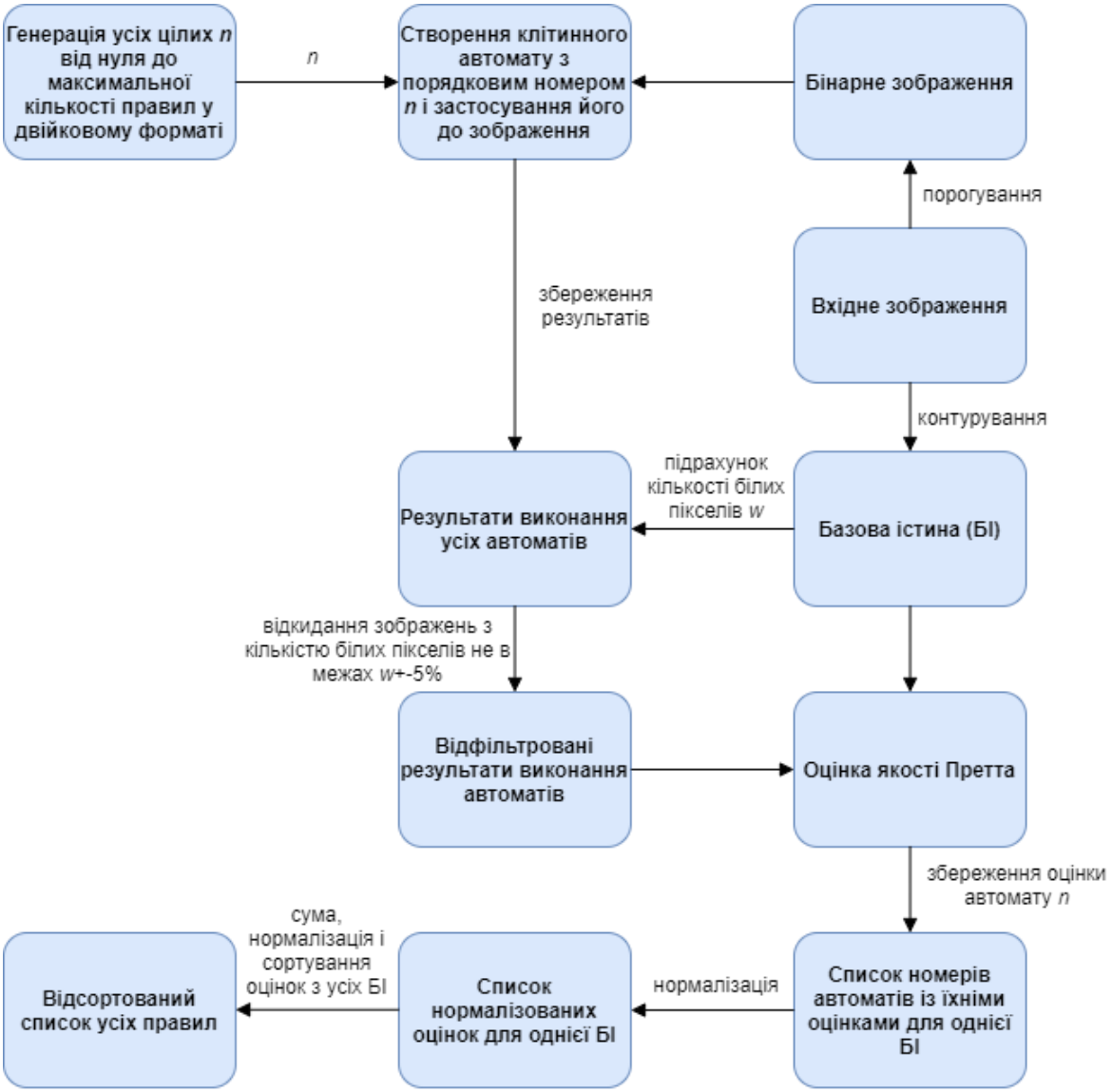
двох різних зображень, то правило, яке дає дуже подібний результат до однієї базової істини, може давати зовсім неприйнятний результат для другої. Наприклад, для двох із протестованих зображень, бабуїна та перців, пропущених через оператор Кенні, правило 47744 дає схожість в понад 91% для першого, але лише 20% для другого.

