Міністерство освіти та науки України
Криворізький державний педагогічний університет
Інститут соціального управління, економіки та підприємництва

В.М. Соловйов
І.О. Теплицький
С.О. Семеріков

Методи математичного моделювання

Лабораторний практикум з курсу

Видання третє, виправлене

Кривий Ріг–Черкаси
2003

Методичні вказівки з виконання лабораторних робіт для студентів денної форми навчання з дисципліни “Методи математичного моделювання”. – 104 стор.

Укладачі: д.ф.-м.н., проф. В.М. Соловйов
к.пед.н., доц. І.О. Теплицький
к.пед.н., доц. С.О. Семеріков
Рецензент: к.т.н., с.н.с. О.П. Поліщук
Комп’ютерний набір: к.пед.н., доц. С.О. Семеріков
Відповідальний за випуск: к.пед.н., доц. С.О. Семеріков

Методичні вказівки розглянуті та рекомендовані до видання на засіданні кафедри інформатики та прикладної математики Криворізького державного педагогічного університету від “24” лютого 2003 р., протокол №6.

У методичних вказівках подаються теоретичні відомості з курсу, зміст завдань лабораторних робіт та приклади програм, необхідні для їх виконання.

Наклад 120 примірників.

Зміст

§1. Метод Монте-Карло 4
§2. Клітковий автомат «Життя» 8
§3. Фрактали 15
§4. Подібність та скейлінг 22
§5. Метод Хаффмана 26
§6. Арифметичне кодування 39
§7. Алгоритм Лемпеля-Зіва-Велча 61
§8. Імітаційне моделювання відмов обладнання 70
§9. Універсальний клітковий автомат 71
§10. Лісові пожежі 74
§11. Визначення перколяційного порогу 75
§12. Клітковий автомат «Сніжинка» 82
§13. Клітковий автомат «Хижак–жертва» 87
§14. Розмивання берегової лінії 90
§15. Електроліз на плоскому катоді 96
§16. Електроліз на точковому катоді 97
§17. Задача багатьох тіл 99

§1.

Метод Монте-Карло

Існують явища, для яких не можливо за допомогою точних формул врахувати численні й різноманітні дії певних випадкових факторів. Зокрема, таке може мати місце у ситуаціях, коли характеристики об'єктів дослідження за своєю суттю можуть набувати лише випадкових значень. Наприклад, явища, пов'язані з поведінкою мікрочастинок, технологічні процеси на виробництві, соціологічні дослідження тощо.

Загальноприйнятим методом моделювання систем, які містять такі характеристики, є метод випадкової вибірки, звичайно відомий під назвою методу Монте-Карло. Створення цього методу пов'язане з роботою видатного американського математика, одного із засновників кібернетики, Джона фон Неймана, котрий наприкінці 40-х років увів цей термін при розв'язанні задачі про екранування від ядерних випромінювань. Назва методу походить від імені столиці князівства Монако, відомої своїми гральними домами, де якненайголовніше місце посідає рулетка.

Якщо рулетку гарно збалансовано, кулька може зупинитися у будь-якому положенні, тому ймовірність одержання будь-якого числа однакова для всіх чисел на барабані.

Поглянемо на рулетку не з позиції гравця, а з позиції дослідника. Уявимо, що на диску містяться цілі числа з інтервалу [1; 100]. Після кожного випробування будемо у прямокутній системі координат на вісь ординат відкладати ці числа, а на вісь абсцис – відповідно їхні порядкові номери. Тоді при достатньо великій кількості випробувань ми одержимо множину точок, які будуть рівномірно розподілені на деякій площі.

При необхідності одержати m-розрядні числа можна було б m разів підряд пустити рулетку або випустити з лототрону підряд m куль.

В дійсності немає потреби багаторазово обертати рулетку або барабан лототрону, оскільки випадкові числа визначені заздалегідь і зведені до таблиць.

Якщо говорити строго, то рівномірний розподіл випадкових чисел – це ідеалізоване математичне поняття. У природних і виробничих умовах, у суспільних явищах і т.п. практично зустрічаються розподіли нерівномірні.

Нормальний розподіл характерний для коливань купівельного попиту, для величини врожаю у різні роки, для виробничих похибок та похибок вимірювань, для рівня перешкод при передаванні інформації тощо. Основні властивості такого розподілу:

- випадкові величини групуються навколо деякого певного числа;
- чим менше за абсолютною величиною відхилення від цього числа, тим частіше воно виникає, тобто малі відхилення більш ймовірні, аніж великі;
- рівні за величиною, але протилежні за знаком відхилення виникають однаково часто;
- максимальне значення випадкових відхилень не перебільшує деякої величини, яка зветься граничною похибкою.

Відомі й інші види нерівномірних розподілів випадкових величин.

При моделюванні випадкових величин їх реальний розподіл визначають одним з двох способів:

1) на основі дослідних даних, які одержують шляхом здійснення спеціального експерименту;
2) за певним теоретичним законом методами математичної статистики.

Для роботи ЕОМ з випадковими числами спочатку було здійснено спроби вводити ці числа до машини іззовні. Вводили у пам’ять готові таблиці, будували прилади на основі випадкових фізичних процесів (наприклад, радіоактивного розпаду або підрахунку кількості електронів, що вилітають за деякий фіксований проміжок часу з гарячого катоду) і одержані на цих приладах числа також вводили у пам’ять. Однак і те, і інше працювало однаково погано: таблицю випадкових чисел ЕОМ швидко вичерпувала, а випадкове фізичне явище взагалі не можна відтворити з тією самою послідовністю чисел для перевірки розрахунків. Саме у цій ситуації Джон фон Нейман придумав алгоритм генеруван-

ня (утворення) чисел, дуже схожих на випадкові і рівномірно розподілених у інтервалі [0, 1]. Ці числа ще називають псевдовипадковими (нібито випадковими), оскільки їхня послідовність є періодичною. Кількість чисел у періоді намагаються збільшувати шляхом удосконалення алгоритму їх утворення. У сучасних мовах програмування такі алгоритми реалізовано у спеціальних стандартних функціях. Наприклад, відома функція *random*(*x*) генерує рівномірно розподілену послідовність у інтервалі [0; 1]. Назва функції походить від англійського *random* – випадковий або вибраний навмання.

Цікаво відмітити, що випадкові числа, які з філософської точки зору найчистіше являють собою відображення нашого незнання, покладено у основу методів, за допомогою яких дослідники пізнають поведінку складних систем.

Сутність методу Неймана, або *методу середини квадрату*, полягає у такому. Нехай нам потрібні чотиризначні псевдовипадкові числа. Оберемо довільне число x_0. Наприклад, x_0=8219. Піднесемо його до квадрату – отримаємо восьмизначне число 67551961. Вилучимо середні цифри: 5519. Наступним числом послідовності є x_1=5519. Піднесемо до квадрату 5519 – отримаємо 30459361. Наступне випадкове число x_2=4593. Якщо перші з середніх цифр виявляються нулями, то отримуємо число з меншою кількістю знаків. Так, x_2^2=21095649, x_3=956. Підносячи його до квадрату, можна отримати восьмизначне число, дописавши перед ним нулі: x_3^2=00913936, так що x_4=9139 і т.д.

Випадкові числа, розподілені в інтервалі від нуля до одиниці, отримуються з чисел x_i за формулою $y_i=x_i/10^4$.

Суть *лінійного конгруентного методу* полягає у такому: обирають чотири цілих додатних числа: множник *k*, зсув *c*, модуль *m*, перше число послідовності x_0. Послідовність випадкових чисел визначається формулою

$$x_{n+1}=(kx_n+c) \bmod m.$$

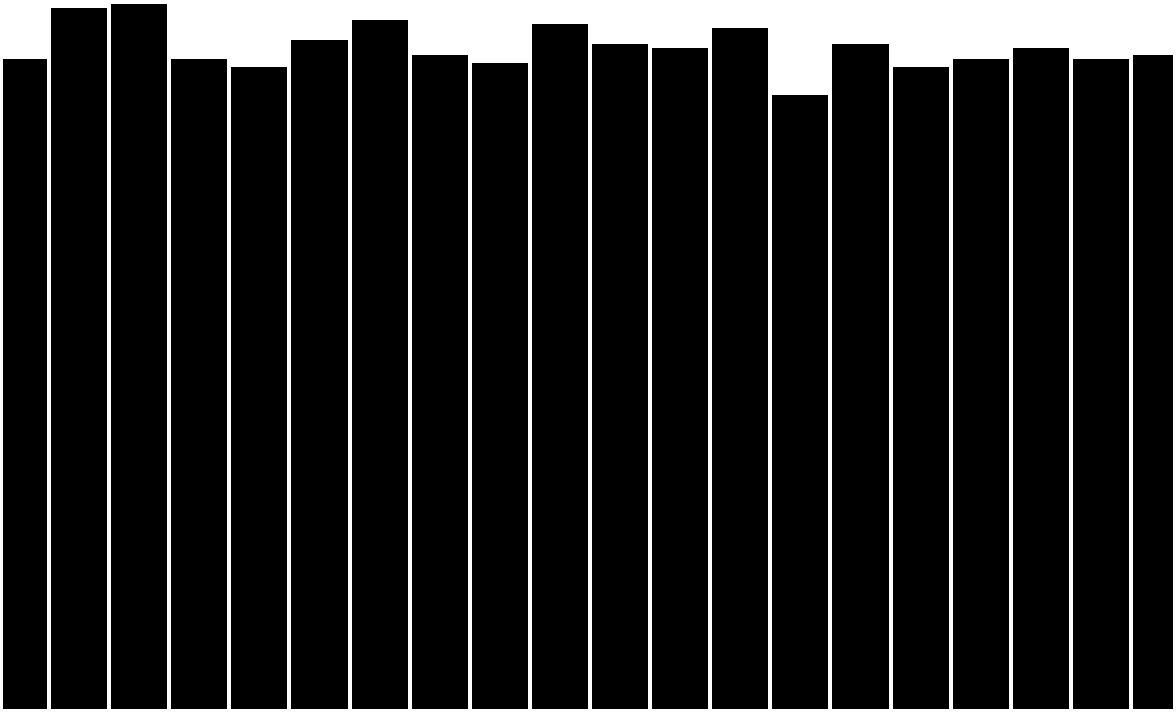
Очевидно, що залишок від ділення не перевищує діленого, тому всі числа послідовності x_n задовольняють нерівності $x_n<m$. Послідовність чисел y_i, рівномірно розподілених в інтервалі від нуля до одиниці, одержується за формулою $y_i=x_i/m$.

Зрозуміло, що серед усіх послідовностей, що їх можна згенерувати вибором різних четвірок чисел, нас найбільше цікавитиме та, період повтору якої найдовший. Це можливо тоді, коли *c* та *m* – взаємно прості числа.

Виходячи з вищевикладеного, сформулюємо таке завдання:

1. Розділить інтервал [0; 1) на 20 рівних проміжків.
2. Використовуючи різні методи генерації випадкових чисел (стандартний генератор, метод середини квадрату та лінійний конгруентний метод), підрахуйте кількість «попадань» у кожний з проміжків за різної кількості випадкових чисел.
3. За результатами обчислювального експерименту для кожного генератора побудуйте стовбчасту діаграму так, щоб висоти стовпців відповідали кількості «попадань» випадкових чисел у проміжки.
4. Зробіть висновок про природу побудованого розподілу.

Типову діаграму зображено на малюнку:



§2.

Клітковий автомат «Життя»

В цьому циклі робіт ми розглянемо клас геометричних моделей, які мають назву ***кліткові автомати***. Істотною особливістю кліткових автоматів є їх повна дискретність і, отже, пристосованість до точного моделювання на цифровому комп'ютері. Кліткові автомати були вперше розглянуті фон Нейманом та Уламом в 1948 р. як можлива ідеалізація біологічного самовідтворення. Зацікавленість ними обумовлена, головним чином, тим, що більшість з них є прикладами простих динамічних систем, які дають впорядковані візерунки, що виникають із випадкових початкових умов. Можна впевнитися в тому, що просторові візерунки багатьох кліткових автоматів нагадують візерунки, які можна спостерігати в природних явищах (поширений приклад – ріст кристалів). Саме тому кліткові автомати розглядають як корисні й цікаві комп'ютерні моделі, які зберігають привабливі можливості для опису складних фізичних систем.

Кліткові автомати являють собою математичні ідеалізації фізичних систем, в яких простір і час дискретні, самі ж фізичні величини (у разі потреби) набувають кінцевої множини дискретних значень. Для прикладу уявимо собі регулярну решітку клітин (комірок), кожна з яких може знаходитися у скінченому числі можливих станів, наприклад, 0 або 1. Стан системи повністю визначається значеннями змінних в кожній клітині.

Важливими особливостями кліткових автоматів є такі:

1. Стан кожної комірки поновлюється за скінчену послідовність кроків (зокрема, на кожному кроці).
2. Ці поновлення значень змінних в кожній комірці відбуваються одночасно («синхронно»), виходячи із значень змінних на попередньому кроці.
3. Новий стан комірки залежить лише від локальних значень у сусідніх комірках.

Кліткові автомати є прикладом появи і розвитку «комп'ютерної культу-

ри» і відбивають вплив техніки на спосіб людського мислення.

У більш пізніх роботах, присвячених клітковим автоматам, дослідження набули дещо іншого напряму. Було з’ясовано, що кліткові автомати здатні слугувати безпосередніми моделями багатьох фізичних систем – від сніжинок до феромагнетиків та Галактики.

Для характеристики кліткового автомата зазначимо такі його особливості:

1. Розташування клітин утворює деяку геометричну фігуру. Для моделі росту сніжинок досить двомірної шестикутної системи, але у більшості інших випадків обирають прямокутну решітку, що складається з квадратів. Існують тримірні схеми (і навіть з більшою кількістю вимірів, але уявити їх складно).
2. За заданою схемою необхідно визначити те оточення, яке дана клітина «вивчає» при обчисленні свого наступного стану.
3. Кількість станів, які може приймати клітина, буває різною. Фон Нейман побудував систему, здатну до самовідтворення, у якій клітини мали 29 можливих станів, однак більшість автоматів значно простіші.
4. Головне джерело змін у світі кліткових автоматів – це величезна кількість можливих правил для визначення наступного стану клітини, виходячи із станів її сусідів у даний момент.

Всі досліджені правила зміни стану комірки можна поділити на чотири класи:

1) правила, за яких еволюція приводить систему до стійкого та однорідного стану;
2) правила, що ведуть до появи простих структур (стійких або періодичних), які в будь-якому випадку залишаються ізольованими одна від одної;
3) правила, які ведуть до появи хаотичних візерунків (хоча й не обов'язково випадкових);
4) правила, які породжують структури істотної просторової та часової складності.

Мабуть, найвідомішим клітковим автоматом є гра «Життя», запропоно-

вана у 1970 р. відомим алгебраїстом, професором Кембріджського університету Джоном Хортоном Конвеєм. Та навряд чи вона отримала таке поширення, якби не книги відомого популяризатора науки Мартіна Гарднера, у якій вперше вона була представлена широкому загалу.

Ситуації, що виникають у процесі гри, дуже нагадують реальні процеси, що відбуваються при зародженні, розвитку та загибелі колонії організмів. Умови народження та загибелі визначаються виключно взаємним розташуванням учасників, а правила гри жорстко визначають, де та коли відбуваються народження та смерть.

Гра відбувається на нескінченому полі, розкресленому на квадратики. У грі беруть участь фішки, які на початку гри розташовуються на ігровому полі у деякому порядку. Цей порядок дуже важливий – саме він і визначає подальшу поведінку фігур – їх еволюцію. Кожна клітка поля може або залишатися порожньою, або бути зайнятою. Гра складається з «циклів життя», або з послідовності дискретних кроків, за допомогою яких імітується зміна поколінь.

У інтерпретації Гарднера, «Життя» – це живе співтовариство конвіків, що населяють Навітрені острови. Кожний такий острів складається з прямокутних ділянок. Ділянка може мати хазяїна – конвіка. З плином часу відбувається смерть та народження конвіків, тобто зміна поколінь. Нове покоління одержується з попереднього за такими правилами:

Виживання	Конвік виживає та переходить у наступне покоління, якщо поряд з ним зайняті іншими конвіками 2 чи 3 сусідні клітинки.
Загибель	Конвік гине у випадку, якщо поряд зайнято більше трьох чи менше двох сусідніх клітинок. У першому випадку система надто перенаселена, у другому конвік гине від самотності.
Народження	Якщо пуста клітинка межує рівно із трьома конвіками, то у цій клітинці народжується новий конвік, тобто у наступній генерації ця клітка буде населеною.

Ці правила завжди відносяться до стану, який існує до початку хода. Усі зміни на ігровому полі відбуваються ніби одночасно, хоча і складаються з деякі-

лькох етапів. На першому етапі визначається, у яких клітинках будуть народження та які фішки будуть у наступній генерації зняті з дошки. На другому етапі задумане втілюється у життя – на відмічених полях з'являються нові суб'єкти еволюції, а фішки, які не витримали сурових законів гри, знімаються з дошки.

Будь-який об'єкт гри, як сукупність активних елементів, число та взаємне розташування яких змінюється від кроку до кроку, характеризується деякими числовими параметрами. Перш за все це – розмір, що визначає кількість одночасно присутніх конвіків на ігровому полі. Початковий об'єкт «Життя», природно, характеризується і початковим розміром.

Щоб підкреслити схожість штучної еволюції з природною, при описі гри використовують термін **популяція**. Тоді можна легко говорити як про *розмір* популяції як кількість особів на даному етапі її розвитку, так й про *потужність* популяції. Потужність – це одна з найважливіших характеристик популяції, яка характеризує запас її життєвих сил та визначається, як сума усіх елементів на протязі усіх циклів гри.

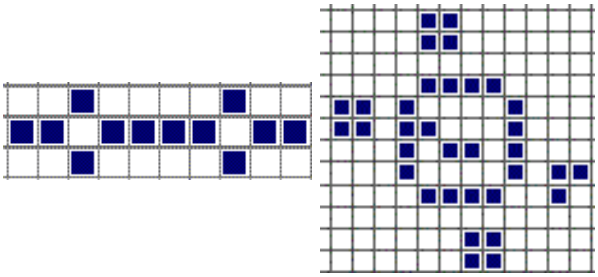
Будь-який об'єкт, що еволюціонує у грі, в решті решт припиняє свою еволюцію:

– популяція може повністю вимерти;
– популяція може деградувати до деякої сукупності стабільних об'єктів;
– популяція, крім фіксованих об'єктів, містить також пульсуючі об'єкти.

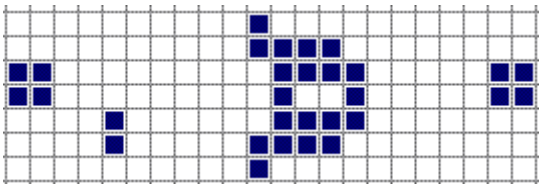
Будь-який з описаних станів для популяції вважається кінцевим. У зв'язку з цим для кожного об'єкта використовується поняття довжини життєвого циклу.

В результаті еволюції виникає багато стабільних об'єктів, деякі з яких зустрічаються настільки часто, що навіть отримали власні імена – це досить зручно для опису результатів еволюції. Назви та форму об'єктів розміру 4–7 наведено на малюнку нижче:

Існує величезна кількість пульсуючих об’єктів, починаючи від найпростіших (лінійка з трьох елементів), до дуже складних, які в принципі можуть будуватися у вигляді комбінацій з простих осцилюючих об’єктів. Приклади таких об’єктів – пентадекатрона та годинника – наведено на малюнку нижче:

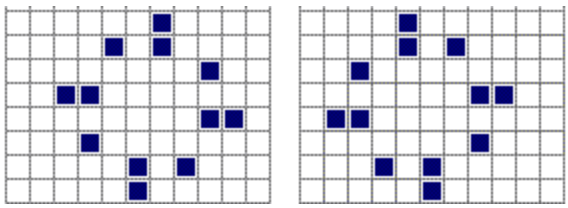


Ще одна фігура, що отримала назву «бджолина матка», є об’єктом, що складається з двох «блоків» та розташованої між ними деякої фігури:

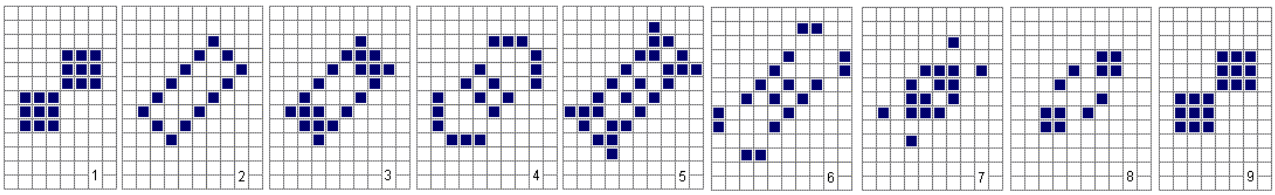


У процесі еволюції це «дещо» переміщується у просторі між «блоками», весь час змінюючи свою форму та почергово наближуючись то до одного, то до іншого «блоку». Об’єкт є осцилюючим (період 30) та відноситься до класу човників (shuttle). За визначенням, човник – це осцилюючий об’єкт, внутрішня частина якого циркулює між границями об’єкта туди-сюди.

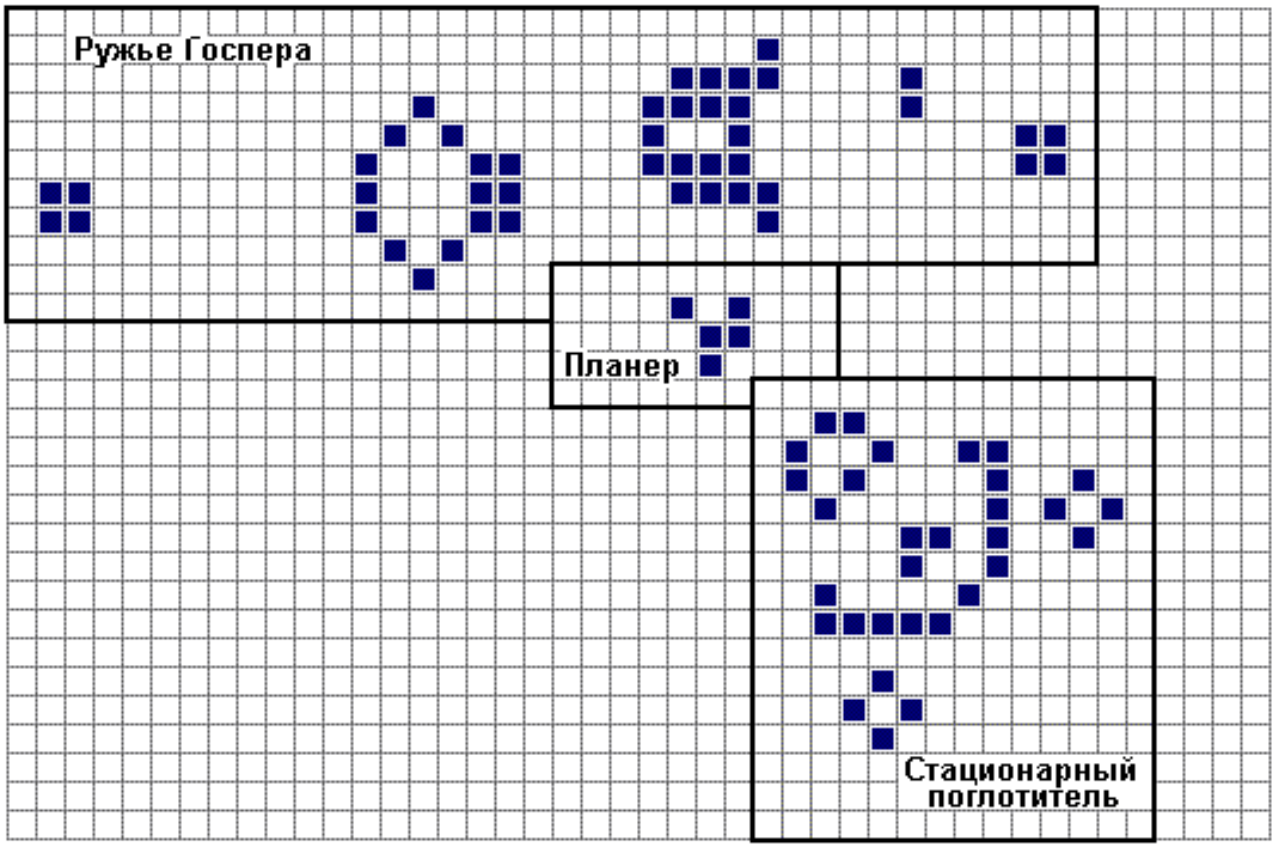
На малюнку нижче представлена фігура, яка відноситься до класу фенікс: легендарних птиць, що відроджувалися з попелу. На кожному новому кроці попередній об’єкт повністю зникає, далі з’являється знову, але у зміненому вигляді (дзеркальному). І далі відбувається повернення до вихідного стану – відродження фенікса. У динаміці можна спостерігати «обертання» фігури навколо своєї вісі:



На наступному малюнку подано повний цикл перетворень об’єкта під назвою «великий бакен» (big beacen), період якого складає 8.

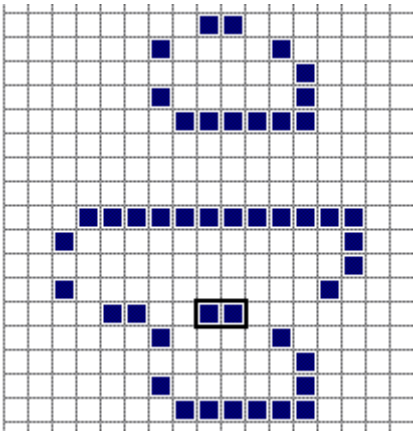


Дуже цікавими об’єктами є «планери», «планерні рушниці» та «планеровинищувачі»:

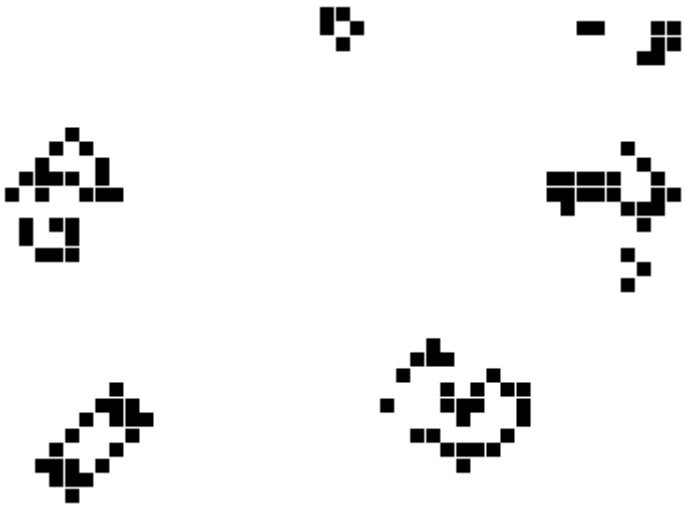


Одним з класів об’єктів є так звані космічні кораблі (spaceship). Космічний корабель – це об’єкт гри, який відтворюється чи з’являється знов через деяке число поколінь, однак при цьому змінює своє просторове положення у деякому напрямі, тобто рухається (транслюється).

Конвей виявив, що довгим кораблям, які він назвав надважкими, для їх стійкості потрібен ескорт з двох чи більше кораблів менших розмірів:



Проте експериментувати самому набагато цікавіше, ніж читати про чужі досліди, тому пропонуємо скласти просту, компактну та швидку програмну реалізацію «Життя». Задавати ігровий простір можна як в файлі, так і інтерактивним редагуванням. У консольному варіанті можна використовувати пересування за допомогою курсорних клавіш, примусово змінити характеристику населеності комірки можна, наприклад, нажавши клавішу Space, клавіша Enter ініціюватиме одноразову зміну поколінь за правилами «Життя», а завершує роботу програми клавіша ESC. Ту ж саму роботу можна виконати і за допомогою миші. Типовий результат роботи клітkового автомату «Життя» за цією програмою зображено на малюнку нижче:



§3.

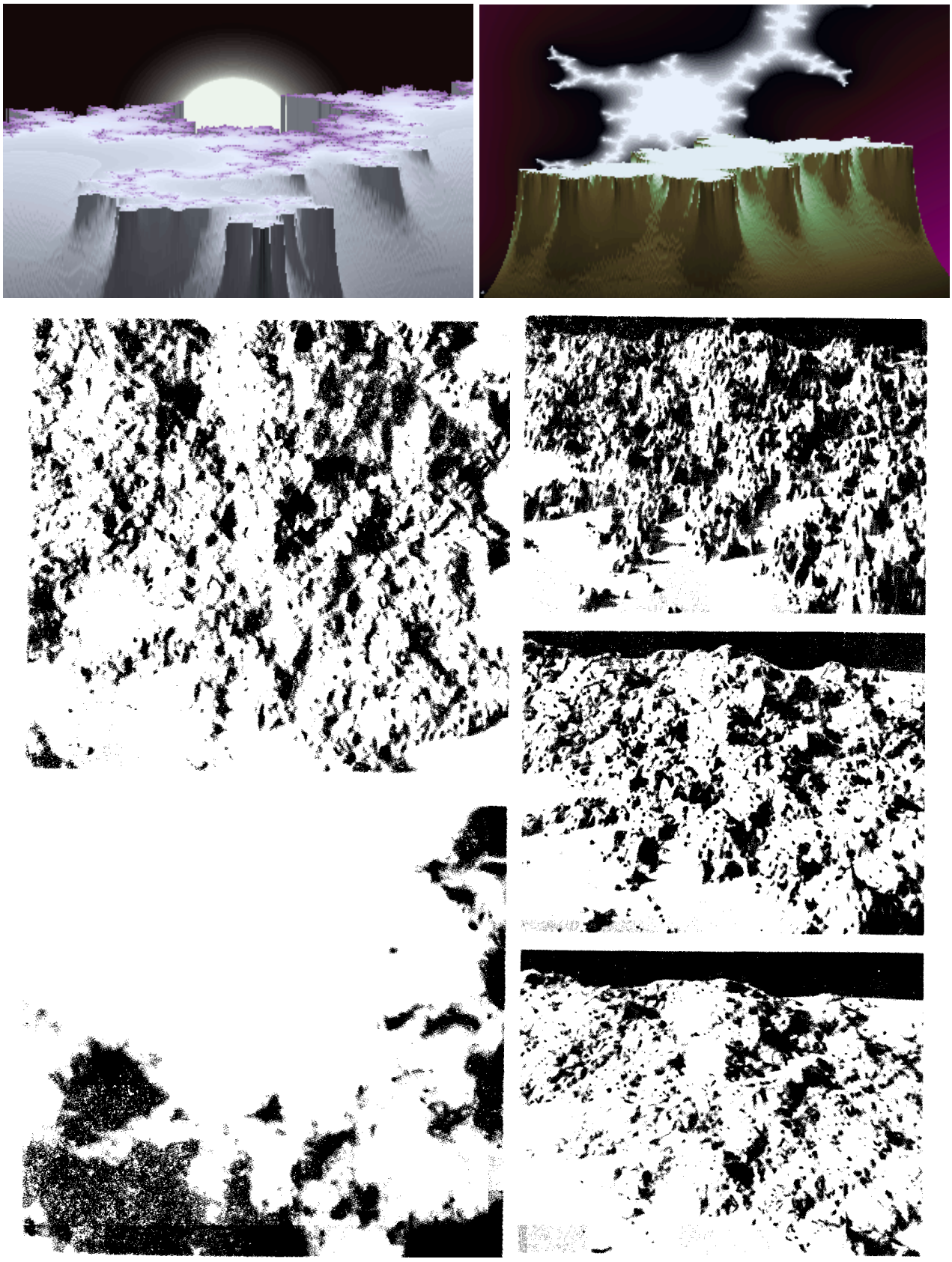
Фрактали

Геометричні властивості найрізноманітніших об'єктів природи – від масштабів атомного ядра до Всесвіту – займають центральне місце в моделях, що їх будують дослідники з метою «зрозуміти» природу. Геометрія траєкторій руху частинок, гірських ланцюгів, берегових ліній, островів, басейнів рік, зерен у скельних породах і металах та композитних матеріалах, комах і живих клітин, геометрична структура кристалів та молекул – коротше кажучи, геометрія природи займає центральне місце у різних галузях природознавства. Традиційно у основі інтуїтивного уявлення і розуміння цієї геометрії були евклідові прямі, кола, сфери, тетраедри тощо. Кожна наука намагалася розвинути свої власні поняття, пристосовані до її потреб: наприклад, у біологів – морфологія, у фізиків – чотиривимірний простір і т.п.

У 1975 році з'являється перша, а у 1982 році – друга книга бельгійського математика Бенуа Б. Мандельброта (зараз працює у США) під назвами «Фрактали: форма, випадок, розмірність» та «Фрактальна геометрія Природи», які збуджують загальний інтерес до фрактальної геометрії. Зокрема, він повідав світові про об'єкти, які він назвав фракталами. Поняття це було введено самим Мандельбротом і походить від латинського *fractus* – дробовий, посічений.

Поява у останні десятиліття ефективної комп'ютерної графіки призвела до інтенсивних досліджень нетрадиційних геометричних об'єктів у багатьох галузях природничих наук. Поняття фракталу полонило уяву вчених, що працюють у багатьох галузях науки, і роботи, у яких фрактали обговорюються з різноманітних позицій, як на погляд спеціалістів, з'являються тепер майже щоденно. Важливо, що на сьогодні єдиного й строгого означення фракталу не існує – можливо, внаслідок його об'єктивної складності та універсальності, а можливо – внаслідок молодості самої фрактальної геометрії. Принаймні, те, що існує, виявляється складним і не завжди достатньо чітким та однозначним. Такого стану у різні часи зазнала кожна наука.

Найбільш глибоке враження справляють комп’ютерні кольорові ілюстрації фрактальних об’єктів. Розглядаючи їх, важко уявити, що це не фотографії реальних ландшафтів, хмар, морських узбереж, вирощених кристалів тощо:



Пропонований до уваги матеріал призначено для ознайомлення з понят-

тям фракталу та практичного дослідження властивостей фрактальних об'єктів з використанням комп'ютера.

Останнім часом опис і вивчення таких об'єктів здійснюється в рамках нової – фрактальної геометрії. Кількісною мірою структурності цих об'єктів є так звана фрактальна розмірність D_f. З метою дати означення фрактальної розмірності D_f пригадаємо спочатку деякі поняття звичайної евклідової геометрії. Розглянемо круговий або сферичний об'єкт масою M і радіусом R. Він може бути або суцільним (однорідна густина), або містити порожнини, але у будь-якому випадку ми припустимо, що його густина не залежить від розміру.

Отже, при збільшенні радіусу об'єкта від R до $2R$ його маса збільшуватиметься у R^2 разів, якщо об'єкт коловий, або у R^3 разів, якщо він сферичний. Цей зв'язок маси й розміру можна подати у вигляді

$$M(R) \sim R^D, \quad (1)$$

де D – розмірність простору. Об'єкт, у якого маса й розмір пов'язані співвідношенням (1), зветься «компактним». Дане співвідношення означає, що при збільшенні лінійного розміру об'єкта у R разів за незмінної форми його маса збільшуватиметься у R^D разів. Це масштабне співвідношення маса – розмір тісно пов'язане з інтуїтивним уявленням про розмірність. Воно також є корисним узагальненням на розмірності, більші трьох, або такі, що не є цілими числами.

Зв'язок між масою об'єкта та його характерним розміром R можна визначати у більш загальному вигляді, ніж за формулою (1). На цьому засновано одне з формулювань означення фрактальної розмірності

$$M(R) \sim R^{D_f}. \quad (2)$$

Об'єкт називають «фрактальним», якщо він задовольняє співвідношенню (2) зі значенням D_f, меншим за просторову (евклідову) розмірність d.

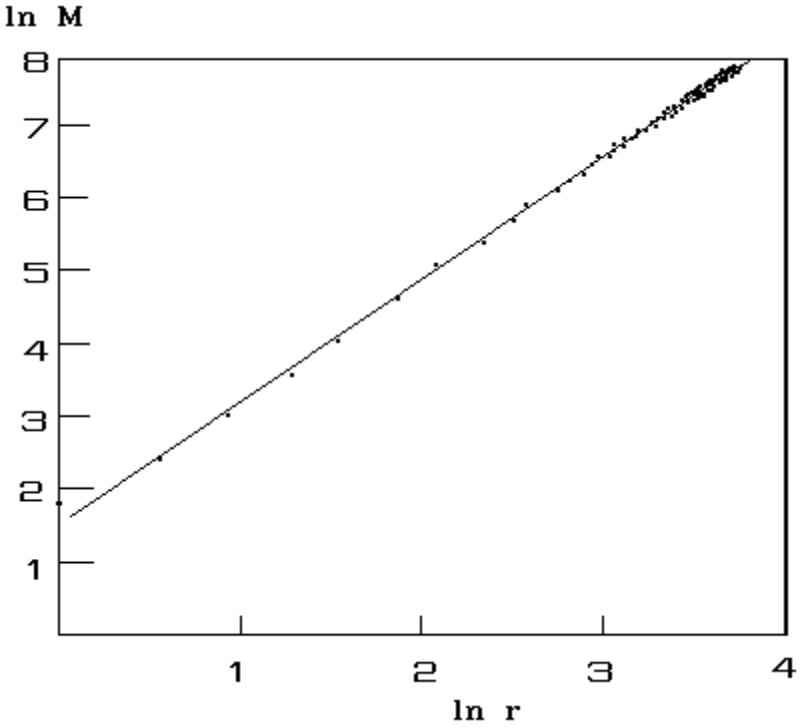
Якщо для об'єкта виконується співвідношення (2), то його густина не буде однаковою для всіх значень R, а масштабується у такий спосіб:

$$\rho(R) \sim \frac{M}{R^D} \sim R^{D_f - D}. \quad (3)$$

Оскільки $D_f < D$, то густина фрактального об'єкта зменшується із

збільшенням його розміру. Дана масштабна залежність (3) густини є кількісною мірою уявлення про фрактали як про розріджені або розгалужені об’єкти. Інший метод опису фрактального об’єкту засновано на припущенні, що фрактальний об’єкт містить порожнини всіх розмірів.

Цей факт добре проглядається на перколяційному кластері, де порожнини являють собою прохідні та непрохідні комірки. Такий перколяційний кластер є прикладом випадкового або статистичного фракталу, оскільки для нього, як виявляється, співвідношення (1) маса–розмір виконується лише «у середньому», наприклад, при усередненні значень $M(R)$ для великої кількості кластерів за умови, що беруться різні початкові точки відліку в середині кластеру. На малюнку нижче у подвійному логарифмічному масштабі зображено залежність *маса–розмір* для перколяційного кластеру, згенерованого на квадратній сітці 61x61 комірок. Як стане видно з подальшого викладу, наш метод визначення фрактальної розмірності таких об’єктів дещо відрізняється від зазначеного вище.

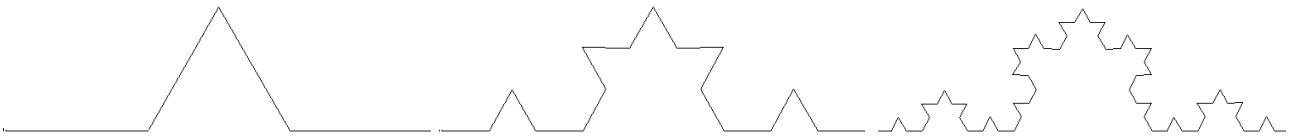


В усіх реальних фізичних системах співвідношення (3) виконується не для будь-яких масштабів довжини, а обмежується верхньою та нижньою границями. Наприклад, нижня границя довжини обумовлюється тією чи іншою мак-

роскопічною відстанню (період решітки, відстань між складовими частинами об'єкту тощо). У чисельному моделюванні верхня границя довжини звичайно обумовлюється кінцевим розміром системи. Наявність зазначених границь ускладнює обчислення фрактальної розмірності.

Зміст виразу $M(R)\sim R^{D_f}$ полягає ще й у тому, що фрактальні об'єкти є самоподібними: вони мають однаковий вигляд у любому просторовому масштабі. Образно кажучи, їх вигляд не повинен змінитися, коли ми подивимося на них неозброєним оком, а потім крізь лупу. Для з'ясування суті самоподібності звернемося до деяких прикладів регулярних фракталів – об'єктів, що є самоподібним за усіх масштабів довжини.

Тріадна крива Кох. Наведемо послідовність операцій (алгоритм) для побудови цієї кривої. Починаємо з відрізку одиничної довжини. Розділимо його на три рівні частини і видалимо середню, яку замінимо на два відрізки довжиною 1/3 кожен так, що на кривій з'являється трикутний горб. При цьому довжина кривої стає рівною 4/3. На наступному кроці кожен сегмент довжиною 1/3 ділиться на три частини довжиною 1/9 і процедура повторюється. Довжина кривої стає рівною 16/9:



Якщо уявити собі, що зазначена процедура буде повторюватися нескінченну кількість разів, то повинна одержатися крива нескінченної довжини, що складається з нескінченної кількості нескінченно малих відрізків. Така крива має назву «тріадна крива Кох».

Для визначення фрактальної розмірності цієї кривої звернемося до дещо простішої ситуації. Розглянемо спочатку відрізок одиничної довжини, поділений на N однакових частин довжиною d кожна, так що $N=1/d$. У міру зменшення d значення N зростатиме лінійно, як і слід чекати для одномірної кривої. Аналогічно, якщо поділити квадрат одиничної площі на N рівних квадратиків зі стороною d, то одержимо $N=1/d^2$ – очікуваний для двомірного об'єкту результат. Можна стверджувати, що у загальному випадку

$$N=1/d^D,$$

тут D – розмірність об’єкту. Після логарифмування обох частин цієї рівності можна подати розмірність у знайомому вигляді

$$D_f = \frac{\ln N}{\ln \frac{1}{d}}.$$

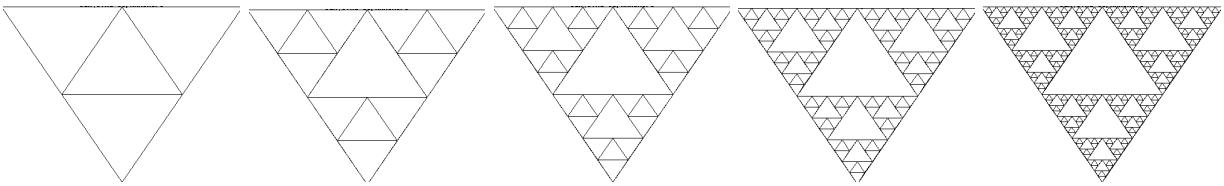
Застосуємо ці міркування до кривої Кох. Ми знайшли, що при кожному зменшенні довжини D нашої одиниці виміру у три рази кількість сегментів збільшується у чотири рази. Таким чином, маємо N=4, D=1/3.