Кенні	Правило 47744	Кенні	Правило 47744
			

Малюнок 13, Порівняння результатів методу Кенні та правила 47744 на різних зображеннях

З огляду на це, необхідно знайти унікальне правило, яке даватиме максимально схожий результат для усіх зображень. Для цього було сформовано загальну оцінку для кожного правила, базуючись на усіх оцінках якості Претта, отриманих із кожного проаналізованого зображення.

Реалізовано це наступним чином. Нехай для якоїсь базової істини найкраще знайдене правило дало результат 0,95 за оцінкою якості Претта. Оскільки воно є найкращим, то воно має вагу 1. Вага усіх інших правил дорівнює їхній оцінці поділеній на оцінку найкращого. Тобто, якщо друге правило має оцінку 0,93, то його вага дорівнює $\frac{0,93}{0,95}$. Після цього ваги кожного правила з кожної базової істини сумуються та діляться на загальну кількість протестованих базових істин. Таким чином отримаємо нормалізовану оцінку від нуля до одиниці роботи кожного з перевірених клітинних автоматів. Тож фінальна версія алгоритму, від вхідного зображення та створення клітинних автоматів до виявлення найкращих із них, виглядає наступним чином:



Малюнок 14, третій етап алгоритму

Згадане вище правило 47744 має значення 0,772, а найкращий же результат показало правило 44704, яке має дуже близьке до одиниці значення – 0,973. Воно має вигляд:

Кількість сусідів	8		7		6		5		4		3		2		1		0	
Поточний стан	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
Наступний стан	0	0	1	0	1	0	1	1	1	0	1	0	1	0	0	0	0	0

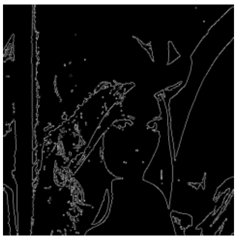
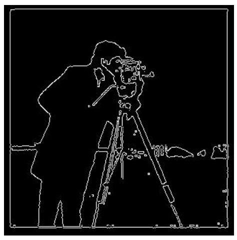
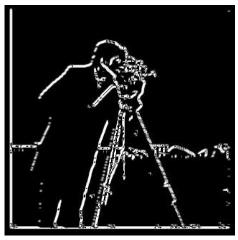
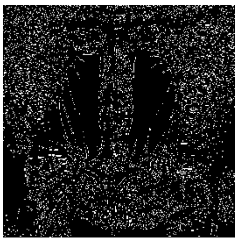
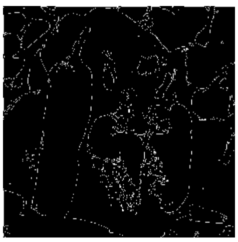
Малюнок 15, Таблиця правил переходу для правила 44704

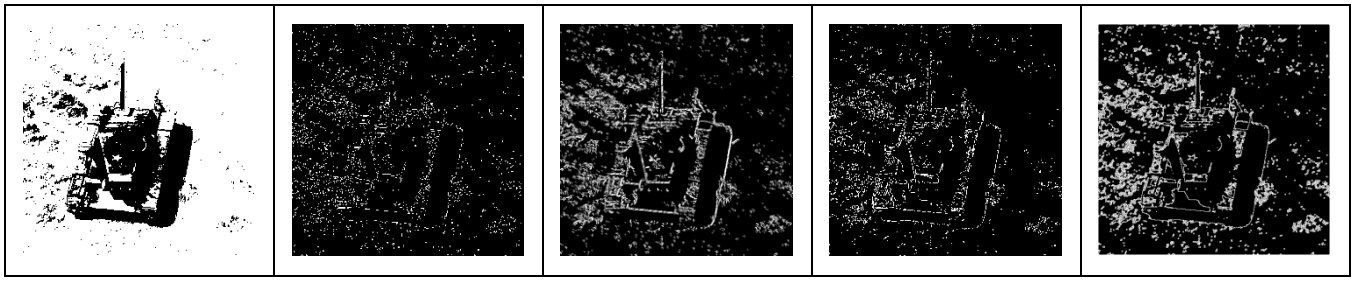
Якщо перекласти це у текстовий набір правил, за аналогією до гри життя Конвея, то матимемо всього два простих правила:

1) Всі білі пікселі, що мають від двох до семи сусідів, лишаються білими, всі інші стають чорними
2) Всі чорні пікселі, що мають рівно п’ятьох сусідів, стають білими, всі інші лишаються чорними

3.2 Порівняння з традиційними алгоритмами

Таким чином було встановлено клітинні автомати, що можуть використовуватися для контурування зображень на ряду із найпоширенішими алгоритмами. В таблиці нижче наведено порівняння роботи алгоритмів Кенні, Прюіта та Собеля із найкращим знайденим правилом 44704:

Вхідне зображення	Кенні	Прюіт	Собель	Правило 44704
				
				
				
				



Малюнок 16, Порівняння результатів роботи правила 44704 із алгоритмами Кенні, Прюіта та Собеля

3.3 Висновки до розділу 3

У даному розділі було визначено спосіб загального оцінювання роботи клітинного автомату, базуючись на отриманих оцінках якості Претта від порівняння із окремими базовими істинами. За допомогою цього серед знайдених клітинних автоматів було виявлено найкращий для знаходження границь на зображенні. Також продемонстровано порівняння його роботи із роботою відомих алгоритмів контурування, таких як Кенні, Собеля та Прюіта.

Висновки по роботі та рекомендації для подальших досліджень

В ході цієї роботи було досліджено клітинні автомати, розглянуто їхні види та класифікацію, а також сфери практичного застосування. Було розроблено алгоритм для створення та застосування всіх можливих двовимірних тоталістичних і зовнішніх тоталістичних клітинних автоматів до бінарних зображень з метою знаходження границь. За допомогою оцінки якості Претта було визначено найкращі з автоматів для контурування, а також проведено порівняння їхньої роботи із роботою відомих класичних методів знаходження границь. Втім, існує ще багато тем для подальшого дослідження. Наприклад, в даній роботі використовувався окіл Мура з радіусом одиниця для визначення сусідства. Можна дослідити ефективність застосування зовнішніх тоталістичних автоматів із більшим радіусом для контурування. Вони, хоч і можуть продукувати товщі контури, можуть також давати точніші результати, а товщину границь можна зменшити методом зтоншення контурів, який теж можна реалізувати за допомогою клітинних автоматів. Окрім цього, темами для дослідження є не тоталістичні автомати, а також контурування чорно-білих зображень, а не бінарних. Втім дослідження цих тем вимагатиме цілком іншого підходу, адже, як було зазначено в другому розділі, кількість можливих варіантів у цих випадках виключатиме можливість перебору й перевірки їх усіх.

Список використаних джерел

[1] John Canny, A Computational Approach to Edge Detection, 1986, http://doi.org/10.1109/tpami.1986.4767851

[2] Irwin Sobel, An Isotropic 3x3 Image Gradient Operator, https://www.researchgate.net/publication/239398674_An_Isotropic_3x3_Image_Gradient_Operator

[3] Stephen Wolfram, Statistical mechanics of cellular automata, 1983, http://doi.org/10.1103/RevModPhys.55.601

[4] Matthew Cook, Universality in Elementary Cellular Automata, https://wpmedia.wolfram.com/uploads/sites/13/2018/02/15-1-1.pdf