Отже, фрактальна розмірність тріадної кривої Кох дорівнює

$$D_f=\ln 4/\ln 3=1{,}26186\ldots$$

Таким чином, можна стверджувати, що *крива Кох вже не є одномірною, але ще й не двомірна*. Виявилося, що криві, подібні до кривої Кох, у природі складають скоріш правило, аніж виключення. Такими, зокрема, є морські узбережжя або видимі обриси хмар.

Серветка та килим Серпінського. Серветка Серпінського являє собою побудову, що породжує криву з петлями усіх розмірів. Затравкою є рівнобічний трикутник з усіма внутрішніми точками. Утворюючий елемент виключає із затравки центральний трикутник, сторона якого вдвічі менша за сторону попереднього, тобто при кожному наступному кроці трикутник разом з внутрішніми точками (зафарбований) замінюється N=3 трикутниками, зменшеними з коефіцієнтом d=1/2. На малюнку подано перші п’ять поколінь:

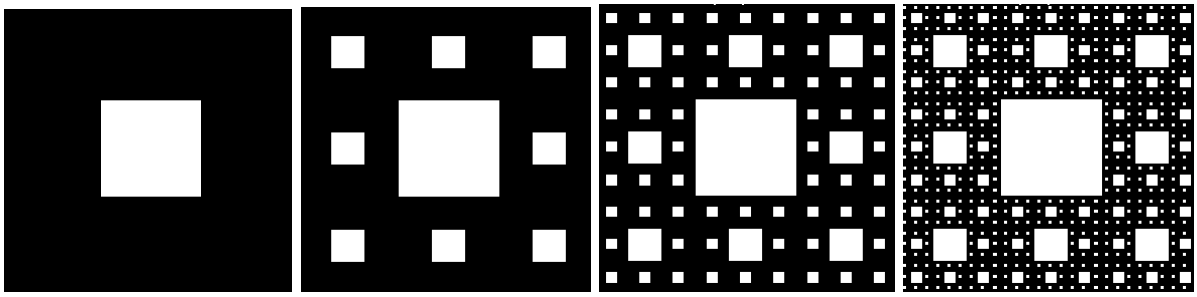


Для обчислення фрактальної розмірності серветки Серпінського знов скористаємося співвідношенням

$$D_f = \frac{\ln N}{\ln \frac{1}{d}},$$

де N=3, D=1/2. Одержуємо D_f=ln 3/ln 2=1,58…

З серветкою Серпінського тісно пов’язана інша крива – так званий килим Серпінського. Затравкою тут є квадрат, а утворюючий елемент складається з N=8 квадратів, одержаних із затравки стисненням з коефіцієнтом подібності d=1/3. Нескінченно багато поколінь породжують фрактальну криву. При переході до граничної фрактальної кривої чорні ділянки зникають, а повний периметр «дір» у килимі Серпінського стає нескінченним.



Фрактальна розмірність цього об’єкту D_f=ln 8/ln 3=1,89…

Криві Серпінського часто використовують при моделюванні багатьох фізичних явищ. Зокрема, при дослідженнях деяких властивостей металевих плівок їхню структуру подавали у вигляді серветки Серпінського 10-го покоління: тим самим забезпечувалися достатньо малі відстані між структурними елементами. Результати такого моделювання на диво гарно узгоджуються з теоретичними передбаченнями.

§4.

Подібність та скейлінг

Пряма – особлива множина точок у просторі: при будь-якій зміні масштабу ми одержимо ту ж саму множину точок. Крім того, виконавши над прямою паралельний перенос, ми знову одержимо ту ж саму множину точок. Пряма *інваріантна* щодо паралельного переносу та зміни масштабу, чи скейлінга, – можна сказати, що пряма самоподібна.

Уточнимо наше твердження. Задамо точки в просторі їхніми декартовими координатами $\mathbf{x}=(x_1, x_2, x_3)$. Пряма, що проходить через точку $\mathbf{x}_0$ у напрямку $\mathbf{a}=(a_1, a_2, a_3)$, є множина точок $\mathcal{L}$, обумовлене співвідношенням

$$\mathbf{x}=\mathbf{x}_0+t\mathbf{a},\ -\infty<t<+\infty.$$

Параметр t тут будь-яке дійсне число. Якщо змінити масштаб довжини в те саме число раз r для всіх компонентів радіуса-вектора $\mathbf{x}$, точки $\mathbf{x}$ відобразяться в нові точки $\mathbf{x}'=r\mathbf{x}=(rx_1, rx_2, rx_3)$, і ми одержимо нову множина точок $r(\mathcal{L})$, обумовлене співвідношенням

$$\mathbf{x}'=r(\mathbf{x}_0+t\mathbf{a})=\mathbf{x}_0+t'\mathbf{a}-(1-r)\mathbf{x}_0.$$

Тут $t'=rt$ знову будь-яке дійсне число. Якщо зрушити нову множину точок $r(\mathcal{L})$, піддавши всі його точки паралельному переносу на величину $(1-r)\mathbf{x}_0$, то в результаті ми одержимо вихідну множина точок $\mathcal{L}$: пряма інваріантна щодо зміни масштабу довжини. Пряма інваріантна і щодо паралельного переносу $\mathbf{x}\to\mathbf{x}+\mathbf{a}n$, де n – будь-яке дійсне число.

Як показують аналогічні міркування, площина інваріантна щодо паралельних переносів у будь-якому напрямку, що лежить у ній самій, і щодо зміни масштабів довжини. Нарешті, тривимірний простір інваріантний щодо паралельних переносів у будь-якому напрямку і щодо зміни масштабів довжини.

Інші множини точок не мають настільки міцні симетрії – інваріантості щодо паралельних переносів і скейлінга. Окружність не інваріантна ні відносно паралельного переносу, ні відносно скейлінга, а інваріантна щодо поворотів на-

вколо власного центра. Фрактали також не мають властивості деяких чи навіть усіх цих простих інваріантостей.

Корисно розглянути обмежені множини, такі, як скінченний відрізок прямої. Відрізок прямої не має трансляційну симетрію – будь-яке зсув його завжди породжує нову множину точок. Але якщо змінити довжини в r раз, де $r<1$, то отримаємо нову множину точок $\mathcal{L}'=r(\mathcal{L})$, що складатиме невелику частину прямої. Цим відрізком прямої, піддавши його паралельному переносу, можна покрити частину вихідного прямолінійного відрізка $\mathcal{L}$. При належному виборі числа r ми можемо однократно покрити вихідний відрізок N відрізками, що не перетинаються. Можна сказати, що множина $\mathcal{L}$ *самоподібна* з коефіцієнтом подоби r. Для відрізка прямої одиничної довжини ми можемо вибрати $r(N)=1/N$, де N – будь-яке ціле число. Прямокутну ділянку площини можна покрити її зменшеними копіями, якщо їхні довжини змінити в $r(N)=(1/N)^{1/2}$ разів. Аналогічно прямокутний паралелепіпед можна покрити його зменшеними копіями, якщо вибрати $r(N)=(1/N)^{1/2}$. У загальному випадку масштабний множник варто вибирати рівним

$$r(N)=(1/N)^{1/d}$$

Розмірність подоби d для прямих, площин і кубів дорівнює відповідно 1, 2 і 3.

Розглянемо тріадну криву Кох з попередньої роботи. З масштабним множником $r=1/3$ ми одержуємо першу третину всієї кривої. Нам необхідно $N=4$ таких фрагментів, щоб покрити вихідну множину його зменшеними копіями, піддаючи їх повторним паралельним переносам та поворотам. Ми можемо також вибрати масштабний множник $r=(1/3)^n$ і покрити вихідну множину його $N=4^n$ зменшеними копіями. Як було показано, для тріадної кривої Кох масштабний множник визначається виразом

$$r(N)=(1/N)^{1/D}$$

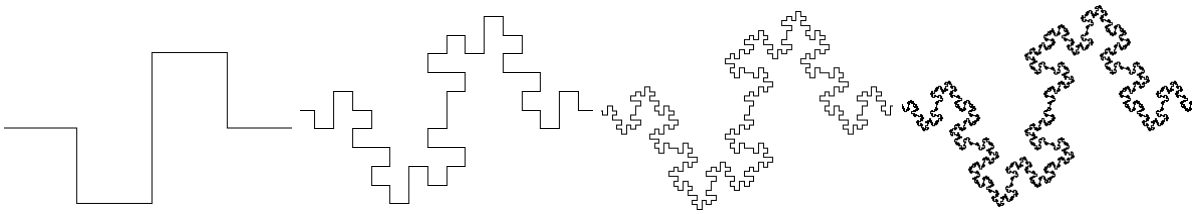
з розмірністю подоби d, рівній розмірності Хаусдорфа-Безіковича $D=\ln 4/\ln 3$.

У загальному випадку розмірність подоби D_S визначається виразом

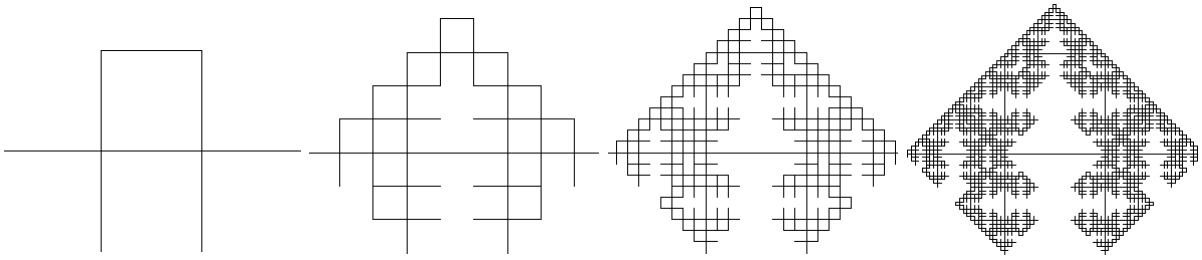
$$D(S)=-\ln N/\ln r(N).$$

Для самоподібних фракталів розмірність Хаусдорфа-Безіковича D дорівнює D_S, і для таких фракталів ми будемо опускати індекс S у розмірності подоби.

Розмірність подоби легко піддається визначенню для різних фракталів, що виходять за допомогою різних варіантів побудови Кох. Розглянемо передфрактал Кох, побудований з одиничним відрізком в якості затравки і з утворюючим елементом , що складається з N=8 ламаних довжиною r=1/4, зображених на малюнку. Ця крива має розмірність подоби $D=-\ln 8/\ln 1/4 = 3/2$ і дорівнює розмірності Хаусдорфа-Безіковича множини, що виходить після нескінченно великого числа ітерацій. Помітимо, однак, що фігура в цілому не витримує перетворення подоби. Справа в тім, що фрактальна скейлінгова інваріантість досягається тільки в межі при $\delta\to 0$, і ми робимо висновок, що фрактальна природа кривих Кох є, строго говорячи, *локальна властивість*.



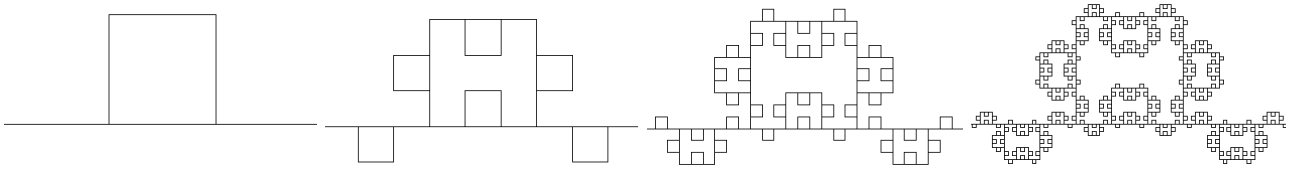
Побудова кривих Кох, зображених нижче, належить Мандельброту та Гівену. Утворюючий елемент для цієї кривої поділяє прямолінійний відрізок на частині довжиною r=1/3 і з'єднує їх у петлю, що складається з трьох частин, до якої прилаштовуються дві вітини.



Мандельброт і Гивен використовували цю криву й аналогічні криві як моделі перколяційних кластерів. Ця крива цікава тим, що має петлі всіх можливих розмірів і виступи усіх можливих розмірів. І виступи, і петлі декоровані петлями і виступами і т.д. При кожній ітерації (переході від одного покоління

передфракталів до наступного) утворюючий елемент робить заміну кожної прямолінійної ланки в передфракталі на N=8 ланок, зменшених з r=1/3. Використовуючи формулу для розмірності подоби, робимо висновок, що крива Мандельброта-Гівена має фрактальну розмірність D=ln 8/ln 3=1,89….

Уявимо криву Мандельброта-Гивена, виготовлену з якого-небудь електропровідного матеріалу, і нехай струм тече від лівого кінця кривої до правого. Ясно, що в жодній вітині, що виникає з двох вертикальних відрізків утворюючого елемента, струму не буде. Тік буде текти тільки по кістяку – по кривій, що вийде, якщо від кривої Мандельброта-Гивена відітнути всі вітини, з’єднані з вихідним прямолінійним відрізком (затравкою) тільки одним зв'язком. Відкинувши всі вітини, ми одержимо криву, зображену на малюнку. Фрактальна розмірність такої кривої без вільних (“висячих”) кінців дорівнює D_B=ln 6/ln 3=1,63..., тому що утворюючий елемент заміняє кожен прямолінійний відрізок N=6 відрізками, зменшеними (r=1/3) копіями замінного відрізка.



У скількох місцях ми можемо перетинати ординарний (однозв’язний) зв’язок, щоб кінці затравки виявилися роз’єднаними? Щоразу, застосовуючи утворюючий елемент, ми породжуємо N=2 однозв’язних зв’язків, тому ці зв’язки утворять множину точок із фрактальною розмірністю D_{SC}=ln 2/ln 3=0,63...

§5.

Метод Хаффмана

Стискання скорочує обсяг простору, необхідного для збереження файлів в ЕОМ, і кількість часу, необхідного для передачі інформації з каналу встановленої ширини. Це є форма кодування. Іншими цілями кодування є пошук і виправлення помилок, а також шифрування. Процес пошуку і виправлення помилок протилежний стисканню – він збільшує надмірність даних, коли їх не потрібно представляти в зручній для сприйняття людиною формі.

Видаляючи з тексту надмірність, стиск сприяє шифруванню, що утрудняє пошук шифру доступним для зломщика статистичним методом.

У цій роботі ми розглянемо оборотний стиск чи стиск без наявності перешкод, де первісний текст може бути в точності відновлений зі стиснутого. Необоротний стиск використовується для цифрового запису аналогових сигналів, таких як людська мова чи малюнки. Оборотний стиск особливо важливий для текстів, записаних на природних і на штучних мовах, оскільки в цьому випадку помилки звичайно неприпустимі. Хоча першочерговою областю застосування розглянутих методів є стиск текстів, що відбиває і термінологія, однак, ця техніка може знайти застосування й в інших випадках, включаючи оборотне кодування послідовностей дискретних даних.

Існує багато вагомих причин виділяти ресурси ЕОМ у розрахунку на стиснуте представлення, тому що більш швидка передача даних і скорочення простору для їхнього збереження дозволяють зберегти значні обсяги і найчастіше поліпшити показники ЕОМ. Стиск, ймовірно, буде залишатися в сфері уваги через усе зростаючі обсяги збережених і переданих в ЕОМ даних, крім того, його можна використовувати для подолання деяких фізичних обмежень.

Одне й те саме повідомлення можна закодувати різними способами і при цьому виникає питання про найвигідніші (оптимальні) способи кодування. Можна, наприклад, вважати найкращим такий код, при якому на передачу повідомлень затрачається мінімальний час. Якщо на передачу елемента повідомлення

– двійкового числа 0 чи 1 – витрачається однаковий час, то оптимальним буде такий код, що на передачу повідомлення заданої довжини вимагає найменшої кількості елементів (нулів і одиниць). Нехай нам потрібно закодувати двійковим кодом букви російської абетки так, щоб кожній букві відповідала унікальна комбінація одиниць і нулів і щоб середня кількість цих елементів на одну букву тексту було мінімальним. В алфавіті 32 букви + пробіл між словами. Найпростіше рішення – перенумерувати їх від 0 до 31 із записом у двійковій системі числення. Домовимося для простоти не розрізняти, наприклад, “ь”, щоб разом із пробілом число букв було дорівнює найближчого ступеня двійки – числу $2^5=32$. Тоді кожна буква може бути зображена п’ятизначним двійковим числом:

а ~ 00000
б ~ 00001
в ~ 00010
г ~ 00011
...
я ~ 11111

При такому способі на кожну букву витрачається по 5 елементів. Подумаємо, чи не можна скласти код, у якому в середньому на 1 букву буде витрачатися менша кількість елементів кодування. Для цього проаналізуємо, з якою частотою в російському тексті зустрічаються різні букви і спробуємо ті букви, що часто зустрічаються, кодувати меншим числом елементів, а ті, що рідше зустрічаються – великим. Відносні частоти букв у тексті в порядку убування, визначені за множиною різних текстів від романів Лева Толстого до постанов уряду разом з отриманими кодами виглядають так:

Буква	*Частота*	*Код*
пробіл	0.145	000
о	0.095	001
е	0.074	0100
а	0.064	0101
и	0.064	0110

Буква	*Частота*	*Код*
т	0.056	0111
н	0.056	1000
с	0.047	1001
р	0.041	10100
в	0.039	10101
л	0.036	10110
к	0.029	10111
м	0.026	11000
д	0.026	110010
п	0.024	110011
у	0.021	110100
я	0.019	110110
ы	0.016	110111
з	0.015	111000
ь,ъ	0.015	111001
б	0.015	111010
г	0.014	111011
ч	0.013	111100
й	0.010	1111010
х	0.009	1111011
ж	0.008	1111100
ю	0.007	1111101
ш	0.006	11111100
ц	0.004	11111101
щ	0.003	11111110
э	0.003	111111110
ф	0.002	111111111

Максимальна інформація, що її передає 1 елемент повідомлення, дорівнює 1 біту (1 і 0 рівноймовірні). Тому в основу нашого оптимального кодування покладемо вимогу, щоб всі елементи кодів у тексті зустрічалися однаково часто. Для цього кодовані букви розділимо на 2 рівноймовірні групи: у першій групі на першому місці кодової комбінації ставимо 0, а в другий – 1. Далі кожну групу знову розділимо на 2 приблизно рівноймовірні підгрупи і для символів першої на другому місці поставимо 0, а для другої підгрупи – 1 і т.д. Результат кодування записаний у ту ж таблицю, де записані частоти букв. Спосіб кодування зветься *кодами Шеннона-Фено*. За допомогою цієї таблиці можна закодувати і декодувати будь-яке повідомлення, наприклад фраза “теорія інформації” буде виглядати після кодування так:

01110100001101000110110000011010001111111110011010011000010111111101011001 10

Тут немає необхідності відокремлювати букви друг від друга спеціальним знаком – декодування виконується однозначно і без цього (але будь-яка помилка переплутування 1 і 0 буде фатальною – наступний за помилкою текст уже не може бути декодований правильно).

Щоб переконатися, декодуйте за допомогою таблиці фразу:

1001110011001100100111101000010111001110010011010100001101010101100001101101 10

(“спосіб кодування”).

Статичне *кодування Хаффмана* засноване на кодах Шеннона-Фено, але припускає, що від файлу до файлу частоти символів міняються. Перевагами методу Хаффмана є його досить висока швидкість і гарна якість стиску. Цей алгоритм порівняно давно відомий і широко застосовується; прикладами можуть служити програма compact ОС UNIX (програмна реалізація) і стандарт кодування для факсів (апаратна реалізація).

Кодування Хаффмана має мінімальну надмірність за умови, що кожен символ кодується окремим ланцюжком в алфавіті {0, 1}. Недоліком кодування Хаффмана є залежність ступеня стиску від близькості ймовірностей символів до від’ємних ступенів 2; це пов’язано з тим, що кожен символ кодується цілим числом бітів.

Інформацію про кожний кодований байт будемо зберігати в записі з наступними полями:

```
type stat=record
 b:byte;              – ASCII-код символу
 count:longint;       – кількість повторень даного символу у файлі
 weight:real;         – питома вага кожного символу, дорівнює відношенню кількості повторень даного символу у файлі до загального числа символів, тобто довжині файлу
 code:string[234];    – рядок, що буде містити бітовий код для байта, лише для зручності записуваного в символьному виді
end;
```

А для всіх можливих кодів прийдеться організувати масив таких структур на 256 (0–255) елементів, що будемо надалі називати масивом статистики. Крім нього, нам знадобляться ще 2 файлові змінні (для вихідного і стиснутого файлів) для байтових файлів, 2 байти – початок і кінець області кодів, що зустрічаються в даному файлі (скажемо, якщо у файлі зустрічаються лише три букви, то з усього масиву статистики нам треба буде звертати увагу лише на три елементи, що відповідають цим символам; тоді початок буде рівним 0, а кінець – 2, індексу останнього елемента) і одне довге ціле для збереження довжини файлу. Зважаючи на те, що згодом нам доведеться записати його в байтовий файл, то для перетворення цього довгого цілого (подвійного слова) у байтову послідовність доведеться оголосити якийсь тип перетворення довгого цілого в масив з чотирьох байтів.

Крупноблочно програма кодування може виглядати так:

1. Ініціалізуємо таблицю статистики
2. Зберемо статистику про файл
3. Відсортуємо таблицю статистики за спаданням числа повторень
4. Визначаємо вагу кожного символу і довжину файлу
5. Обнулимо початок і завжди будемо тримати його таким

6. Отримаємо кінець – індекс у таблиці статистики першого невикористовуваного коду
7. Якщо кінець збігається з початком – робимо відповідні висновки і самоусуваємося від процесу кодування
8. Якщо кінець не дорівнює останньому індексу в таблиці статистики – числу 255, – то зменшуємо його, щоб мати в цій змінній індекс останнього використовуваного коду (після сортування за спаданням числа повторень наступний елемент у таблиці буде мати нульове число повторень і нульову вагу)
9. І викликаємо рекурсивну процедуру поділу таблиці статистики з початковими параметрами: початок, кінець, 1/2 (половина суми ваг усіх символів)
10. Розділену таблицю від початку і до кінця сортуємо за зростанням довжині коду
11. Відкриваємо другий файл для запису, а перший для читання
12. Записуємо заголовок архівного файлу (таблицю відповідностей виду символ – код)
13. Читаємо перший файл, записуючи коди в другий і
14. Закриваємо обидва файли

Тепер деякі з цих пунктів деталізуємо.

Ініціалізується таблиця відповідностей у циклі по всіх її елементах наступними початковими значення: у поле b заноситься поточний індекс елемента таблиці, поля кількості і ваги обнуляються, а в ряжкове поле коду заноситься порожній рядок (не пробіл).

Збираємо статистику так:

```
відкриваємо перший файл для читання
доки він не закінчився
    читаємо з нього один байт
    і інкрементуємо поле лічильника елемента масиву статистики,
    за індексом рівному зчитаному байту
після чого файл закриваємо
```

Сортування за спаданням числа зустрічей – звичайна бульбашка:

```
зовнішній цикл – від 0 до 255
    внутрішній цикл – починаючи з наступного елемента, що йде
    за індексом зовнішнього циклу і теж до кінця масиву статистики
        якщо в записі за індексом зовнішнього циклу поле лічильника
        менше того ж полючи, але за індексом внутрішнього циклу
            обмінюємо відповідні записи масиву
            статистики місцями
```

Вагу кожного символу і довжину файлу визначити дуже просто: спочатку довжина обнуляється, а потім ми проходимо по всій таблиці, нарощуючи її на відповідне поле статистичного запису. Потім ми повторно проходимо по всій таблиці, “зважуючи” кожен код поділом кількості його зустрічей у файлі на підраховану довжину. По закінченні цього процесу повернемо значення довжини в основну програму.

Як одержати індекс у масиві статистики останнього використовуваного символу? Просто пройдемось по всій таблиці і, якщо під час цих блукань зустрінемо запис, у якому поле числа повторень (чи ваги, що в даному випадку несуттєво) дорівнює нулю, то перервемося і повернемо індекс, на якому перервалися.

Процедура поділу таблиці навпіл, як ми вже домовлялися, буде мати три параметри – початок, кінець і коефіцієнт поділу. У процесі поділу нам доведеться визначати середину поточного інтервалу, такий новий кінець і додавати до кодів символів у визначеному діапазоні таблиці символи нуля й одиниці відповідно. Отже,

```
якщо початок і кінець збігаються
    додаємо код нуля від початку до кінця
    і виходимо
якщо початок і кінець розташовані поруч (тобто між ними
одинична різниця), то
    додаємо код нуля від початку до початку
```

```
    додаємо код одиниці від кінця до кінця
інакше (якщо відстань між початком і кінцем трохи більше)
    визначаємо новий кінець – середину поточного інтервалу в
    таблиці (від початку до кінця) за параметром “коефіцієнт поділу”
    якщо новий кінець і початок збігаються, то
        додаємо код нуля від початку до нового кінця
        додаємо код одиниці від наступного за новим кінцем
        елементу і до старого кінця інтервалу
        і викликаємо цю же процедуру з параметрами: наступний
        за новим кінцем елемент, старий кінець інтервалу
        і зменшений удвічі коефіцієнт поділу
    інакше
        якщо старий кінець збігається з новим, то
            додаємо код нуля від початку до елемента,
            попереднього старому кінцю
            викликаємо цю же процедуру з параметрами:
            початок, елемент, що передує старому кінцю
            і зменшений удвічі коефіцієнт поділу
            додаємо код одиниці від кінця до кінця
        інакше (якщо новий кінець не збігається ні з початком,
        ні зі старим кінцем)
            додаємо код нуля від початку до нового кінця
            додаємо код одиниці від елемента, що слідує
            за новим кінцем і до старого кінця
            викликаємо цю ж процедуру дворазово –
            с  параметрами: початок, новий кінець,
            і зменшений удвічі коефіцієнт поділу
            і з параметрами: елемент, що слідує за новим
            кінцем, старий кінець і зменшений удвічі
            коефіцієнт поділу
```

От і всі можливі варіанти для поділу. Звичайно, їх можна було б і не враховувати, а просто знайти середину, додати до середини нулі, після – одиниці і викликати дворазово саму себе з новими параметрами, але такий додатковий аналіз дозволяє сильно зменшити глибину рекурсії, що дає можливість обійтися навіть стандартним стеком.

Процедура додавання кодового символу як параметри може приймати початок і кінець поточної зони додавання символу і, звичайно, сам цей символ. У ній ми в заданому діапазоні індексів таблиці відповідностей до поля коду поточного запису приклеюємо переданий символ.

Як можна одержати середину поточного діапазону (“новий кінець”), заданого початком, кінцем і коефіцієнтом поділу? Просто заведемо якийсь сумator, і будемо його нарощувати в циклі по всьому заданому діапазоні на вагу поточного елемента і, як тільки цей суматор перевищить переданий коефіцієнт поділу – перериваємося і повертаємо поточний індекс як шукане значення.

Досортування за довжиною кодових послідовностей необхідnt тому, що після виконання процедури поділу ми одержуємо ненадлишковий код, але не завжди там, де потрібно (наприклад, коду з меншою вагою може відповідати менший, а не більший код), тому виникає необхідність виконати досортування за зростанням довжини кодових послідовностей, але при перевищенні довжини коду поточного запису по зовнішньому циклу довжини коду поточного запису по внутрішньому циклу змінювати у відповідних елементах таблиці відповідностей не запиу цілком (щоб не порушити частотну упорядкованість), а тільки строкове поле коду.

Тепер ми готові до завершального етапу – запису таблиці відповідностей символ-код і самих кодових груп для даного файлу. Як ми будемо записувати таблицю відповідностей? От так:

Спочатку нам доведеться записати подвійне слово, що відповідає довжині файлу. Перетворимо його в спеціальний тип, визначений раніше як чотирьохбайтовий масив, і заженемо ці байти один за одним у файл, а слідом за ними, не барячись, відправимо байт із кінцем (якщо ви вже остаточно заплутали-

ся, то в даному випадку кінець – це індекс останнього елемента масиву статистики з ненульовою вагою).

Далі, у циклі від початку до кінця

визначаємо довжину в бітах поточного коду

запишемо у файл поле b поточного запису і отриману довжину

скопіюємо код у якийсь тимчасовий рядок

якщо довжина не кратна восьми бітам, то

доповнюємо тимчасовий рядок до найближчої кратниої

вісімці довжини нулями

поки довжина тимчасового рядка число натуральне

перетворимо рядок у байт

записуємо цей байт у файл

і вирізаємо з рядка перші вісім символів

Перетворити рядок у байт елементарно. Наприклад, це можна зробити так

```
function Str2Byte(st:string):byte;
var
  i,b:byte;
begin
  b:=0;
  for i:=1 to 8 do
    b:=b+(ord(st[i])-ord('0')) shl (8-i);
  Str2Byte:=b;
end;
```

До неї зворотною буде така:

```
function Byte2Str(b:byte):string;
var
  st:string;
  i:word;
begin
  st:='';
```

```
  for i:=0 to 7 do
    st:=st+chr(byte(byte(b shl i) shr 7) + ord('0'));
  Byte2Str:=st;
end;
```

Нарешті, власне про кодування. Ми припускаємо, що перший файл відкритий для читання, а другий – для запису. Нам доведеться ще завести який-небудь тимчасовий рядок, довжину якого спочатку установимо в нуль.

Поки не кінець першого файлу
    читаємо байт із першого файлу
    і йдемо в циклі по всій таблиці статистики
        якщо поле b з поточної запису збігається із зчитаним
        байтом, то
            додаємо код з поточної запису до тимчасового
            рядку і,
                поки довжина рядка більше чи дорівнює восьми,
                    перетворимо цей рядок у байт
                    записуємо цей байт у файл і
                    і вирізаємо з рядка перші вісім символів
            насильно перериваємо цикл (це робиться
            виключно для прискорення програми – шукати далі
            просто безглуздо)

Якщо по закінченні циклу рядок не порожній, це означає, що в ній залишився ще обривок коду – адже вони різної довжини, і їхня сума не завжди кратна восьми, тобто цілому числу байт. У цьому випадку нам потрібно доповнити цю рядка до довжини 8 нулями і записати її у файл.

От, загалом, і вся програма кодування. Декодер буде використовувати ті ж самі структури даних, змінні і навіть деякі процедури з тих, що використовує кодер.

Крупноблочно програма декодування може складатися з таких частин:

1. Ініціалізуємо таблицю статистики

2. Відкриваємо другий файл для запису, а перший для читання
3. Обнуляємо початок і кінець
4. Читаємо заголовок архівного файлу (довжину нестиснутого файлу, значення кінця і таблицю відповідностей виду символ–код)
5. Декодуємо архівний файл
6. І закриваємо обидва

Читаємо заголовок архіву...

Спочатку зчитуємо чотири байти, що перетворюємо в довге ціле – довжину файлу, потім один байт – значення кінця і лише після того,

у циклі від початку до кінця

читаємо значення поля b поточного масиву статистики і

довжину кодової групи

визначаємо кількість байт, що відповідає даної

бітовій довжині – найближче більше число, кратне восьми

далі, читаємо байт із файлу, перетворюємо його на рядок і

приклеюємо цей рядок до поля коду поточної запису – усі

це, природно, стільки разів, скільки байт відповідає

бітовій довжині даного коду

Так як поле коду поточної запису після цієї процедури

має неправильну довжину (кратну восьми), то скоригуємо

її, занісши в нульовий байт рядка, що зберігає її

довжину, зчитане значення бітової довжини

Декодування за відомої таблиці відповідностей здійснюється настільки ж просто, як і кодування. Єдине додавання – крім тимчасового рядка, нам ще знадобиться змінна – лічильник числа декодованих байт того ж типу, що і довжина файлу. Значення лічильника і довжини рядка спочатку встановлюється в нуль.

Поки не кінець файлу і значення лічильника менше довжини файлу

читаємо один байт із файлу

додаємо до тимчасового рядка цей байт, перетворений у рядок

йдемо по таблиці статистики від початку до кінця і

якщо поле коду поточної запису дорівнює фрагменту
тимчасового рядка, за довжиною однаковим з даним полем, то
видаляємо з тимчасового рядка число символів,
за довжиною рівне довжині поточного коду
записуємо у файл поле b поточного запису
збільшуємо лічильник
якщо лічильник зрівнявся з довжиною файлу, то
перериваємося
перезапускаємо поточний цикл установкою циклової змінної в мінус одиницю

От, власне, і все.

§6.

Арифметичне кодування

При арифметичному кодуванні текст представляється дійсними числами в інтервалі від 0 до 1. В міру кодування тексту, інтервал, що відображає його, зменшується, а кількість бітів для його представлення зростає. Чергові символи тексту скорочують величину інтервалу, виходячи зі значень їхніх ймовірностей, обумовлених моделлю. Більш ймовірні символи роблять це в меншому ступені, чим менш ймовірні, і, отже, додають менше бітів до результату.

Перед початком роботи відповідний тексту інтервал є [0; 1). При обробці чергового символу його ширина звужується за рахунок виділення цьому символу частини інтервалу. Наприклад, застосуємо до тексту “eaii!” алфавіту {a, e, i, o, u, !} модель з постійними ймовірностями, заданими в таблиці:

Символ	*Імовірність*	*Інтервал*
a	.2	[0.0; 0.2)
e	.3	[0.2; 0.5)
i	.1	[0.5; 0.6)
o	.2	[0.6; 0.8)
u	.1	[0.8; 0.9)
!	.1	[0.9; 1.0)

І кодувальнику, і декодувальнику відомо, що на самому початку інтервал є [0; 1). Після перегляду першого символу “e”, кодувальник звужує інтервал до [0.2; 0.5), що модель виділяє цьому символу. Другий символ “a” звузить цей новий інтервал до першої його п’ятої частини, оскільки для “a” виділений фіксований інтервал [0.0; 0.2). У результаті одержимо робочий інтервал [0.2; 0.26), тому що попередній інтервал мав ширину в 0.3 одиниці й одному п’ята від нього є 0.06. Наступному символу “i” відповідає фіксований інтервал [0.5; 0.6), що стосовно до робочого інтервалу [0.2; 0.26) звужує його до інтервалу [0.23, 0.236). Продовжуючи в тім же дусі, маємо:

На початку [0.0; 1.0)

Після перегляду “e”	[0.2;	0.5)
Після перегляду “a”	[0.2;	0.26)
Після перегляду “i”	[0.23;	0.236)
Після перегляду “i”	[0.233;	0.2336)
Після перегляду “!”	[0.23354;	0.2336)

Припустимо, що всі що декодувальник знає про текст, це кінцевий інтервал [0.23354; 0.2336). Він відразу ж розуміє, що перший закодований символ є “e”, тому що підсумковий інтервал цілком лежить в інтервалі, виділеному моделлю цьому символу відповідно до таблиці. Тепер повторимо дії кодувальника:

Спочатку [0.0; 1.0)

Після перегляду “e” [0.2; 0.5)

Звідси ясно, що другий символ – це “a”, оскільки це приведе до інтервалу [0.2; 0.26), що цілком уміщає підсумковий інтервал [0.23354; 0.2336). Продовжуючи працювати в такий же спосіб, декодувальник витягне весь текст.

Декодувальнику немає необхідності знати значення обох границь підсумкового інтервалу, отриманого від кодувальника. Навіть єдиного значення, що лежить усередині нього, наприклад 0.23355, уже досить. (Інші числа – 0.23354,0.23357 чи навіть 0.23354321 – цілком годяться). Однак, щоб завершити процес, декодувальнику потрібно вчасно розпізнати кінець тексту. Крім того, те саме число 0.0 можна представити і як “a”, і як “aa”, “aaa” і т.д. Для усунення неясності ми повинні позначити завершення кожного тексту спеціальним символом EOF, відомим і кодувальнику, і декодувальнику. Для алфавіту з таблиці для цієї мети, і тільки для неї, буде використовуватися символ "!". Коли декодувальник зустрічає цей символ, він припиняє свій процес.

Для фіксованої моделі, що задається моделлю таблиці, ентропія п’ятисимвольного тексту “eaii!” буде:

$$-\log 0.3 - \log 0.2 - \log 0.1 - \log 0.1 - \log 0.1 = -\log 0.00006 \sim 4.22.$$

(Тут застосовуємо логарифм за соновою 10, тому що вищерозглянуте кодування виконувалося для десяткових чисел). Це пояснює, чому потрібно 5 де-

сяткових цифр для кодування цього тексту. По суті, ширина підсумкового інтервалу є 0.2336–0.23354=0.00006, а ентропія – від'ємний десятковий логарифм цього числа. Звичайно ми працюємо з двійковою арифметикою, передаємо двійкові числа і вимірюємо ентропію в бітах.

П'яти десяткових цифр здається забагато для кодування тексту з 4-х голосних! Може бути не зовсім удало було закінчувати приклад розгортанням, а не стиском. Однак, ясно, що різні моделі дають різну ентропію. Краща модель, побудована на аналізі окремих символів тексту "eaii!", є наступна множина частот символів: {"e" (0.2), "a" (0.2), "i" (0.4), "!" (0.2)}. Вона дає ентропію, рівну 2.89 у десятковій системі числення, тобто кодує вихідний текст числом з 3-х цифр. Однак, більш складні моделі, як відзначалося раніше, дають у загальному випадку набагато кращий результат.

Нижче подано фрагмент псевдокоду, що поєднує процедури кодування і декодування. Символи в ньому нумеруються як 1, 2, 3, …. Частотний інтервал для i-го символу задається від cum_freq[i] до cum_freq[i-1]. При убуванні і cum_freq[i] зростає так, що cum_freq[0]=1. (Причина такого "зворотного" погодження полягає в тому, що cum_freq[0] буде потім містити множник, що нормалізує, який зручно зберігати на початку масиву). Поточний робочий інтервал задається в [low; high] і буде на самому початку дорівнює [0; 1) і для кодувальника, і для декодувальника.

На жаль, цей псевдокод дуже спрощений, тоді як на практиці існує кілька факторів, що ускладнюють і кодування, і декодування.

```
/* АЛГОРИТМ АРИФМЕТИЧНОГО КОДУВАННЯ */
/* З кожним символом тексту звертатися до процедури encode_symbol() */
/* Перевірити, що "завершальний" символ закодований останнім */
/* Вивести отримане значення інтервалу [low; high) */
encode_symbol(symbol,cum_freq)
        range = high - low
        high = low + range*cum_freq[symbol-1]
        low = low + range*cum_freq[symbol]
```

```
/* АЛГОРИТМ АРИФМЕТИЧНОГО ДЕКОДУВАННЯ */
/* Value – це число, що надійшло на вхід */
/* Звертання до процедури decode_symbol() поки вона не поверне */
/* "завершальний" символ */
decode_symbol(cum_freq)
        пошук такого символу, що
        cum_freq[symbol] <= (value - low)/(high - low) < cum_freq[symbol-1]
        /* Це забезпечує розміщення value усередині нового інтервалу */
        /* [low; high), що відбито в частині програми, що залишилася */
                range = high - low
                high = low + range*cum_freq[symbol-1]
                low = low + range*cum_freq[symbol]
return symbol
```

Зауваження:

1. Описаний алгоритм кодування нічого не передає до повного завершення кодування всього тексту, також і декодувальник не починає процес, поки не одержить стиснутий текст цілком. Для більшості випадків необхідний поступовий режим виконання.
2. Бажаним є використання цілочисельної арифметики. Необхідна для представлення інтервалу [low; high) точність зростає разом з довжиною тексту. Поступове виконання допомагає перебороти цю проблему, вимагаючи при цьому уважного обліку можливостей переповнення і негативного переповнення.
3. Реалізація моделі повинна мінімізувати час визначення наступного символу алгоритмом декодування. Крім того, адаптивні моделі повинні також мінімізувати час, необхідний для підтримки частот, що накопичуються.