[5] Martin Gardner, The fantastic combinations of John Conway's new solitaire game "life", 1970, https://web.stanford.edu/class/sts145/Library/life.pdf

[6] Dan Gordon, On the Computational Power of Totalistic Cellular Automata, 1987, https://doi.org/10.1007/BF01692058

[7] Paul Chapman, Life Universal Computer, 2002, http://www.igblan.free-online.co.uk/igblan/ca/

[8] Stephen Coombes, The Geometry and Pigmentation of Seashells, 2009, https://www.maths.nottingham.ac.uk/plp/pmzsc/pdfs/Seashells09.pdf

[9] David Peak et al., Evidence for complex, collective dynamics and emergent, distributed computation in plants, 2004, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC327117/

[10] Gerhardt, Schuster, A cellular automaton describing the formation of spatially orderd structures in chemical systems, 1989, http://doi.org/10.1016/0167-2789(89)90081-x

[11] Kendall Preston, Michael Duff, Modern Cellular Automata: Theory and Applications, 1984, 978-1489903952

[12] Nandi et al., Theory and Applications of Cellular Automata in Cryptography, 1994, http://doi.org/10.1109/12.338094

[13] Tomassini et al., On the generation of high-quality random numbers by two-dimensional cellular automata, 2000, http://doi.org/10.1109/12.888056

[14] Lawrence Johnson, Cellular automata for real-time generation of infinite cave levels, 2010, http://doi.org/10.1145/1814256.1814266

[15] William Pratt, Quantitative Design and Evaluation of Enhancement/Thresholding Edge Detectors, 1979, http://doi.org/10.1109/PROC.1979.11325