Наступна програма містить робочий код процедур арифметичного кодування і декодування. Він значно більш детальний, ніж псевдокод. Реалізацію двох різних моделей (фіксованої та адаптивної) подано нижче, при цьому про-

грама може використовувати кожну з них.

arithmetic_coding.h

```
 1 /* ОГОЛОШЕННЯ, НЕОБХІДНІ ДЛЯ АРИФМЕТИЧНОГО */
 2 /* КОДУВАННЯ І ДЕКОДУВАННЯ */
 3
 4 /* ІНТЕРВАЛ ЗНАЧЕНЬ АРИФМЕТИЧНОГО КОДУ */
 5
 6 #define Code_value_bits 16 /* Кількість бітів для коду */
 7 typedef long code_value;   /* Тип арифметичного коду */
 8
 9 #define Top_value (((long) 1 << Code_value_bits) - 1)
10                            /* Максимальне значення коду */
11
12 /* ПОКАЖЧИКИ НА СЕРЕДИНУ І ЧВЕРТІ ІНТЕРВАЛУ ЗНАЧЕНЬ КОДУ */
13
14 #define First_qtr  (Top_value/4+1) /* Кінець першої чверті */
15 #define Half       (2*First_qtr)   /* Кінець першої половини */
16 #define Third_qtr  (3*First_qtr)   /* Кінець третьої чверті */
```

model.h

```
17 /* ІНТЕРФЕЙС ІЗ МОДЕЛЛЮ */
18
19
20 /* МНОЖИНА КОДОВАНИХ СИМВОЛІВ */
21
22 #define No_of_chars    256 /* Кількість вихідних символів */
23 #define EOF_symbol     (No_of_chars+1) /* Індекс кінця файлу */
24
25 #define No_of_symbols  (No_of_chars+1) /* Усього символів */
26
27
28 /* Таблиці перекодування вихідних і робочих символів */
29
30 int char_to_index[No_of_chars]; /* З вихідного в робочий */
31 unsigned char index_to_char[No_of_symbols+1]; /* Навпаки */
32
33
34 /* ТАБЛИЦЯ НАКОПИЧЕНИХ ЧАСТОТ */
35
```

```
36 #define Max_frequency 16383    /* Максимальне значення */
37                                /* частоти = 2^14 - 1 */
38 int cum_freq[No_of_symbols+1]; /* Масив накопичених частот */
```

encode.c

```
39 /* ГОЛОВНА ПРОЦЕДУРА КОДУВАННЯ */
40
41 #include <stdio.h>
42 #include "model.h"
43
44 main()
45 { start_model();
46   start_outputing_bits();
47   start_encoding();
48   for (;;) {                        /* Цикл обробки символів */
49     int ch; int symbol;
50     ch = getc(stdin);               /* Читання вихідного символу */
51     if (ch==EOF) break;             /* Вихід по кінці файлу */
52     symbol = char_to_index[ch];     /* Знайти робочий символ */
53     encode_symbol(symbol,cum_freq); /* Закодувати його */
54     update_model(symbol);           /* Обновити модель */
55   }
56   encode_symbol(EOF_symbol,cum_freq); /* Кодування EOF */
57   done_encoding();            /* Додавання ще декількох біт */
58   done_outputing_bits();
59   exit(0);
60 }
```

arithmetic_encode.c

```
61 /* АЛГОРИТМ АРИФМЕТИЧНОГО КОДУВАННЯ */
62
63 #include "arithmetic_coding.h"
64
65 static void bit_plus_follow();
66
67
68 /* ПОТОЧНИЙ СТАН КОДУВАННЯ */
69
70 static code_value low, high; /* Краї поточної області кодів */
71 static long bits_to_follow;  /* Кількість бітів, що виводяться */
```

```
 72        /* після наступного біта зі зворотним йому значенням */
 73
 74
 75 /* ПОЧАТОК КОДУВАННЯ ПОТОКУ СИМВОЛІВ */
 76
 77 start_encoding()
 78 { low = 0;             /* Повний кодовий інтервал */
 79   high = Top_value;
 80   bits_to_follow = 0; /* Додавати біти поки не треба */
 81 }
 82
 83
 84 /* КОДУВАННЯ СИМВОЛУ */
 85
 86 encode_symbol(int symbol, int *cum_freq)
 87               /* symbol - кодований символ */
 88               /* cum_freq - частоти, що накопичуються */
 89 { long range;                  /* Ширина поточного */
 90   range = (long)(high-low)+1; /* кодового інтервалу */
 91   high = low +                 /* Звуження інтервалу кодів до */
 92         (range*cum_freq[symbol-1])/cum_freq[0]-1;
 93   low = low +                  /* виділеного для symbol*/
 94         (range*cum_freq[symbol])/cum_freq[0];
 95   for (;;) {                   /* Цикл виведення бітів */
 96     if (high<Half) {           /* Якщо в нижній половині */
 97       bits_plus_follow(0);     /* вихідного інтервалу, */
 98     }                          /* то виводимо 0 */
 99     else if (low>=Half) {      /* Якщо у верхній, то */
100       bit_plus_follow(1);      /* вивести 1, а потім */
101       low -= Half;             /* забрати відому у */
102       high -= Half;            /* границь спільну частину */
103     }
104     else if (low>=First_qtr    /* Якщо поточний інтервал */
105          && high<Third_qtr) { /* містить середину */
106       bits_to_follow +=1;      /* вихідного, то вивід ще */
107       low -= First_qtr;        /* одного зворотного біта */
108       high -= First_qtr;       /* пізніше, а зараз */
109     }                          /* забрати спільну частину */
110     else break;                /* Інакше вийти з циклу */
```

```
111     low = 2*low;              /* Розширити поточний */
112     high = 2*high+1;          /* робочий кодовий інтервал */
113   }
114 }
115
116
117 /* ЗАВЕРШЕННЯ КОДУВАННЯ ПОТОКУ */
118
119 done_encoding()                  /* Виведення двох бітів, що */
120 { bits_to_follow += 1;           /* визначають чверть, */
121   if (low<First_qtr) bit_plus_follow(0); /* що лежить в */
122   else bit_plus_follow(1);       /* поточному інтервалі */
123 }
124
125
126 /* ВИВЕДЕННЯ БІТА РАЗОМ З НАСТУПНИМИ ЗА НИМ ЗВОРОТНИМ ДО НЬОГО */
127
128 static void bit_plus_follow(int bit)
129 {
130   output_bit(bit);
131   while (bits_to_follow>0) {
132     output_bit(!bit);
133     bits_to_follow -= 1;
134   }
135 }
```

decode.c

```
136 /* ГОЛОВНА ПРОЦЕДУРА ДЛЯ ДЕКОДУВАННЯ */
137
138 #include <stdio.h>
139 #include "model.h"
140
141 main()
142 { start_model();
143   start_inputing_bits();
144   start_decoding();
145   for (;;) {
146     int ch; int symbol;
147     symbol = decode_symbol(cum_freq);
148     if (symbol == EOF_symbol) break;
```

```
149     ch = index_to_char(symbol);
150     putc(ch,stdout);
151     update_model(symbol);
152   }
153   exit(0);
154 }
```

arithmetic_decode.c

```
155 /* АЛГОРИТМ АРИФМЕТИЧНОГО ДЕКОДУВАННЯ */
156
157 #include "arithmetic_coding.h"
158
159
160 /* ПОТОЧНИЙ СТАН ДЕКОДУВАННЯ */
161
162 static code_value value;     /* Поточне значення коду */
163 static code_value low, high; /* Границі поточного */
164                              /* кодового інтервалу */
165
166 /* ПОЧАТОК ДЕКОДУВАННЯ ПОТОКУ СИМВОЛІВ */
167
168 start_decoding();
169 { int i;
170   value = 0;                 /* Введення бітів для заповнення */
171   for (i = 1; i<=Code_value_bits; i++) {     /* значення коду */
172     value = 2*value+input_bit();
173   }
174   low = 0;                   /* На самому початку поточний */
175   high = Top_value;          /* робочий інтервал дорівнює */
176 }                            /* вихідному */
177
178
179 /* ДЕКОДУВАННЯ НАСТУПНОГО СИМВОЛУ */
180
181 int decode_symbol(int *cum_freq) /* Накопичені частоти */
182 {
183   long range;                    /* Ширина інтервалу */
184   int cum;                       /* Накопичена частота */
185   int symbol;                    /* Декодовуваний символ */
186   range = (long)(high-low)+1;
```

```
187   cum = /* Знаходження значення накопиченої частоти для value */
188     (((long)(value-low)+1)*cum_freq[0]-1)/range;
189   for (symbol = 1; cum_freq[symbol]>cum; symbol++);
190   high = low +                    /* Після знаходження символа */
191     (range*cum_freq[symbol-1])/cum_freq[0]-1;
192   low = low +
193     (range*cum_freq[symbol])/cum_freq[0];
194   for (;;) {                      /*Цикл відкидання бітів */
195     if (high<Half) {              /* Розширення нижньої половини */
196         /* нічого */
197     }
198     else if (low>=Half) {         /* Розширення верхньої */
199       value -= Half;              /* половини після віднімання */
200       low -= Half;                /* зсуву Half */
201       high -= Half;
202     }
203     else if (low>=First_qtr       /* Розширення середньої */
204         && high<Third_qtr) {      /* половини */
205       value -= First_qtr;
206       low -= First_qtr;
207       high -= First_qtr;
208     }
209     else break;
210     low = 2*low;                  /* Збільшити масштаб */
211     high = 2*high+1;              /* інтервалу */
212     value = 2*value+input_bit(); /* Додати новий біт */
213   }
214   return symbol;
215 }
```

bit_input.c

```
216 /* ПРОЦЕДУРИ ВВЕДЕННЯ БИТОВ */
217
218 #include <stdio.h>
219 #include "arithmetic_coding.h"
220
221
222 /* БІТОВИЙ БУФЕР */
223
224 static int buffer;          /* Сам буфер */
```

```
225 static int bits_to_go;     /* Скільки бітів у буфері */
226 static int garbage_bits;   /* Кількість бітів */
227                            /* після кінця файлу */
228
229 /* ІНІЦІАЛІЗАЦІЯ ПОБИТНОГО ВВЕДЕННЯ */
230
231 start_inputing_bits()
232 { bits_to_go = 0;          /* Спочатку буфер порожній */
233   garbage_bits = 0;
234 }
235
236
237 /* УВЕДЕННЯ БІТА */
238
239 int input_bit()
240 { int t;
241   if (bits_to_go==0) {     /* Читання байта, якщо */
242     buffer = getc(stdin);  /* буфер порожній */
243     if (buffer==EOF) {
244       garbage_bits += 1;   /* Приміщення будь-яких бітів */
245       if (garbage_bits>Code_value_bits-2) {     /* після */
246         fprintf(stderr,"Bad input file\n");     /* кінця */
247         exit(-1);          /* файлу з перевіркою */
248       }                    /* на занадто велику їх */
249     }                      /* кількість */
250     bits_to_go = 8;
251   }
252   t = buffer&1;            /* Видача чергового */
253   buffer >>= 1;            /* біта з правого кінця */
254   bits_to_go -= 1;         /* (дна) буфера */
255   return t;
256 }
```

bit_output.c

```
257 /* ПРОЦЕДУРИ ВИВЕДЕННЯ БІТІВ */
258
259 #include <stdio.h>
260
261
262 /* БІТОВИЙ БУФЕР */
```

```
263
264 static int buffer;     /* Біти для виведення */
265 static int bits_to_go; /* Кількість вільних */
266                        /* бітів у буфері */
267
268 /* ІНІЦІАЛІЗАЦІЯ БІТОВОГО ВИВЕДЕННЯ */
269
270 start_outputing_bits()
271 { buffer = 0; /* Спочатку буфер порожній */
272   bits_to_go = 8;
273 }
274
275
276 /* ВИВЕДЕННЯ БІТА */
277
278 output_bit(int bit)
279 {
280   buffer >>= 1;            /* Біт - у початок буфера */
281   if (bit) buffer |= 0x80;
282   bits_to_go -= 1;
283   if (bits_to_go==0) {
284     putc(buffer,stdout);   /* Виведення повного буфера */
285     bits_to_go = 8;
286   }
287 }
288
289
290 /* ВИМИВАННЯ ОСТАННІХ БІТІВ */
291
292 done_outputing_bits()
293 { putc(buffer>>bits_to_go,stdout);
294 }
```

fixed_model.c

```
 1 /* МОДЕЛЬ З ФІКСОВАНИМ ДЖЕРЕЛОМ */
 2
 3 #include "model.h"
 4
 5 int freq[No_of_symbols+1] = {
 6    0,
```

```
 7    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,124,  1,  1,  1,  1,  1,
 8    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
 9
10 /*     !   "   #   $   %   &   '   (   )   *   +   ,   -   .   / */
11 1236,  1, 21,  9,  3,  1, 25, 15,  2,  2,  2,  1, 79, 19, 60,  1,
12
13 /* 0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ? */
14   15, 15,  8,  5,  4,  7,  5,  4,  6,  3,  2,  1,  1,  1,  1,  1,
15
16 /* @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O */
17    1, 24, 15, 22, 12, 15, 10,  9, 16, 16,  8,  6, 12, 23, 13,  1,
18
19 /* P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _ */
20   14,  1, 14, 28, 29,  6,  3, 11,  1,  3,  1,  1,  1,  1,  1,  3,
21
22 /* '   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o */
23    1,491, 85,173,232,744,127,110,293,418,  6, 39,250,139,429,446,
24
25 /* p   q   r   s   t   u   v   w   x   y   z   {   |   }         */
26  111,  5,388,375,531,152, 57, 97, 12,101,  5,  2,  1,  2,  3,  1,
27
28    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
29    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
30    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
31    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
32    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
33    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
34    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
35    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
36    1
37 };
38
39
40 /* ІНІЦІАЛІЗАЦІЯ МОДЕЛІ */
41
42 start_model()
43 { int i;
44   for (i = 0; i<No_of_chars; i++) {   /* Установка таблиць */
45     char_to_index[i] = i+1;           /* перекодування типів */
```

```
46      index_to_char[i+1] = i;
47    }
48    cum_freq[No_of_symbols] = 0;
49    for (i = No_of_symbols; i>0; i--) { /* Установка */
50      cum_freq[i-1] = cum_freq[i] + freq[i]; /* лічильників */
51    }                                   /* накопичених частот */
52    if (cum_freq[0] > Max_frequency) abort(); /* Перевірка */
53  }                                     /* лічильників по границях */
54
55
56  /* ОБНОВИТИ МОДЕЛЬ У ЗВ'ЯЗКУ З НОВИМ СИМВОЛОМ */
57
58  update_model(int symbol)
59  {
60    /* Нічого не робиться */
61  }
```

adaptive_model.c

```
1  /* МОДЕЛЬ ІЗ ДЖЕРЕЛОМ, ЩО НАБУДОВУЄТЬСЯ */
2
3  #include "model.h"
4
5  int freq[No_of_symbols+1] /* Частоти символів */
6
7
8  /* ІНІЦІАЛІЗАЦІЯ МОДЕЛІ */
9
10 start_model()
11 { int i;
12   for (i = 0; i<No_of_chars; i++) {   /* Установка таблиць */
13     char_to_index[i] = i+1;           /* перекодування між */
14     index_to_char[i+1] = i;           /* вихідними і робочими */
15   }                                   /* символами */
16   for (i = 0; i<No_of_symbols; i++) { /* Установка значень */
17     freq[i] = 1;                      /* лічильників частот у 1 */
18     cum_freq[i] = No_of_symbol-i;     /* для всіх символів */
19   }
20   freq[0] = 0;                        /* freq[0] повинен */
21 }                                     /* відрізнятися від freq[1] */
22
```

```
23
24 /* ОНОВЛЕННЯ МОДЕЛІ У ВІДПОВІДНОСТІ ДО НОВОГО СИМВОЛУ */
25
26 update_model(int symbol)              /* Індекс нового символу */
27 {
28   int i;                              /* Новий індекс */
29   if (cum_freq[0]==Max_frequency) {   /* Якщо лічильники частот */
30     int cum;                          /* досягли свого */
31     cum = 0;                          /* максимуму */
32     for (i = No_of_symbols; i>=0; i--) { /* Тоді ділимо */
33       freq[i] = (freq[i]+1)/2;        /* їх усіх навпіл, */
34       cum_freq[i] = cum;              /* не приводячи до нуля */
35       cum += freq[i];
36     }
37   }
38   for (i = symbol; freq[i]==freq[i-1]; i--);
39   if (i<symbol) {
40     int ch_i, ch_symbol;
41     ch_i = index_to_char[i];            /* Відновлення таблиць */
42     ch_symbol = index_to_char[symbol]; /* перекодування */
43     index_to_char[i] = ch_symbol;       /* у випадку */
44     index_to_char[symbol] = ch_i;       /* переміщення символа */
45     char_to_index[ch_i] = symbol;
46     char_to_index[ch_symbol] = i;
47   }
48   freq[i] += 1;                         /* Збільшити значення */
49   while (i>0) {                         /* лічильника частоти для */
50     i -= 1;                             /* символу й обновити */
51     cum_freq[i] += 1;                   /* накопичені частоти */
52   }
53 }
```

Реалізація моделі. Розглянемо інтерфейсу з моделлю (рядки 20-38). У мові Си байт являє собою ціле число від 0 до 255 (тип char). Тут же ми подаємо байт як ціле число від 1 до 257 включно (тип index), де EOF трактується як 257-ий символ. Бажано відсортувати модель у порядку спадання частот для мінімізації кількості виконань циклу декодування (рядок 189). Перехід від типу char у index, і навпаки, реалізується за допомогою двох таблиць – index_to_char[] і

char_to_index[]. В одній з наших моделей ці таблиці формують index простим додаванням 1 до char, але в іншій виконується більш складне перекодування, що надає часто використовуваним символам маленькі індекси.

Імовірності представляються в моделі як цілочисельні лічильники частот, а частоти, що накопичуються, зберігаються в масиві cum_freq[]. Як і в попередньому випадку, цей масив – "зворотний", і лічильник загальної частоти, застосовуваний для нормалізації всіх частот, розміщається в cum_freq[0]. Частоти, що накопичуються, не повинні перевищувати встановлений у Max_frequency максимум, а реалізація моделі повинна запобігати переповненню відповідним масштабуванням. Необхідно також хоча б на 1 забезпечити розходження між двома сусідніми значеннями cum_freq[], інакше розглянутий символ не зможе бути переданий.

Прирощувана передача й отримання. На відміну від псевдокода, програма представляє low і high цілими числами. Для них, і для інших корисних констант, визначений спеціальний тип даних code_value. Це – Top_value, що визначає максимально можливий code_value, First_qtr і Third_qtr, що представляють частини інтервалу (рядки 6–16). У псевдокоді поточний інтервал представлений через [low; high), а в програмі це [low; high] – інтервал, що включає в себе значення high. Насправді більш правильно, хоча і більш незрозуміло, стверджувати, що в програмі інтервал, що представляється, є [low; high + 0.1111...) з тієї причини, що при масштабуванні границь для збільшення точності, нулі зміщаються до молодших бітів low, а одиниці зміщаються в high. Хоча можна писати програму на основі різних домовленостей, дана має деякі переваги в спрощенні коду програми.

В міру звуження кодового інтервалу, старші біти low і high стають однаковими, і тому можуть бути передані негайно, тому що на них майбутні звуження інтервалу все рівно вже не будуть впливати. Оскільки ми знаємо, що low<=high, це втілиться в наступну програму:

```
for (;;) {
  if (high < Half) {
    output_bit(0);
```

```
    low = 2 * low;
    high = 2 * high + 1;
  }
  else if (low >= Half) {
    output_bit(1);
    low = 2 * (low - Half);
    high = 2 * (high - Half) + 1;
  }
  else break;
}
```

яка гарантує, що після її завершення буде справедливою нерівність: low<Half<=high. Це можна знайти в рядках 95–113 процедури encode_symbol(). Крім того, є кілька додаткових складностей, пов'язаних з можливостями втрати значимості. Як відзначено вище, потрібно бути уважним при зсуві одиниць до початку high.

Прирощуване введення вихідних даних виконується за допомогою числа, названого value. У програмі оброблені біти переміщаються у верхню частину, а заново одержувані надходять до нижньої. Спочатку, start_decoding() (рядка 168–176) заповнює value отриманими бітами. Після визначення наступного вхідного символу процедурою decode_symbol(), відбувається винесення непотрібних, однакових у low і в high, бітів старшого порядку зсувом value на цю кількість розрядів (виведені біти компенсуються введенням нових з нижнього кінця).

```
for (;;) {
  if (high < Half) {
    value = 2 * value + input_bit();
    low = 2 * low;
    high = 2 * high + 1;
  }
  else if (low > Half) {
    value = 2 * (value - Half) + input_bit();
    low = 2 * (low - Half);
    high = 2 * (high - Half) + 1;
  }
  else break;
```

```
}
```

Перевіримо вірність визначення процедурою decode_symbol() наступного символу. З псевдокоду видно, що decode_symbol() повинна використовувати value для пошуку символу, що скоротив при кодуванні робочий інтервал таким чином, що він продовжує містити в собі value. Рядки 186–188 у decode_symbol() визначають такий символ, для якого

$$\text{cum\_freq[symbol]} <= \left\lfloor \frac{(\text{value} - \text{low} + 1) * \text{cum\_freq}[0] - 1}{\text{high} - \text{low} + 1} \right\rfloor < \text{cum\_freq[symbol-1]},$$

де "[ ]" позначає операцію узяття цілої частини – ділення з відкиданням дробової частини. Це припускає:

$$\text{low} + \left\lfloor \frac{(\text{high} - \text{low} + 1) * \text{cum\_freq}[\text{symbol}]}{\text{cum\_freq}[0]} \right\rfloor <= \text{value} <=$$

$$<= \text{low} + \left\lfloor \frac{(\text{high} - \text{low} + 1) * \text{cum\_freq}[\text{symbol} - 1]}{\text{cum\_freq}[0]} \right\rfloor,$$

таким чином, що value лежить усередині нового інтервалу, що обчислюється процедурою decode_symbol() у рядках 190–193. Це гарантує коректність визначення кожного символу операцією декодування.

Негативне переповнення. Як показано в псевдокоді, арифметичне кодування працює за допомогою масштабування накопичених ймовірностей, що поставляються моделлю в інтервалі [low; high] для кожного переданого символу. Припустимо, що low і high настільки близькі один до одного, що операція масштабування приводить отримані від моделі різні символи до одного цілого числа, що входить у [low; high]. У цьому випадку подальше кодування продовжувати неможливо. Отже, кодувальник повинен стежити за тим, щоб інтервал [low; high] завжди був досить широкий. Найпростішим способом для цього є забезпечення ширини інтервалу не меншої Max_frequency – максимального значення суми всіх частот, що накопичуються (рядок 36).

Як можна зробити цю умову менш строгою? Пояснена вище операція бітового зсуву гарантує, що low і high можуть тільки тоді ставати небезпечно близькими, коли містять між собою Half. Припустимо, вони стають настільки

близькі, що

First_qtr <= low < Half <= high < Third_qtr.