Додаток А

Текст класу EdgeDetectionCA2

```
PImage input;
PImage output;
PImage binary;
PImage greyScale;
int treshold = 50;
boolean tresholdIncreasing = true;
boolean temp = true;
int counter = 0;

int ruleLen = 10;
Rule[] rules;

int OTRuleLen = 18;
//number of pixels on each edge of the rule that can be disregarded. For example, if a pixel has 0 or 8 neighbours, then it's not an edge and result will always be 0
int edgePixels = 2;
OTRule[] OTRules;

void setup() {
  //size(512, 512);
  binary = loadImage("Binary/Photographer.jpeg");
  //createAllTRules(binary);
  //createAllOTRules(binary);
  //String path = "C:/Processing-3.3.6/Sketches/CellularAutomata/EdgeDetectionCA2/LenaValidated";
  //applyPFOM(path, "GroundTruths/LenaCannyGT.jpeg", "PhotographerPrewittSortedPFOM.txt");
  //validateImages("C:/Processing-3.3.6/Sketches/CellularAutomata/EdgeDetectionCA2/PhotographerUnvalidated",
  //"C:/Processing-3.3.6/Sketches/CellularAutomata/EdgeDetectionCA2/PhotographerValidated(15-35)", 0.001, 0.15);
  //applyPFOM("C:/Processing-3.3.6/Sketches/CellularAutomata/EdgeDetectionCA2/LenaAutoValidated",
  //        "C:/Processing-3.3.6/Sketches/CellularAutomata/EdgeDetectionCA2/GroundTruths/LenaSobelGT.png",
  //        "LenaSobelAutoPFOM.txt");


  //applyBestRules();
```

```
  //findTheBestRule();

  noLoop();
}


void applyBestRules() {
  String[] bestRules = bestRules();

  OTRules = new OTRule[bestRules.length];
  for (int i = 0; i < bestRules.length; i++) {
    OTRules[i] = new OTRule(int(bestRules[i]), OTRuleLen);
  }

  for (int i = 0; i < bestRules.length; i++) {
    print(i + "/" + bestRules.length + ": ");
    PImage bin = binary.copy();
    OTRules[i].apply(bin);
  }
}

//selects best rules from all available
String[] bestRules() {
  String[] files = {"LenaCannyPFOM.txt", "LenaSobelPFOM.txt",
"PhotographerCannyPFOM.txt", "PhotographerSobelPFOM.txt"};
  //number of top rules to keep
  int n = 100;

  ArrayList<String> rulesList = new ArrayList<String>();

  for (int j = 0; j < files.length; j++) {
    String str = loadStrings(files[j])[0].substring(1);
    String[] rules = str.split(",");

    for (int i = 0; i < n; i++) {
      String rule = rules[i].substring(2);
      String ruleNum = rule.substring(0, rule.indexOf("\""));
      //float ruleVal = float(rule.substring(rule.indexOf(":")+1));

      if (!rulesList.contains(ruleNum))
        rulesList.add(ruleNum);
    }
  }
```

```
  return rulesList.toArray(new String[0]);
}

//creates all possible rules for a totalistic CA and applies each of them to the input
image @in
void createAllTRules(PImage in){
  rules = new Rule[(int)pow(2, ruleLen)];
  for(int i = 0; i < pow(2, ruleLen); i++){
    rules[i] = new Rule(i, ruleLen);
  }

  for(int i = 0; i < rules.length; i++){
    PImage bin = in.copy();
    rules[i].apply(bin);
  }
}

//creates all possible rules for an outer totalistic CA and applies each of them to the
input image @in
void createAllOTRules(PImage in){
  //maximum number of possible rules
  int maxRules = (int)pow(2, OTRuleLen - edgePixels*2);

  OTRules = new OTRule[maxRules];
  for (int i = 0; i < maxRules; i++) {
    OTRules[i] = new OTRule(i, OTRuleLen);
  }

  for (int i = 0; i < maxRules; i++) {
    print(i + "/" + maxRules + ": ");
    PImage bin = in.copy();
    OTRules[i].apply(bin);
  }
}

//goes through all the images in @path and compares them to the ground truth image
GTPath. Outputs to a file @out all the rules with their PFOM values sorted from best
to worst
void applyPFOM(String path, String GTPath, String out) {
  String[] files = new File(path).list();

  FloatDict RulesPFOM = new FloatDict();

  PFOM pfom = new PFOM();
```

```
  pfom.loadGTImg(GTPath);

  for (int i = 0; i < files.length; i++) {
    println(i + "/" + files.length);
    pfom.loadImg(path + "/" + files[i]);
    RulesPFOM.set(files[i].substring(0, files[i].length()-4), pfom.validate());
  }

  PrintWriter pw = createWriter(out);
  RulesPFOM.sortValuesReverse();
  println(RulesPFOM.toJSON());
  pw.println(RulesPFOM.toJSON());
  pw.flush();
  pw.close();
}

//Checks all images from @in and saves the ones which have between @lowTreshold
and @hiTreshold percentage of white pixels to @out
void validateImages(String in, String out, float lowTreshold, float hiTreshold) {
  String[] files = new File(in).list();
  for (int i = 0; i < files.length; i++) {
    println(i + "/" + files.length);
    PImage img = loadImage(in + "/" + files[i]);

    float white = countWhitePixels(img);

    if (white < lowTreshold || white > hiTreshold)
      continue;

    img.save(out + "/" + files[i]);
  }
}

//returns the percentage [0..1] of white pixels in image @img
float countWhitePixels(PImage img) {
  float white = 0.;
  for (int j = 0; j < img.pixels.length; j++)
    if (brightness(img.pixels[j]) == 255)
      white++;

  return white / img.pixels.length;
}

//evaluates the overall merit of each rule based on all gathered PFOM values and
outputs in descending order to file BestRules.txt
```

```
void findTheBestRule() {
  String[] PFOMs = {"BaboonCannyPFOM.txt", "BaboonPrewittPFOM.txt",
"BaboonSobelPFOM.txt",
    "PeppersCannyPFOM.txt", "PeppersPrewittPFOM.txt", "PeppersSobelPFOM.txt",
    "TankCannyPFOM.txt", "TankPrewittPFOM.txt", "TankSobelPFOM.txt",
    "LenaCannyAutoPFOM.txt", "LenaPrewittAutoPFOM.txt",
"LenaSobelAutoPFOM.txt",
    "PhotographerCannyPFOM.txt", "PhotographerPrewittPFOM.txt",
"PhotographerSobelPFOM.txt"};

  FloatDict fd = new FloatDict();

  for (int i = 0; i < PFOMs.length; i++) {
    String str = loadStrings(PFOMs[i])[0].substring(1);
    String[] rules = str.split(",");

    float maxVal = 0;

    for (int j = 0; j < rules.length; j++) {
      String rule = rules[j].substring(2);
      String ruleNum = rule.substring(0, rule.indexOf("\""));
      float ruleVal = float(rule.substring(rule.indexOf(":")+1));

      if(j == 0)
        maxVal = ruleVal;

      //to make sure each PFOM has the same weight
      ruleVal = ruleVal / maxVal;

      if (fd.hasKey(ruleNum))
        ruleVal += fd.get(ruleNum);

      fd.set(ruleNum, ruleVal);
    }
  }

  String[] keys = fd.keyArray();
  for(int i = 0; i < keys.length; i++){
    fd.set(keys[i], fd.get(keys[i]) / PFOMs.length);
  }

  PrintWriter pw = createWriter("BestRules.txt");
  fd.sortValuesReverse();
  println(fd.toJSON());
  pw.println(fd.toJSON());
```

```
  pw.flush();
  pw.close();
}

boolean arraysIdentical(int[] a1, int[] a2) {
  if (a1.length != a2.length)
    return false;

  for (int i = 0; i < a1.length; i++)
    if (a1[i] != a2[i])
      return false;

  return true;
}
```