Тоді наступні два біти висновку будуть мати взаємообернені значення: 01 чи 10. Наприклад, якщо наступний біт буде нулем (тобто high опускається нижче Half і [0; Half] стає робочим інтервалом), то наступний за ним – одиницею, тому що інтервал повинний розташовуватися вище середньої крапки робочого інтервалу. Навпаки, якщо наступний біт виявився 1, то за ним буде слідувати 0. Тому тепер інтервал можна безпечно розширити вправо, якщо тільки ми запам'ятаємо, що який би біт не був наступним, слідом за ним необхідно також передати у вихідний потік його зворотне значення. Таким чином, рядки 104-109 перетворять [First_qtr; Third_qtr] у цілий інтервал, запам'ятовуючи в bits_to_follow значення біта, за яким треба посилати зворотний йому. Це пояснює, чому весь вивід відбувається через процедуру bit_plus_follow() (рядки 128-135), а не безпосередньо через output_bit().

Але що робити, якщо після цієї операції дане співвідношення залишається справедливим? У загальному випадку необхідно спочатку порахувати кількість розширень, а потім слідом за черговим бітом послати у вихідний потік знайдену кількість зворотних йому бітів (рядки 106 і 131-134).

Слідуючи цим рекомендаціям, кодувальник гарантує, що після операцій зсуву буде чи low < First_qtr < Half <= high, чи low < Half < Third_qtr <= high. Виходить, поки цілочисельний інтервал, охоплюваний накопиченими частотами, уміщується в її чверті, представленої в code_value, проблема негативного переповнення не виникне. Це відповідає умові:

Max_frequency <= (Top_value + 1)/4 + 1,

яке задовольняє програмі, тому що Max_frequency=2^{14}–1 і Top_value=2^{16}–1 (рядкb 36, 9). Не можна без збільшення кількості бітів, виділюваних для code_values, використовувати для представлення лічильників накопичених частот більше 14 бітів.

Ми розглянули проблему негативного переповнення тільки відносно кодувальника, оскільки при декодуванні кожного символу процес слідує за опе-

рацією кодування, і негативне переповнення не відбудеться, якщо виконується таке ж масштабування з тими ж умовами.

Переповнення. Тепер розглянемо можливість переповнення при цілочисельному множенні, що має місце в рядках 91-94 і 190-193. Переповнення не відбудеться, якщо добуток range*Max_frequency уміщується в ціле слово, тому що накопичені частоти не можуть перевищувати Max_frequency. Range має найбільше значення в Top_value + 1, тому максимально можливий добуток у програмі 1 є $2^{16}*(2^{14}–1)$, що менше 2^{30}. Для визначення code_value (рядок 7) і range (рядки 89, 183) використано тип long, щоб забезпечити 32-х бітову точність арифметичних обчислень.

Обмеженість реалізації. Обмеження, пов’язані з довжиною слова і викликані можливістю переповнення, можна узагальнити, припускаючи, що лічильники частот представляються f бітами, а code_values – с бітами. Програма буде працювати коректно при f <= c – 2 і f + c <= p, де p є точність арифметики.

У більшості реалізацій на Сі, p=31, якщо використовуються цілі числа типу long, і p=32 – при unsigned long. У програмі f=14 і c=16. При відповідних змінах в оголошеннях на unsigned long можна застосовувати f=15 і c=17. Мовою асемблера c=16 є природним вибором, оскільки він прискорює деякі операції порівняння і маніпулювання бітами (наприклад, для рядків 95–113 і 194–213).

Якщо обмежити p 16 бітами, то кращі з можливих значень c і f є відповідно 9 і 7, що не дозволяє кодувати повний алфавіт з 256 символів, оскільки кожний з них буде мати значення лічильника не менше одиниці. З меншим алфавітом (наприклад, з 26 букв чи 4-х бітових величин) справиться ще можна.

Завершення. При завершенні процесу кодування необхідно послати унікальний завершальний символ (EOF-символ, рядок 56), а потім послати вслід достатню кількість бітів для гарантії того, що закодований рядок потрапить у підсумковий робочий інтервал. Так як процедура done_encoding() (рядки 119–123) може бути упевнена, що low і high обмежено, йому потрібно тільки передати 01 чи 10 відповідно для видалення невизначеності, що залишилася. Зручно це робити за допомогою раніше розглянутої процедури bit_plus_follow(). Про-

цедура input_bit() насправді буде читати трохи більше бітів, з тих, що вивела output_bit(), тому що їй потрібно зберігати заповнення нижнього кінця буфера. Неважливо, яке значення мають ці біти, оскільки EOF унікально визначається останніми переданими бітами.

Програма повинна працювати з моделлю, що являє собою пару перекодувальних таблиць index_to_char[] і char_to_index[], і масив накопичених частот cum_freq[]. Причому до останнього пред'являються наступні вимоги: 1) cum_freq[i-1] >= cum_freq[i]; 2) ніколи не робиться спроба кодувати символ i, для якого cum_freq[i-1] = cum_freq[i]; 3) cum_freq[0] <= Max_frequency.

Якщо дані умови дотримані, значення в масиві не повинні мати зв'язку з дійсними значеннями накопичених частот символів тексту. І декодування, і кодування будуть працювати коректно, причому останньому знадобиться менше місця, якщо частоти точні. (Згадаємо успішне кодування "eaii!" відповідно до моделі з таблиці, що не відбиває, однак, справжньої частоти в тексті).

Фіксовані моделі. Найпростішою моделлю є та, у якій частоти символів постійні. Перша модель із програми задає частоти символів, наближені до загального для англійського тексту. Накопиченим частотам байтів, що не появлялись у цьому зразку, даються значення, рівні 1 (тому модель буде працювати і для двійкових файлів, де є всі 256 байтів). Усі частоти були нормалізовані в цілому до 8000. Процедура ініціалізації start_model() просто підраховує накопичену версію цих частот (рядка 48–51), спочатку ініціалізуючи таблиці перекодування (рядки 44–47). Швидкість виконання буде прискорена, якщо ці таблиці переупорядкувати так, щоб найбільш часті символи розташовувалися на початку масиву cum_freq[]. Так як модель фіксована, то процедура update_model(), що викликається з encode.c і decode.c, буде просто заглушкою.

Строгою моделлю є та, де частоти символів тексту в точності відповідають моделі. Наприклад, фіксована модель із програми близька до строгої моделі для деякого фрагмента тексту, з якого вона була узята. Однак, для того, щоб бути справді строгою, її символи, що не з'являлися в цьому фрагменті, повинні мати лічильники рівні нулю, а не 1 (при цьому жертвуючи можливостями вихі-

дних текстів, що містять ці символи). Крім того, лічильники частот не повинні масштабуватися до заданої накопиченої частоти, як це було в програмі. Строга модель може бути обчислена і передана перед пересиланням тексту.

Адаптивна модель. Вона змінює частоти вже знайдених у тексті символів. На початку всі лічильники можуть бути рівні, що відбиває відсутність початкових даних, але в міру перегляду кожного вхідного символу вони змінюються, наближаючи до частот, що спостерігаються. І кодувальник, і декодувальник використовують однакові початкові значення (наприклад, рівні лічильники) і той самий алгоритм відновлення, що дозволить їх моделям завжди залишатися на одному рівні. Кодувальник одержує черговий символ, кодує його і змінює модель. Декодувальник визначає черговий символ на підставі своєї поточної моделі, а потім обновляє її.

Адаптивна модель рекомендується для використання в програмі, оскільки на практиці вона перевершує фіксовану модель за ефективністю стиску. Ініціалізація проводиться також, як для фіксованої моделі, за винятком того, що всі частоти встановлюються в 0. Процедура update_model(symbol) викликається з encode_symbol() і decode_symbol() після обробки кожного символу.

Відновлення моделі досить дороге через необхідність підтримки накопичених сум. У програмі використовувані лічильники частот оптимально розміщені в масиві в порядку спадання своїх значень, що є ефективним видом самоорганізовуваного лінійного пошуку. Процедура update_model() спочатку перевіряє нову модель на предмет перевищення нею обмежень по величині накопиченої частоти, і якщо воно має місце, то зменшує всі частоти діленням на 2, піклуючись при цьому, щоб лічильники не перетворилися в 0, і переобчислює накопичені значення. Потім, якщо необхідно, update_model() переупорядковує символи для того, щоб розмістити поточний у його правильній категорії щодо частотного порядку, чергуючи для відображення змін перекодувальні таблиці. У підсумку процедура збільшує значення відповідного лічильника частоти й упорядковує відповідні накопичені частоти.

§7.

Алгоритм Лемпеля-Зіва-Велча

LZW-стискання названо за іменами його розробників, А. Лемпеля і Дж. Зіва (з більш пізніми модифікаціями Террі А. Велча). Це – передова методика для універсального стиску даних через свою простоту й експлуатаційну гнучкість. Як правило, можна очікувати, що LZW стисне текст, виконуваний код, і звичайний файл даних приблизно наполовину. Для таблиць чисел, комп'ютерних вихідних текстів ступінь стиску 5:1 – звичайна справа.

LZW-стиск завжди використовується у файлах формату GIF і пропонується як опція в TIFF і PostScript.

LZW-стиск використовує кодову таблицю. Звичайно вибирають таблицю з 4096 чи 65536 елементів. У цьому випадку LZW-кодовані дані складаються з 12(16)-розрядних кодів, кожний з який відповідає одному з елементів кодової таблиці. Декодування досягається узяттям коду зі стиснутого файлу і трансляцією його за кодовою таблицею в послідовність символів, що він представляє. Перші 256 кодів таблиці (0–255) відповідають ASCII-кодам і призначені для кодування одиночних байтів зі стисукваного файлу. Наприклад, якби тільки ці перші 256 кодів використовувалися, кожен байт у файлі оригіналу був би перетворений у 12 (16) біт у LZW-кодованому файлі, приводячи до 50(100)-процентного збільшення розміру файлу.

Приклад кодової таблиці для методу LZW (00000-00255 – ідентичні коди, 00256-65535 – унікальні коди):

код	*трансляція*
00000	0
00001	1
00002	2
…	…
00254	254
00255	255

код	*трансляція*
00256	145 201 4
00257	243 245
…	…
65535	xxx xxx xxx

Метод LZW досягає стиску, використовуючи коди від 256 до 65535 для представлення послідовностей байтів. Наприклад, код 523 може представляти послідовність із трьох байтів: 231 124 234. Щораз, коли алгоритм стиску зіштовхується з цією послідовністю у вхідному файлі, код 523 записується в закодований файл. Під час декодування код 523 транслюється по кодовій таблиці, щоб відтворити послідовність з 3 байтів. Чим довше послідовність, призначувана на одиночний код, і чим частіше ця послідовність повторюється, тим ступінь вище стиску, що досягається.

Незважаючи на простоту підходу, є два основних моменти, що повинні бути з’ясовані:

1) як визначити, які послідовності повинні бути в кодовій таблиці?
2) як програма декодування одержить ту ж саму кодову таблицю, що використовувала програма кодування?

Алгоритм LZW красиво розв’язує обидві ці проблеми. Коли LZW-програма починає кодувати файл, кодова таблиця містить тільки перші 256 входів, залишок таблиці поки порожній. Це означає, що перші коди, що записуються в стиснутий файл – просто одиночні байти з вхідного файлу, перетворені у 12(16) біт. Оскільки кодування продовжується, алгоритм LZW ідентифікує повторювані послідовності даних, і додає їх у кодову таблицю. Стиск починається, коли послідовність зустрічається в другий раз. Важливим тут є те, що послідовність із вхідного файлу не додається до кодової таблиці доти, поки вона не буде записана в стиснутий файл як окремі символи (з кодами від 0 до 255). Це важливо, тому що саме це дозволяє програмі декодування відновлювати кодову таблицю безпосередньо зі стиснутих даних, без окремого її збереження.

Тепер, коли суть методу ясна, перейдемо до його деталізації. Насамперед, нам буде потрібно таблиця на 2^{12} чи 2^{16} елементів. При цьому кодом елемента таблиці для спрощення будемо вважати його індекс. Самі елементи таблиці являють собою послідовності елементів байтового типу заздалегідь невизначеної довжини. В об'єктно-орієнтованій мові програмування це може бути контейнерний клас "Гумовий масив однобайтових чисел", у процедурному – структура, що складається з двох полів: покажчика на дані і довжини блоку даних.

Крім таблиці table, нам знадобляться:

– два бінарних файли input і output, відкритих для читання і запису відповідно;
– однобайтова змінна char для збереження одного байта;
– байтова послідовність перемінної довжини string, що має той же тип, що й елементи таблиці table.

Під string+char будемо розуміти послідовність байт, що містить усі байти з string, за якими випливає байт char; довжина цієї послідовності на 1 більше, ніж довжина string.

```
у циклі від 0 до 255
  заповнимо відповідні елементи table числами від 0 до 255
читаємо перший байт із input і зберігаємо його в string
поки в input є байти для введення
  читаємо байт із input і зберігаємо його в char
  якщо string+char у таблиці, то
    додамо char у кінець string
  інакше
    одержимо код для string
    виведемо код у файл output
    додамо в таблицю string+char
    запишемо char у string
одержимо код для string
виведемо код у файл output
```

Для того, щоб з'ясувати, є чи даний елемент у таблиці, досить її переглянути.

Для того, щоб знайти код для заданої трансляції, необхідно переглядати таблицю доти, поки в ній не знайдеться елемент, що збігається з трансляцією. Його індекс у таблиці і буде кодом.

Для додавання нової трансляції в таблицю необхідно знайти в ній невикористовуваний елемент і записати трансляцію в нього. Якщо вся таблиця заповнена, то додавання неможливе, однак це не повинно нас бентежити – просто надалі ми будемо спиратися тільки на наявні в таблиці трансляції.

Покажемо покроково, як описаний алгоритм кодує вхідний файл із 45 байт, що містить рядок

the/rain/in/Spain/falls/mainly/on/the/plain

При цьому, коли ми говоримо, що алгоритм LZW читає символ “a” із вхідного файлу, мається на увазі, що читається значення 01100001 (8-бітове число 97), де 97 – ASCII-код “a”. Коли ми говоримо, що пишемо символ “a” у закодований файл, ми маємо на увазі, що пишеться 000001100001 (0000000001100001) – 97, виражене в 12 (16) бітах.

#	*char*	*string + char*	*У таблиці?*	*output*	*Додавання в таблицю*	*Нова string*	*Коментарі*
1.	t	t				t	перший символ
2.	h	th	ні	t	256=th	h	
3.	e	he	ні	h	257=he	e	
4.	/	e/	ні	e	258=e/	/	
5.	r	/r	ні	/	259=/r	r	
6.	a	ra	ні	r	260=ra	a	
7.	i	ai	ні	a	261=ai	i	
8.	n	in	ні	i	262=in	n	
9.	/	n/	ні	n	263=n/	/	
10.	i	/i	ні	/	264=/i	i	
11.	n	in	так (262)			in	1-е співпадіння
12.	/	in/	ні	262	265=in/	/	

#	*char*	*string + char*	*У таблиці?*	*output*	*Додавання в таблицю*	*Нова string*	*Коментарі*
13.	S	/S	ні	/	266=/S	S	
14.	p	Sp	ні	S	267=Sp	p	
15.	a	pa	ні	p	268=pa	a	
16.	i	ai	так (261)			ai	співпало ai
17.	n	ain	ні	261	269=ain	n	ain → в таблицю
18.	/	n/	так (263)			n/	
19.	f	n/f	ні	263	270=n/f	f	
20.	a	fa	ні	f	271=fa	a	
21.	l	al	ні	a	272=al	l	
22.	l	ll	ні	l	273=ll	l	
23.	s	ls	ні	l	274=ls	s	
24.	/	s/	ні	s	275=s/	/	
25.	m	/m	ні	/	276=/m	m	
26.	a	ma	ні	m	277=ma	a	
27.	i	ai	так (261)			ai	співпало ai
28.	n	ain	так (269)			ain	співпало ain
29.	l	ainl	ні	269	278=ainl	l	
30.	y	ly	ні	l	279=ly	y	
31.	/	y/	ні	y	280=y/	/	
32.	o	/o	ні	/	281=/o	o	
33.	n	on	ні	o	282=on	n	
34.	/	n/	так (263)			n/	
35.	t	n/t	ні	263	283=n/t	t	
36.	h	th	так (256)			th	співпало th
37.	e	the	ні	256	284=the	e	the → в таблицю
38.	/	e/	так (258)			e/	
39.	p	e/p	ні	258	285=e/p	p	

#	*char*	*string + char*	*У таблиці?*	*output*	*Додавання в таблицю*	*Нова string*	*Коментарі*
40.	l	pl	ні	p	286=pl	l	
41.	a	la	ні	l	287=la	a	
42.	i	ai	так (261)			ai	співпало ai
43.	n	ain	так (269)			ain	співпало ain
44.	/	ain/	ні	269	288=ain/	/	
45.	EOF	/		/			виведення string

Алгоритм компресії використовує дві змінні: char і string. Змінна char – один байт, що зберігає значення від 0 до 255. Перемінна string є байтовим масивом змінної довжини, тобто група з одного чи більш символів, кожен з яких представлений одним байтом. Програма починається читанням першого байта з вхідного файлу і занесенням його в змінну string. У таблиці ця дія показана в рядку 1. Потім йде цикл читання всіх інших байт із вхідного файлу. Усякий раз, коли байт читається з вхідного файлу, він зберігається в змінній char. У таблиці даних здійснюється пошук для визначення, чи був уже призначений код для конкатенації з цих двох перемінних, string+char.

Якщо відповідність у кодовій таблиці не знайдено, виконуються три дії:

1. У стиснутий файл записується код, що відповідає змінній string.
2. У таблицю додається новий код для конкатенації string+char.
3. У перемінну string записується значення змінної char.

Приклад цих дій показаний у рядках з 2 по 10 у вищенаведеній таблиці для перших 10 байт файлу приклада.

Коли відповідність у кодовій таблиці знайдено, конкатенація string+char зберігається в змінній string без яких-небудь інших дій. Тобто, якщо відповідна послідовність знайдена в таблиці, ніяких дій не повинно бути зроблено перед визначенням, чи мається в таблиці більш довга співпадаюча послідовність. Приклад цього показується в рядку 11, де послідовність string+char = in ідентифікована як така, що вже має код у таблиці. У рядку 12 наступний символ із вхідного файлу, /, додається до послідовності, і в кодовій таблиці шукається:

in/. Ця більш довга послідовність у таблиці відсутня, тому програма додає її в таблицю, виводить код для більш короткої послідовності, що знаходиться в таблиці (код 262), і починає пошук для послідовностей, що починаються із символу /. Цей процес продовжується доти, поки є символи у вхідному файлі. Програма завершується записом коду, що відповідає поточному значенню змінної string, у стиснутий файл.

При декомпресії за алгоритмом LZW кожен код читається зі стиснутого файлу і порівнюється з кодовою таблицею для отримання трансляції. Оскільки кожен код опрацьовується у такий спосіб, кодова таблиця модифікується так, щоб постійно відповідати тієї, котра використовувалася при стисканні. Однак в алгоритмі декомпресії є маленька складність, пов'язана з тим, що деякі комбінації даних породжують код, якого ще немає в кодовій таблиці. У зв'язку з цим в алгоритмі декодування для збереження коду ми будемо використовувати дві 12(16)-бітні змінні замість однієї – ocode (старий код) і ncode (новий код) на додаток до тих, що застосовувалися в алгоритмі кодування – table, input, output, char і string. З використовуваних в алгоритмі кодування підпрограм нам знадобиться тільки підпрограма додавання в кодову таблицю нової трансляції.

Отже,

```
у циклі від 0 до 255
  заповнимо відповідні елементи table числами від 0 до 255
читаємо перший код з input і зберігаємо його в ocode
одержуємо трансляцію ocode і виводимо її у файл
поки в input є байти для введення
  читаємо код з input і зберігаємо його в ncode
  якщо ncode вже в таблиці, те
    заносимо в string трансляцію ncode
  інакше
    заносимо в string трансляцію ocode
    додамо char у кінець string
  виводимо string у output
  заносимо в char перший символ string
  додаємо в таблицю трансляцію для table[OCODE]+CHAR
  ocode=ncode
```

L

Для написання найпростішого LZW-кодера/декодера потрібно усього кілька десятків рядків програми. Реальні труднощі полягають в ефективному керуванні кодовою таблицею. Лобові рішення приводять до великих витрат пам'яті і повільному виконанню програми. У комерційних LZW-програмах використовується безліч хитрувань для підвищення продуктивності. Наприклад, проблеми пам'яті виникають, тому що заздалегідь невідомо, якої довжини буде трансляція кожного коду. Більшість програм стиску уникають її, користуючись перевагою надлишкової природи кодової таблиці. Наприклад, у рядку 29 покрокового приклада код 278 визначає ainl. Замість збереження цих чотирьох байтів, код 278 може бути збережений як код 269 + l, де код 269 був попередньо визначений як ain у рядку 17. Аналогічно, код 269 міг бути збережений як код 261 + n, де код 261 був попередньо визначений як ai у рядку 7. Це працює скрізь: кожен код може бути виражений як попередній плюс власне новий символ.