Текст класу ImgProcessor

```
void toGreyscale(PImage in, PImage out) {
  out.loadPixels();
  for (int x = 0; x < w; x++) {
    for (int y = 0; y < h; y++ ) {
      int loc = x + y*in.width;
      out.pixels[loc] = color(brightness(in.pixels[loc]));
    }
  }
  out.updatePixels();
}

void toBinary(PImage in, PImage out, int treshold) {
  out.loadPixels();
  for (int x = 0; x < w; x++) {
    for (int y = 0; y < h; y++ ) {
      int loc = x + y*in.width;
      if (brightness(in.pixels[loc]) < treshold) {
        out.pixels[loc] = color(0);
      } else {
        out.pixels[loc] = color(255);
      }
    }
  }
  out.updatePixels();
}

void treshold(PImage in, PImage out) {
  //set initial treshold to be mean of the entire image
```

```
  double t = 0;
  double treshold;
  for (int i = 0; i < in.pixels.length; i++) {
    t += brightness(in.pixels[i]);
  }
  treshold = t/in.pixels.length;

  double prevTreshold = -1;

  //separate background and foreground and find new treshold to be average of their
means. Repeat until treshold value stops changing
  double backgroundMean = 0;
  double foregroundMean = 0;
  do {
    double back = 0;
    int backCount = 0;
    double front = 0;
    int frontCount = 0;
    for (int i = 0; i < in.pixels.length; i++) {
      if (brightness(in.pixels[i]) > treshold) {
        front += brightness(in.pixels[i]);
        frontCount++;
      } else {
        back += brightness(in.pixels[i]);
        backCount++;
      }
    }
    backgroundMean = back/backCount;
    foregroundMean = front/frontCount;
    prevTreshold = treshold;
    treshold = (backgroundMean + foregroundMean)/2;
  } while (treshold != prevTreshold);

  //treshold the output image with found treshold value
  out.loadPixels();
  for (int i = 0; i < in.pixels.length; i++) {
    if (brightness(in.pixels[i]) > treshold)
      out.pixels[i] = color(255);
    else
      out.pixels[i] = color(0);
  }
  out.updatePixels();
}

//sum of Moore Neighbourhood excluding the central cell
```

```
int binaryMooreNeighborSumEx(PImage in, int x, int y) {
  int sum = 0;
  sum += brightness(in.pixels[x-1 + y*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x+1 + y*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x + (y-1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x + (y+1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x-1 + (y-1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x+1 + (y-1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x-1 + (y+1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x+1 + (y+1)*in.width]) == 0 ? 0 : 1;
  return sum;
}

//sum of Moore Neighbourhood including the central cell
int binaryMooreNeighborSumIn(PImage in, int x, int y) {
  int sum = 0;
  sum += brightness(in.pixels[x + y*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x-1 + y*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x+1 + y*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x + (y-1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x + (y+1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x-1 + (y-1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x+1 + (y-1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x-1 + (y+1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x+1 + (y+1)*in.width]) == 0 ? 0 : 1;
  return sum;
}

int binaryNeumanNeighborSum(PImage in, int x, int y) {
  int sum = 0;
  sum += brightness(in.pixels[x-1 + y*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x+1 + y*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x + (y+1)*in.width]) == 0 ? 0 : 1;
  sum += brightness(in.pixels[x + (y+1)*in.width]) == 0 ? 0 : 1;
  return sum;
}
```

Текст класу Rule

```
//rule for a totalistic CA
class Rule {
  int number;
  int[] bin;
```

```
  //creates an array of values that each cell will get on the next generation for the
current rule number going left to right
  //first rule is 0..0, last rule is 255..255
  //values are multiplied by 255 in order to represent white color which is 255 and not
1
  Rule(int number, int maxlen) {
    this.number = number;
    String[] str = binary(number, maxlen).split("");
    bin = new int[maxlen];
    for (int i = 0; i < maxlen; i++)
      bin[i] = Integer.valueOf(str[i])*255;
  }

  void step(PImage img) {
    PImage buffer = img.copy();
    img.loadPixels();

    for (int x = 1; x < img.width-1; x++) {
      for (int y = 1; y < img.height-1; y++ ) {
        int loc = x + y*img.width;
        int sum = binaryMooreNeighborSumIn(buffer, x, y);
        img.pixels[loc] = color(bin[9-sum]);
      }
    }

    img.updatePixels();
  }

  void apply(PImage img) {
    int generations = 0;
    PImage previous;
    do {
      previous = img.copy();
      step(img);
    } while (++generations < 200 && !arraysIdentical(previous.pixels, img.pixels));
    boolean validated = validate(img);
    println("Rule " + number + " done. Generations: " + generations + ". Validated: " +
validated);
    //if (validated)
      img.save("output/" + number);
  }

  //checks if the output image is worth saving
  //if the number of white pixels is below 5% of all the pixels, then no need to save
  //if the number of white pixels is over 40% of all the pixels, then no need to save
```

```
  boolean validate(PImage img) {
    double white = 0;
    for (int i = 0; i < img.pixels.length; i++) {
      if (brightness(img.pixels[i]) == 255)
        white++;
    }
    if (white <= 0.05 * img.pixels.length)
      return false;

    if (white >= 0.4  * img.pixels.length)
      return false;

    return true;
  }
}
```