Час роботи алгоритму стиску обмежено пошуком у кодовій таблиці для визначення відповідності. Як аналогія, уявіть, що вам необхідно знайти вашого друга за списком абонентів телефонної мережі, а єдиний каталог, яким ви розташовуєте, упорядкований по телефонних номерах, а не за алфавітом. Це змушує вас перегортати сторінку за сторінкою в пошуках потрібного імені. Ця неефективна ситуація ідентична пошуку серед усіх 4096 (а то й 65536) кодів відповідності зазначеному рядку. Рецепт: організуйте кодову таблицю так, щоб те, що ви шукаєте, повідомляло вам, куди дивитися (подібно частково упорядкованої за алфавітом телефонній книзі). Іншими словами, не розміщайте 4096 (65536) кодів у пам'яті послідовно, а, скоріше, поділить пам'ять на частині, засновані на тому, які послідовності будуть там збережені. Наприклад, припустимо, що ми хочемо знайти, чи є послідовність код 329 + x у кодовій таблиці. Кодова таблиця повинна бути організована так, щоб "x" указав на початок перегляду. Існує багато схем керування кодовими таблицями такого виду, і вони можуть бути дуже складними.

Наприкінці хочеться відзначити, що моделювання стиску даних – це дуже конкурентноздатна область. У той час як основи стиску даних відносно прості, програми стиску стають усе більш і більш складними. Не слід очікувати від ваших програм, зроблених за пару годин, продуктивності фірмових продуктів, але і Євгеній Рошал починав писати свій rar, відштовхуючи від тих же основ, що й ви.

§8.

Імітаційне моделювання відмов обладнання

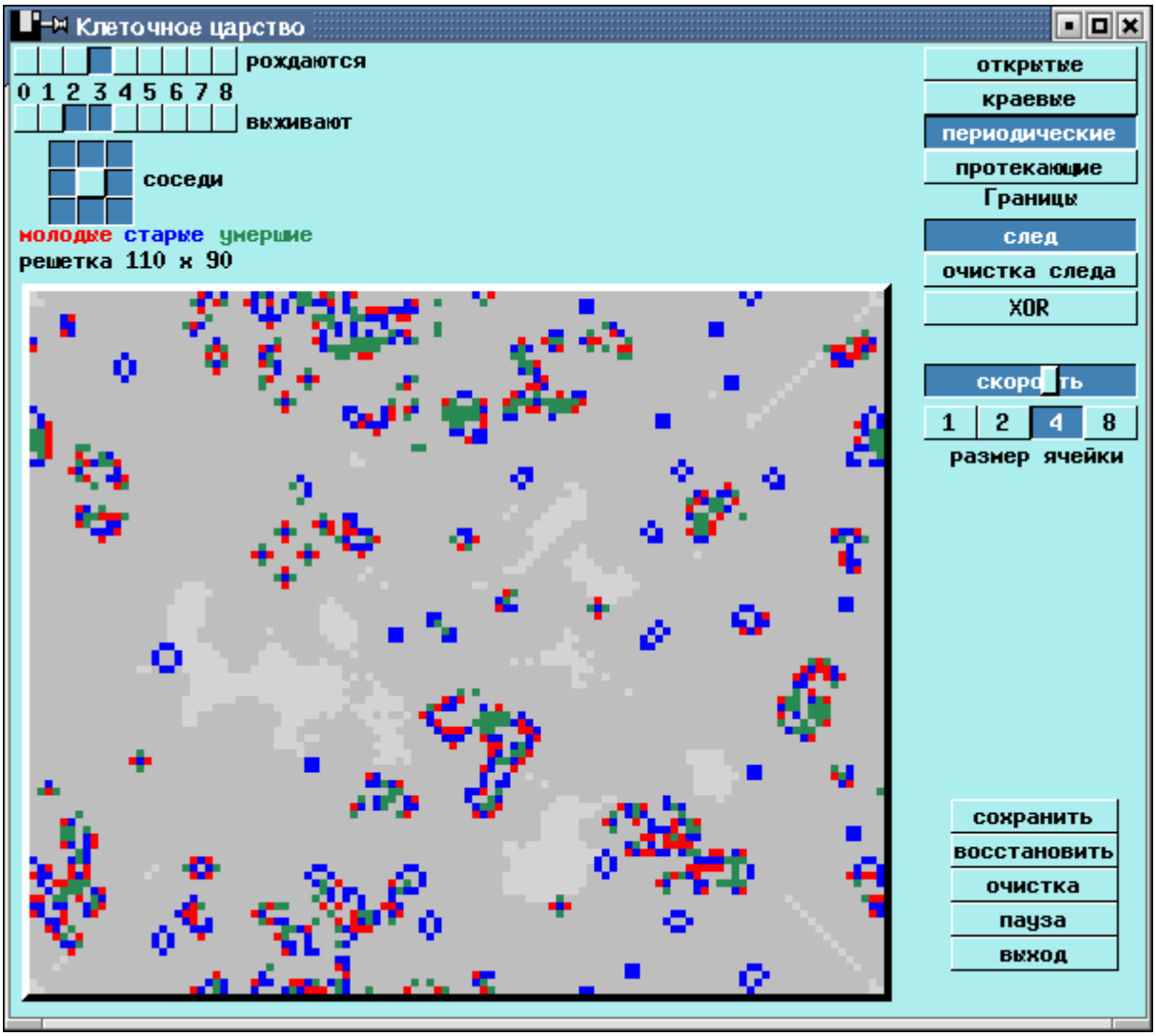
Уявіть себе консультантом онлайнової системи Web Mall, що використовує 10 Web-серверів. Кожного ранку о 9:00 всі 10 серверів перевантажуються. Якщо один чи більше серверів мають збій при завантаженні, пошта не може бути отримана, доки сервер (сервери) не полагодять. Припустимо, що ймовірність збою сервера при завантаженні складає 1% (тобто, у середньому, 1 із 100 запусків завершується невдало). Ваш клієнт хоче знати, який відсоток часу пошту можна отримати (тобто коли усі 10 серверів успішно завантажуються).

Складіть програму, що розраховує, який відсоток днів пошту можна отримати (коли усі 10 серверів успішно завантажуються о 9:00) для 1000, 10000, 100000 та 1000000 днів роботи.

§9.

Універсальний клітковий автомат

І знову пограємося у Бога у власному Всесвіті, керуючі мікрофізикою дискретного світу кліткового автомату. Наша двовимірна Земля – масив комірок, кожна з яких може бути або живою, або порожньою, а їх еволюція у часі відбувається дискретно. Порожні комірки позначимо сірим, новонароджені – червоним, живі – синім, а щойно померлі – зеленим кольором. Якщо увімкнено трасування, залишається слід: комірки, в яких було життя, після того, як воно щезло, змінюють фон.



На кожному кроці долі кожної комірки залежить від кількості її живих сусідів. Визначаючи правила, ви контролюєте, коли у порожній комірці народжується нове життя, і коли життя у комірці продовжується. Повинна також

бути можливість вказувати, які саме сусіди будуть підраховуватись (до 8).

Якщо бути більш точним, окремо треба встановити, за якої кількості обраних сусідів життя у комірці вимирає та народжується, і коли життя у комірці продовжується на наступному кроці.

Границі кліткового Всесвіту можуть бути відкриті (мертві по краях), граничні (живі по краях) чи періодичні. Ще один можливий вид границь – протікаючі – означає, що зверху життя є, а знизу та по боках немає.

Коли система зупинена, її параметри можуть бути змінені у довільний спосіб, а натискання за межами ігрового полю призводить до одного кроку. Стан кожної комірки можна легко змінити на протилежний натискання на неї. Одна комірка можуть бути зображена одним чи більше пікселом.

Збереження та відновлення ігрового поля у файлі є корисним для створення початкових конфігурацій. Найкраще це робити, коли робота системи призупинена.

За восьми сусідів можливо до 18 різних випадків народжень та виживань. Це означає, що в такому випадку можливі 2^{18}=262144 правил, тому навряд чи всіх їх можна перепробувати.

Кількість можливих Всесвітів можна подвоїти за допомогою операції “виключаючого або” (XOR). Якщо такий режим включено, новий стан обчислюється як XOR з попереднім. Тобто, якщо комірка у минулому була живою, новий її стан буде протилежний до того, що визначають правила народження та виживання.

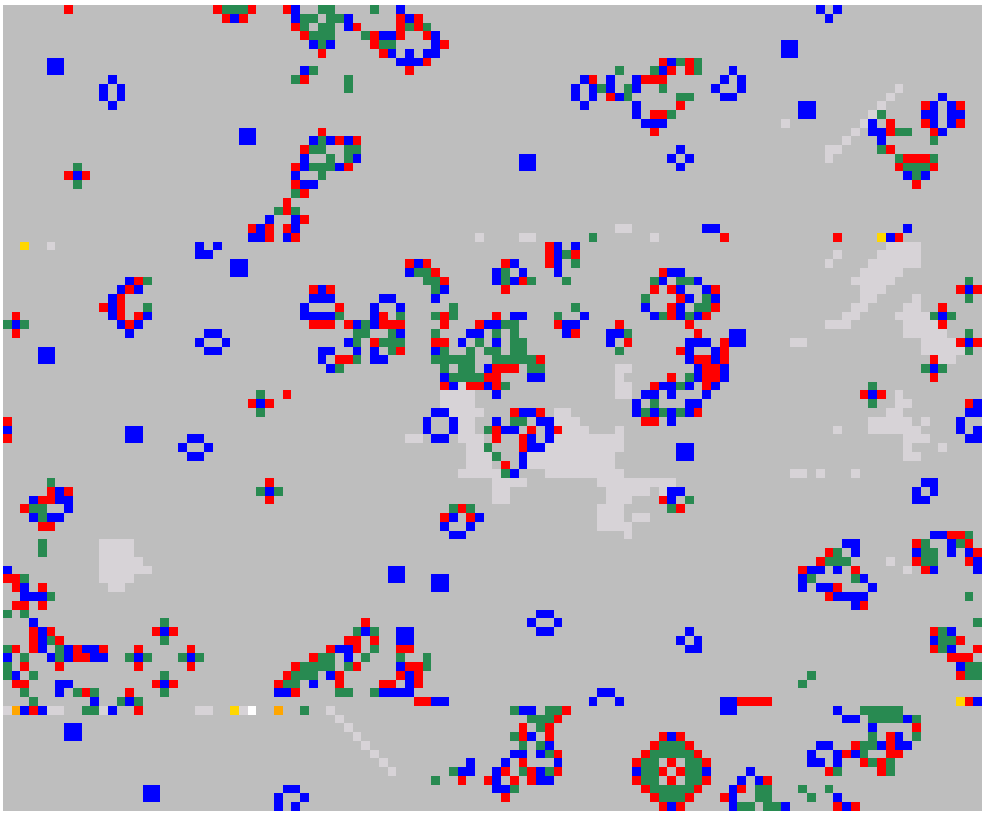
Причиною для цього може бути створення оборотних правил. Якщо історія та поточний стан взаємозамінювані, система повернеться через послідовність конфігурацій до початкового стану. Це приведе до замін місцями новонароджених та померлих комірок.

Така побудова програми дозволяє з легкістю реалізувати в ній класичний клітковий автомат Дж. Конвея “Життя”, в якій використовуються всі 8 сусідів. Нова клітинка народжується рівно за трьох живих сусідів, а живі комірки вимирають від самотності (менше двох сусідів) або перенаселення (менше трьох

сусідів).

Інший широковідомий клітковий автомат – модуль Фрідкіна: дві моделі, що використовують 4 сусідів. Стан комірки змінюється, якщо в неї непарна кількість сусідів, і залишається у протилежному випадку.

Якщо необхідно почати з випадкової позиції, модель запускається з деякими хаотичним правилами (наприклад, більшість правил з народженнями на одного сусіда), а далі – визначити потрібне правило вибору.



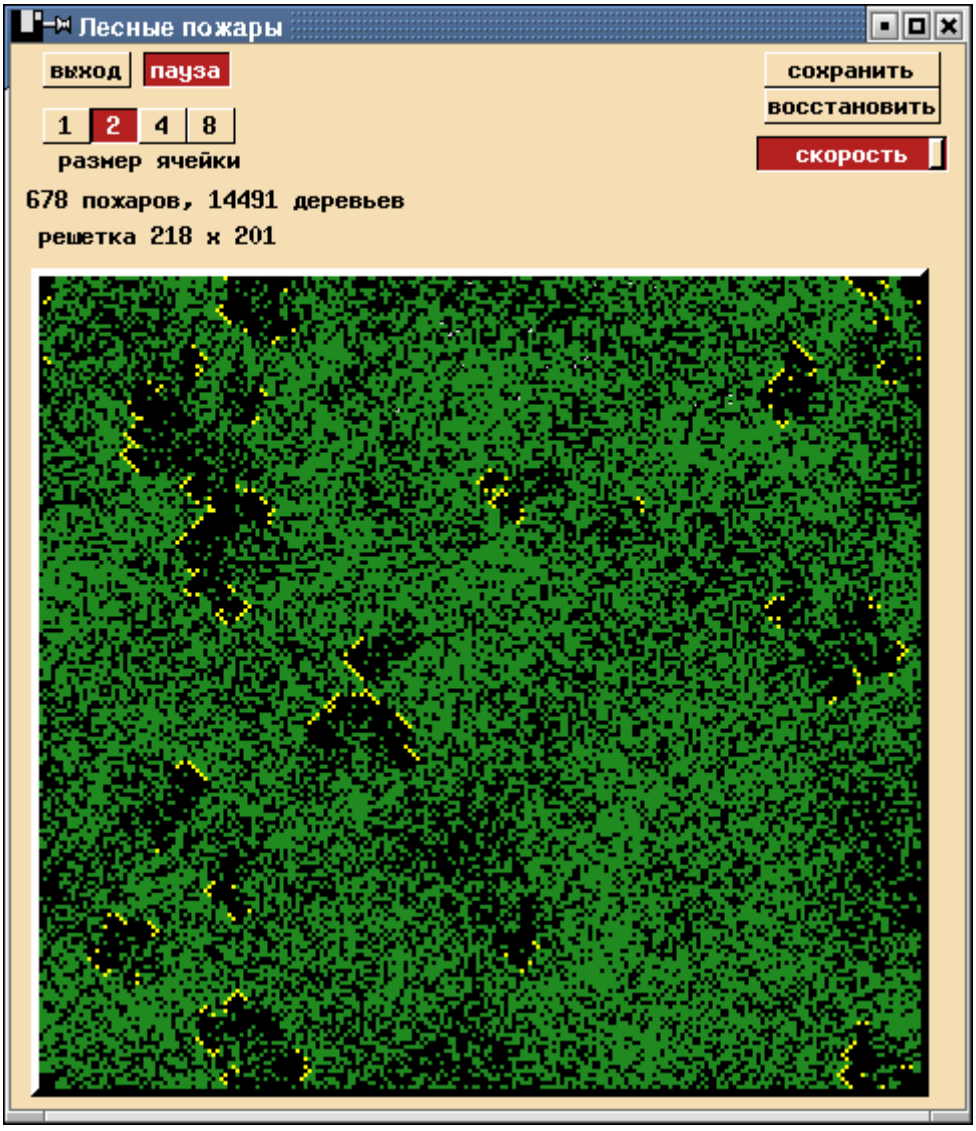
§10.

Лісові пожежі

Завдання. Створіть програму моделювання лісових пожеж за такими правилами:

1. У кожній комірці або дерево, або вогонь, або порожнеча.
2. На кожному кроці вогонь поширюється на сусідні дерева і залишає за собою порожнечу.
3. Дерева народжуються у випадковий спосіб з ймовірністю 1/32 за такт.
4. Якщо немає жодної пожежі, нова починається у довільній позиції.

Зауваження. Пожежі можуть починатися і за діями користувача (наприклад, натисканням кнопок миші).



§11.

Визначення перколяційного порогу

Теорія протікання – ще дуже молода математична дисципліна. Її основні положення було сформульовано у 1957 р. у роботі англійських математиків Бротбента і Хаммерслі. У тій самій роботі було дано й назву теорії – «***percolation theory***». У точному перекладі «***percolation***» означає «просочування», «фільтрація». У вітчизняній науковій літературі поряд з терміном «***протікання***» існує слово «***перколяція***». Незважаючи на молодість теорії, сфера її застосувань надзвичайно велика.

Звернемося до прикладу. Пригадаємо стародавню легенду про Ноєв ковчег, згідно з якою колись на Землі відбулася грандіозна повінь. Врятувалася від неї лише невелика група людей, які заздалегідь побудували великий корабель – ковчег. Першим місцем, куди він прибув, коли спала вода, виявилася вершина гори Арарат... А тепер уявимо собі, що мандрівники вирішили перетнути водою Кавказький хребет з півдня на північ. Чи існує при деякому певному рівні води шлях, який забезпечує можливість такої подорожі?

Ще один приклад. Нехай з металевого листа прямокутної форми безладно вибивають квадратні отвори так, що вони без перетинів можуть торкатися одне одного кутами або сторонами. До двох протилежних сторін листа підімкнуто джерело струму. При якому відношенні загальної площі отворів до площі листа припиняється електричний струм через лист? У місцях дотику електричний контакт порушується.

Ці задачі за умови, що гірська система або металевий лист мають достатньо велику протяжність по всіх напрямах, являють собою типові задачі перколяції.

Узагальнимо обидві умови у вигляді однієї задачі, тобто формалізуємо умову задачі.

Задача. Маємо плоску поверхню з достатньо великою протяжністю в обох вимірах. На цій поверхні у випадковий спосіб розміщено певні перешкоди

прямокутної форми. За якої кількості перешкод поверхня залишатиметься прохідною у горизонтальному (вертикальному) напрямі?

Для розв'язання цієї задачі вдамося до таких дій:

1. Покриємо поверхню сіткою з квадратними комірками, сторона яких визначається розміром елементарної (найменшої) перешкоди.
2. Введемо нове поняття – ***кластер***.

Означення: кластер (від англійського ***cluster*** – гроно, кетяг, китиця, група) – сукупність або група елементів – комірок сітки, – пов'язаних спільною стороною з найближчим сусідом за певною ознакою. Дві зайняті комірки належать одному кластеру, якщо вони пов'язані шляхом, який складається із зайнятих комірок.

Кластер, який простягається від однієї сторони сітки до протилежної, зветься ***пов'язуючим*** кластером.

Таким чином, сформульована вище задача зводиться до пошуку відповідей на такі питання:

1. Чи існує на розглядуваній поверхні пов'язуючий кластер?
2. За яких умов він виникає?

Почнемо з другого питання. Одразу ж зазначимо, що така задача має відверто статистичний характер. Справа в тому, що факт утворення пов'язуючого кластеру суттєво залежить від взаємної орієнтації перешкод, яка є випадковою. Може, наприклад, статися так, що за достатньо малої їх кількості один раз такий кластер з'явиться, а у решті, припустимо, десяти випадках (тобто при інших взаємних орієнтаціях перешкод) його не буде. Теорія протікання (перколяції) стверджує, що сам факт появи пов'язуючого кластеру визначається відношенням кількості перешкод N_p до повної кількості N_0 комірок на сітці, яке позначають X_c і яке має зміст концентрації перешкод (їх відносної долі):

$$X_c = N_p / N_0.$$

Таке значення X_c називають ***пороговим***.

Виникає природне питання: чи є величина X_c випадковою і не відтворюваною у різних дослідах, чи вона є цілком визначеною? Здоровий глузд підка-

зує, що, беручи ту саму сітку з тією самою кількістю перешкод, ми повинні одержувати різні значення для X_c. Але у якій мірі вони будуть відрізнятися одне від одного? Виявляється, що із збільшенням загальної кількості N_0 комірок значення X_c відрізнятимуться мало. Зокрема, теорія перколяції доводить, що при необмеженому зростанні N_0 середнє значення величини X_c прямує до певної границі. Адже у необмежено великої сітки зустрічатимуться безліч різноманітних комбінацій перешкод, так що вплив випадковості на коливання значень X_c буде зникаюче малим.

У спеціально проведених дослідах із квадратними сітками, які містили багато комірок, було встановлено, що поріг протікання за умов, схожих з нашими, становить приблизно 0,41.

Зіставлення результатів, одержаних різними методами, дозволяє припустити, що з точністю до другого знаку після коми для згаданих сіток X_c=0,41.

Розв’язування задачі може бути реалізовано за таким алгоритмом:

1. Виділити більшу частину графічного екрану під зображення сітки; розміри і положення цієї частини зафіксувати. Меншу частину (праворуч сітки) відвести під коментарі.
2. У режимі діалогу давати запит на кількість комірок: m – вздовж вісі X; n – вздовж вісі Y; а також k – кількість комірок, якими ми будемо імітувати вільні від перешкод ділянки сітки.
3. У залежності від значень змінних m, n і k обчислювати розмір однієї комірки.
4. У випадковий спосіб заповнювати (фарбувати) ці k комірок.
5. Дослідити намічені комірки з метою встановити, чи утворюють вони пов’язуючий кластер. Якщо такий кластер існує, перефарбувати його у новий колір і дати про це повідомлення; інакше – кольори залишити без змін і повідомити про відсутність такого кластеру.
6. У випадку появи пов’язуючого кластеру обчислювати і виводити на екран повідомлення про те, яку частину від загальної кількості комірок становить кількість перешкод у вигляді значення змінної X_c.

7. Пропонувати користувачу повторення роботи.
8. Обчислювати і виводити на екран середнє значення X_c та його середню квадратичну похибку як результат статистичного нагромадження відповідних значень.