Текст класу OTRule

```
//rule for an outer totalistic CA
class OTRule {
  int number;
  int[] bin;

  //creates an array of values that each cell will get on the next generation for the
current rule number going left to right
  //first rule is 0..0, last rule is 255..255
  //even indexes are for white central cell, odd indexes are for balck central cell [8/1,
8/0, 7/1, 7/0, ..., 0/1, 0/0]
  //values are multiplied by 255 in order to represent white color which is 255 and not
1
  OTRule(int number, int maxlen) {
    String[] str = binary(number, maxlen - edgePixels*2).split("");
    bin = new int[maxlen];
    for (int i = edgePixels; i < maxlen - edgePixels; i++)
      bin[i] = Integer.valueOf(str[i - edgePixels])*255;

    //calculating the actual rule number (equal to transforming binary to decimal)
    int n = 0;
    for (int i = 0; i < bin.length; i++)
      if(bin[i] != 0)
      n += pow(2, bin.length-i-1);
    this.number = n;
  }

  void step(PImage img) {
```

```
  PImage buffer = img.copy();
  img.loadPixels();

  for (int x = 1; x < img.width-1; x++) {
    for (int y = 1; y < img.height-1; y++ ) {
      int loc = x + y*img.width;
      int sum = binaryMooreNeighborSumEx(buffer, x, y);

      //if the central pixel is black, then offset the index by 1
      int offset = brightness(buffer.pixels[loc]) == 0 ? 1 : 0;
      img.pixels[loc] = color(bin[(8-sum)*2 + offset]);
    }
  }

  img.updatePixels();
 }

 void apply(PImage img) {
  int generations = 0;
  PImage previous;
  do {
    previous = img.copy();
    step(img);
  } while (++generations < 100 && !arraysIdentical(previous.pixels, img.pixels));
  //boolean validated = validate(img);
  //println("Rule " + number + " done. Generations: " + generations + ". Validated: "
+ validated);
  //if (validated)
  //  img.save("output/" + number);

  println("Rule " + number + " done. Generations: " + generations);
  img.save("Test/" + number);
 }

 //checks if the output image is worth saving
 //if the number of white pixels is below 5% of all the pixels, then no need to save
 //if the number of white pixels is over 40% of all the pixels, then no need to save
 boolean validate(PImage img) {
  double white = 0;
  for (int i = 0; i < img.pixels.length; i++) {
    if (brightness(img.pixels[i]) == 255)
      white++;
  }
  if (white <= 0.05 * img.pixels.length)
    return false;
```

```
    if (white >= 0.4  * img.pixels.length)
      return false;

    return true;
  }
}
```

Текст класу PFOM

```
class PFOM {
  //width and height of images
  int w, h;
  //ground truth image
  PImage gt;
  int[] trueEdgePoints;
  //image to compare
  PImage img;
  int[] actualEdgePoints;

  //load the ground truth image and store the coordinates of its edge points
  void loadGTImg(String path) {
    gt = loadImage(path);
    w = gt.width;
    h = gt.height;

    IntList points = new IntList();
    for (int i = 0; i < gt.pixels.length; i++)
      if (brightness(gt.pixels[i]) == 255)
        points.append(i);

    trueEdgePoints = points.array();
  }

  //load an image to be validated and store the coordinates of its edge points
  void loadImg(String path) {
    img = loadImage(path);
    IntList points = new IntList();
    for (int i = 0; i < img.pixels.length; i++)
      if (brightness(img.pixels[i]) == 255)
        points.append(i);
    actualEdgePoints = points.array();
  }

  float validate() {
```

```
    float r = 0;

    //number of ideal edge points
    int Ii = trueEdgePoints.length;
    //number of actual edge points
    int Ia = actualEdgePoints.length;
    int In = Ii > Ia ? Ii : Ia;
    float a = 1./9;

    float sum = 0;

    for (int i = 0; i < actualEdgePoints.length; i++) {
      float d = Float.MAX_VALUE;
      int x1 = actualEdgePoints[i] % w;
      int y1 = actualEdgePoints[i] / w;

      for (int j = 0; j < trueEdgePoints.length; j++) {
        int x2 = trueEdgePoints[j] % w;
        int y2 = trueEdgePoints[j] / w;
        float dist = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
        if (dist < d)
          d = dist;
        //if already found a pixel in the right spot, no need to keep checking the rest
        if (d == 0)
          continue;
      }

      sum += 1.0 / (1.0 + a*d*d);
    }

    r = sum / In;
    return r;
  }
}
```