Парам випадкових чисел, помноженим на масштабуючий коефіцієнт, надаватимемо зміст координат, за якими будемо заповнювати комірки сітки. Заповнення полягатиме у фарбуванні комірки.

Маючи перед собою зафарбовану сітку, яка одержується після виконання п. 4. алгоритму, виявити візуально наявність пов'язуючого кластеру порівняно легко при невеликій кількості комірок. Ситуація суттєво ускладнюється, коли їх багато. Саме цю непродуктивну та занадто стомлюючу роботу ми перекладаємо на комп'ютер.

Створення алгоритму для виявлення пов'язуючого кластеру є достатньо складним завданням, оскільки він не зводиться до зовні схожої відомої задачі про лабіринт. Дійсно, задача маркірування пов'язуючого кластеру навіть серед фахівців вважається достатньо складною. Один з порівняно простих, а тому і прийнятних варіантів ідеї такого алгоритму описано у відомій книзі Х. Гулда та Я. Тобочника.

Зміст роботи з програмою полягає в тому, що користувач поступово зменшує відносну долю перешкод, аж поки не виходе на поріг перколяції і одержує граничне числове значення X_c. Такі значення нагромаджуються і статистично обробляються. Результатом цієї обробки є середнє значення порогу X_c.

Нашу задачу не складно переформулювати на інше фізичне явище.

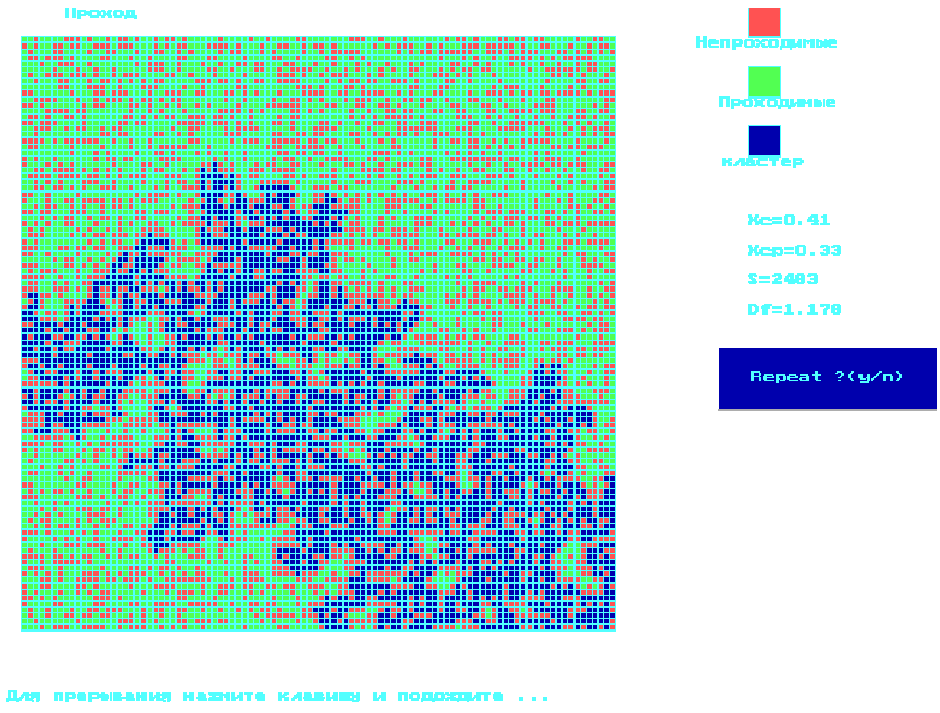
Розглянемо кристалічне тіло, що являє собою твердий розчин (суміш) магнітних і немагнітних частинок, хаотично розташованих у вузлах кристалічної решітки. Позначимо через X концентрацію (долю) магнітних частинок. Внаслідок взаємодії двох поруч розташованих магнітних комірок вони орієнтуються у просторі так, що їхні магнітні моменти співпадають. З'ясуємо, при яких значеннях X у тілі існує спонтанна намагніченість, тобто речовина є феромагнетиком.

Не дуже важко побачити, що це також задача перколяції. Вона зводиться до пошуку пов'язуючого кластеру для X, що наближатиметься до певної границі при необмеженому зростанні загальної кількості комірок. Ця умова автоматично задовольняється, оскільки реальну макроскопічну систему – розглядуване тіло – завжди можна вважати нескінченно великою. Адже один кубічний сантиметр її містить близько 10^{10} атомів.

За допомогою теорії протікання стає можливим вивчення переходів речовини з діелектричного стану у провідний, що має місце у напівпровідниках; руйнування гірських порід, коли через давній вік або від навантажень утворюються мікрощілини, які, зливаючись, розколюють камінь, і багато інших явищ.

Основне значення теорії протікання полягає в тому, що вона дозволяє не тільки відшукувати порогові значення, але й вивчає закони, які визначають поведінку різноманітних величин поблизу порога. Самим цікавим тут є те, що закони, за якими зникає пов'язуючий кластер, є універсальними для самих різних явищ.

Типовий перколяційний кластер на порозі протікання зображено на малюнку нижче:



Фрактальна розмірність перколяційного кластеру. Було встановлено, що поріг перколяції на нескінченій квадратній сітці виникає за умови, що

ймовірність появи «перешкод» становитиме 0,41. При цьому візуальний огляд одержаного кластеру показує, що він виявляється розрідженим, проте сам по собі цей факт ще не дає підстави стверджувати, що одержаний об'єкт є фракталом. Наявність розріджень можна розглядати як необхідну, але не достатню умову фрактальності об'єкту. Висновок про фрактальність можна зробити лише за умови, що фрактальна розмірність об'єкту D_f менша за евклідову D і при цьому не є цілим числом.

Наше подальше дослідження матиме на меті пошук відповіді на питання, чи є згаданий кластер фрактальним. У основу його покладено співвідношення (2), за яким визначалася фрактальна розмірність

$$D_f = \ln M / \ln L.$$

Робота ведеться за таким планом:

1. За допомогою програми на квадратній сітці розмірами LxL комірок генерується перколяційний кластер для порогу протікання (доля непрохідних комірок від загальної їх кількості становить 41%).
2. Підраховується кількість комірок, які належать кластеру. Знайдене число розглядається як маса M кластера.
3. До звітної таблиці заносяться значення L та M.
4. п.п. 1-3 виконуються для L=50, 60, 70, 80, 90 і 100 (для кожного L по декілька разів з наступним усередненням).

За даними таблиці будується графік залежності $M=M(L)$ у подвійному логарифмічному масштабі (на горизонтальній вісі відкладаються $\ln(L)$, а на вертикальній – $\ln(M)$). Нахил цього графіка характеризує фрактальну розмірність D_f.

Обчислюється фрактальна розмірність згідно виразу

$$D_f = \ln M / \ln L.$$

Для задовільного виконання п. (2) доведеться вдосконалити програму так, щоб підрахунок кількості комірок перколяційного кластеру виконувався автоматично. Адже це сама трудомістка частина роботи, а за значної кількості комірок вона стає взагалі неможливою.

В результаті дослідження було одержано D_f=1.7, що дозволило дістати

висновку: розглянуті кластери з пороговим значенням перколяції є фрактальними. Сподіваємося, що ви дійдете такого ж висновку, і побудована модель явища перколяції відкриє вам ще одну, несподівану грань.

§12.

Клітковий автомат «Сніжинка»

В навколишньому світі існує дуже багато різних процесів, які вимагають для опису складного і громіздкого математичного апарату. Наприклад, дивно, що молекули води «знають», як створювати складні симетричні сніжинки. Ніякий «архітектор» не керує будівництвом, і молекули не несуть інформацію про цю кристалічну структуру. Весь візерунок з'являється як наслідок найближчих взаємодій множини однакових частинок. Кожна молекула реагує лише на вплив найближчих сусідів, але розташування у визначеному порядку зберігається у всій структурі, яка складається приблизно з 10^{10} молекул.

Щоб краще зрозуміти цей процес, уявимо собі, що кожне місце, де молекула може бути розташована (так званий центр), визначається за допомогою елементарного комп'ютера. У міру росту кристалу комп'ютер спостерігає за сусідніми центрами і, знаходячи їх, встановлює за деяким певним правилом, чи повинно бути зайняте молекулою це місце, чи воно повинно пустувати. Такі обчислення ведуться для всіх центрів за одним і тим самим правилом.

Обчислювальна модель росту сніжинки є клітковим автоматом, тобто однорідною схемою багатьох ідентичних клітин – комірок, кожна з них має декілька можливих станів і її взаємодія відбувається лише з декількома сусідніми клітинами. *Правило обчислення наступного стану клітини може бути простим, але викликати складну еволюцію.*

Постановка задачі. Припустимо, що середовище (насичуюча водяна пара) моделюється двомірною шестикутною решіткою. Елементи цього середовища являють собою правильні шестикутні комірки, які можуть бути зафарбовані (лід) і не зафарбовані (пара), тобто значенням кожної комірки є номер відповідного кольору. Для визначеності та зручності домовимося «замерзлим» коміркам присвоювати значення 1, а решті – 0. Уявимо далі, що в центрі кожної комірки міститься центр конденсації і наступної кристалізації. Нарешті, нехай процес починається з однієї єдиної «замерзлої» комірки.

Розв'язання. Для імітації послідовних стадій у процесі утворення сніжинки скористаємося правилами кліткового автомату, які задаємо так:

1. Коли комірка «заморозилася», то вона вже не розтає.
2. Комірки, які розташовані на краях моделі при її рості, «замерзають» за умови, що у них не виявляється достатньої кількості «льодяних» сусідів, які б екранували їх від розсіювання тепла у оточуюче середовище.

Зазначені правила скоріш за все відносяться до класу (4), оскільки вони породжують стійку структуру істотної просторової та часової складності.

Правило 2 потребує уточнень:

1. На кожному кроці наступне значення довільної комірки на кордоні сніжинки залежить від сумарного значення всіх шести комірок, які є її безпосередніми сусідами. Якщо ця сума значень сусідів виявляється непарною (1, 3, 5), то комірка перетворюється на «лід» і набуває кольору, номер якого відповідає твердому стану речовини, тобто 1. В протилежному випадку комірка залишається у пароподібному стані і зберігає значення 0. В такий спосіб визначаються «кандидати» на фарбування. Саме ж фарбування відбувається лише після виявлення всіх «кандидатів» у граничному шарі.

2. Оскільки клітковий автомат вимагає на кожному кроці одночасного поновлення значень всіх комірок, то й відповідний алгоритм повинен на кожному кроці передбачати можливість витирати попередній малюнок і починати обробку всіх клітин від самого початку і до чергового граничного шару. В цьому контексті зазначимо таке: з фізичних міркувань зрозуміло, що процес росту сніжинки відбувається саме на граничному шарі, а не кожного разу від початку. Отже, щоб при роботі програми за таким алгоритмом спостерігач не помічав перемальовувань, слід використовувати швидку машину. На повільній машині краще вже забудовувати лише черговий граничний шар, залишаючи всі попередні без змін. Так само треба було б зробити і за умови економії часу роботи та оперативної пам’яті машини.

3. Детальний аналіз Правила 2, заснований на його покроковому виконанні привів нас до висновку, що потреба у перевірці всіх шести сусідів кожної

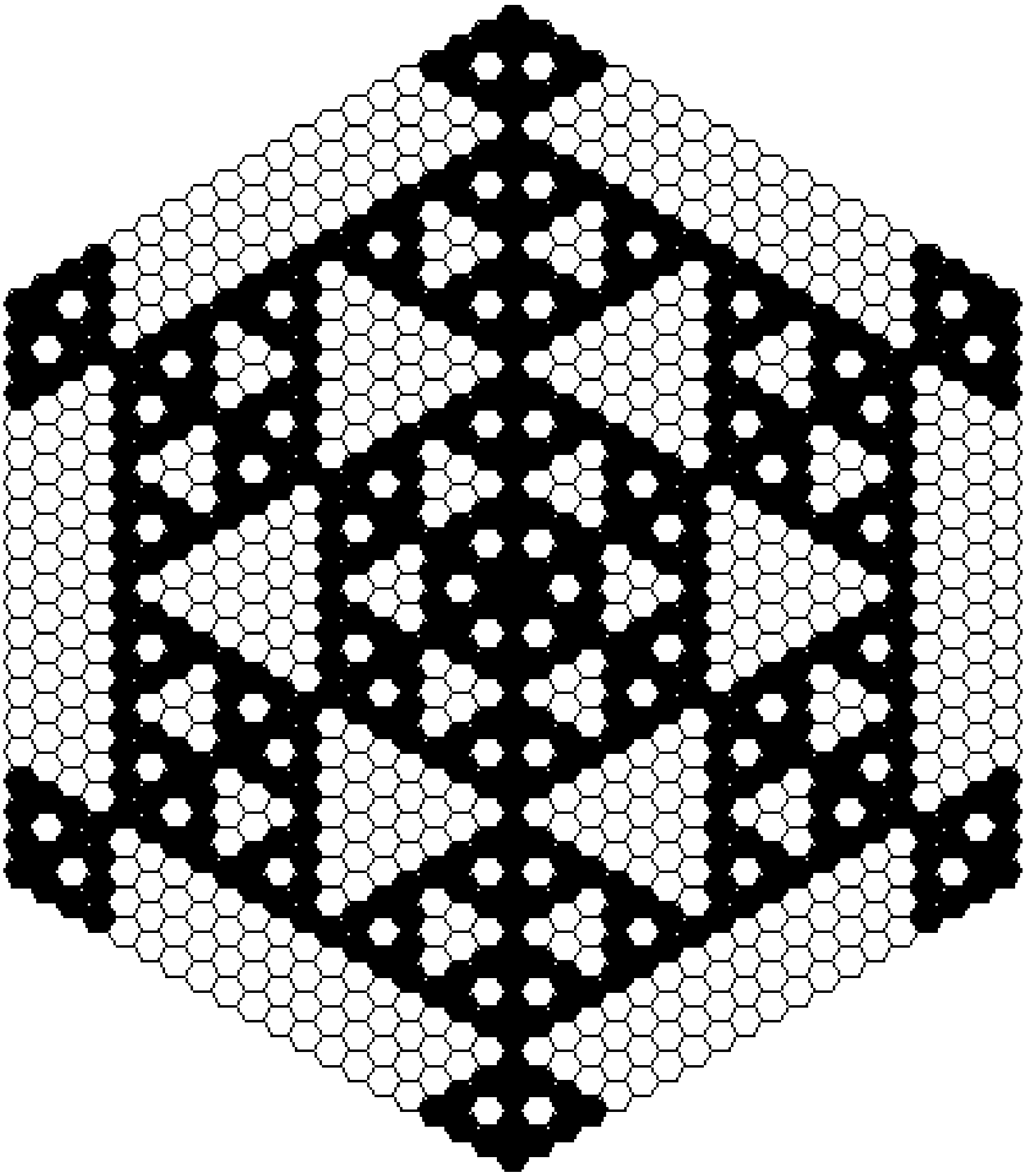
комірки у граничному шарі виявляється надлишковою. Як це видно з побудови, будь-яка комірка з такого шару фактично може мати лише одного або двох «замерзлих» сусідів у залежності від того, де вона знаходиться. Якщо вона належить діагоналі граничного шестикутника, то кількість «замерзлих» сусідів дорівнює одному, у решті інших випадків – нулю або двом. Проте з естетичної точки зору обхід шістьох сусідів нам видається більш привабливим, і ми вирішили залишити алгоритм роботи з моделлю таким, яким він є.

Фахівці стверджують, що ріст сніжинок можна також описати за допомогою системи диференційних рівнянь, однак більш проста модель на основі кліткового автомату, мабуть, краще передає суть процесу утворення візерунку. Кліткові автомати широко використовують і для опису біологічних систем, де складні моделі росту подають простими алгоритмами для кліткових автоматів. Як було зазначено вище, незважаючи на те, що правила, за якими працює кліткový автомат, можуть бути простими, та вони із плином часу здатні призводити до складної еволюції моделі.

Однією з головних проблем теорії кліткових автоматів є пошук відповіді на питання: а чи існує взагалі можливість *заздалегідь* передбачити кінцевий стан у еволюції кліткового автомату, тобто стан його після певної кількості N кроків, не виконуючи самі ці кроки? Якщо система є достатньо простою, то, маючи відомості про її початковий стан, не складно розрахувати й стан її на довільному наступному кроці. Для нашої сніжинки, мабуть, це ще можливо (?), але вже не просто. Адже у цьому випадку довелося б знайти алгоритм, який за номером шару дозволяв би заповнювати кожну комірку з цього шару. Взагалі ж відповідь на поставлене запитання є *негативною*. Це означає, що *для складних систем результат еволюції можливо одержати лише шляхом прямої покрокової імітації.* Не існує жодного способу передбачити цей результат, треба просто чекати, коли еволюція відбудеться. Можливо, саме тому для дослідження складних фізичних систем сьогодні відомо лише два способи: або фізичний (натурний) експеримент, або ж обчислювальний експеримент з моделлю досліджуваної системи.

Нарешті, ще один принциповий факт, повз який не можна пройти. Добре відомо, що комп’ютер здатний обробляти інформацію хоч і з великою швидкістю, але *послідовно*. Фізичні ж системи (навіть наша сніжинка) обробляють інформацію *паралельно*, а тому виникає потреба у такому описі, який був би паралельним за своєю природою. Сьогодні ще не існує загальної структури для опису паралельних процесів. Коли ж її буде створено, стане можливим більш ефективний опис фізичних явищ на більш високому рівні.

Типовий результат роботи кліткового автомату «Сніжинка» зображено на малюнку нижче:



§13.

Клітковий автомат «Хижак–жертва»

Продовжуючи тему кліткових автоматів, перейдемо до однієї з задач математичної популяційної екології і розглянемо украй ідеалізовану ***модель «Хижак–жертва»***. Весь біоценоз середовища, що являє собою двомірне водоймище, зведемо всього до двох популяцій – хижаків (щуки) та жертв (карасі).

Опишемо правила «гри»:

1. Всі риби плавають, переміщуючись за один такт еволюції з тієї комірки, де вони знаходились, в одну із сусідніх. При цьому карась переходить з однаковою ймовірністю у будь-яку вільну комірку, а щука спочатку визначає, чи є поруч карась, і якщо це так, то пливе саме туди і з'їдає свою жертву. Якщо ж поруч зі щукою всі комірки вільні, то вона з однаковою ймовірністю займає будь-яку з них.

2. За один такт еволюції вік всіх риб збільшується на 1. При досягненні певного зрілого віку (для кожного виду свого) кожна риба починає через рівні проміжки часу продукувати потомство, з якого виживає лише одна осіб. Діти спочатку розташовуються поруч з мамашею, а потім поводять себе за законами дорослих.

3. Якщо щука протягом деякої кількості тактів не їла, то вона гине від голоду. Карась може загинути лише в зубах щуки.

4. Ставок має кінцеві розміри, а щоб риби не вкидалися на берег, протилежні краї ставка склеєні.

У випадковий спосіб

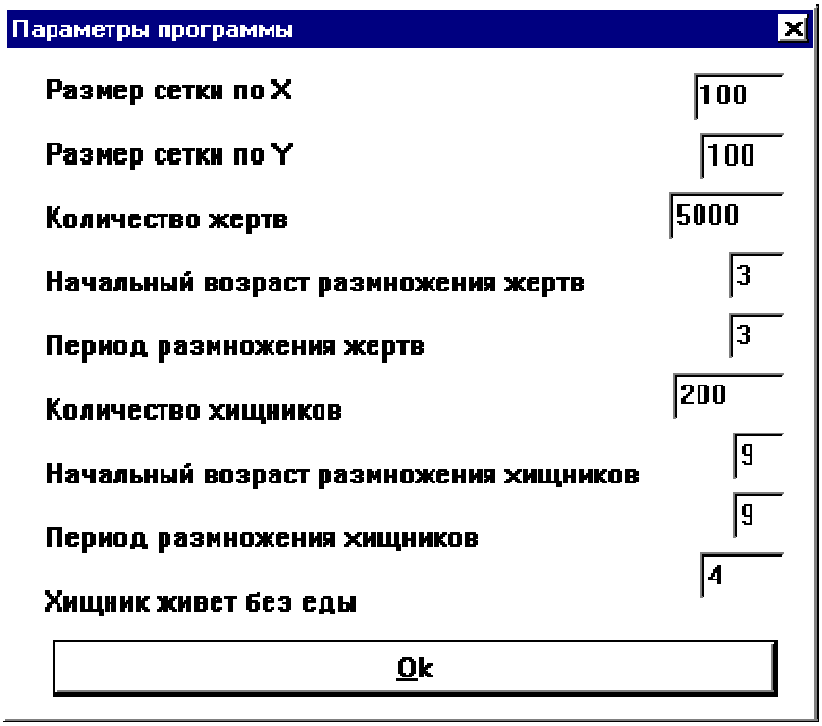
- розкидаємо щук та карасів по ставку;
- встановимо кожній рибі вік від нуля (дитя) до зрілого і у ставку з'являється різновікова компанія.

Далі встановлюємо

- для кожної щуки період, із закінченням якого вона вмирає, якщо не з'їсть карася;

– вік, з якого починається розмноження особів (для кожного виду свій);

– періоди розмноження $T_к$ и $T_щ$:

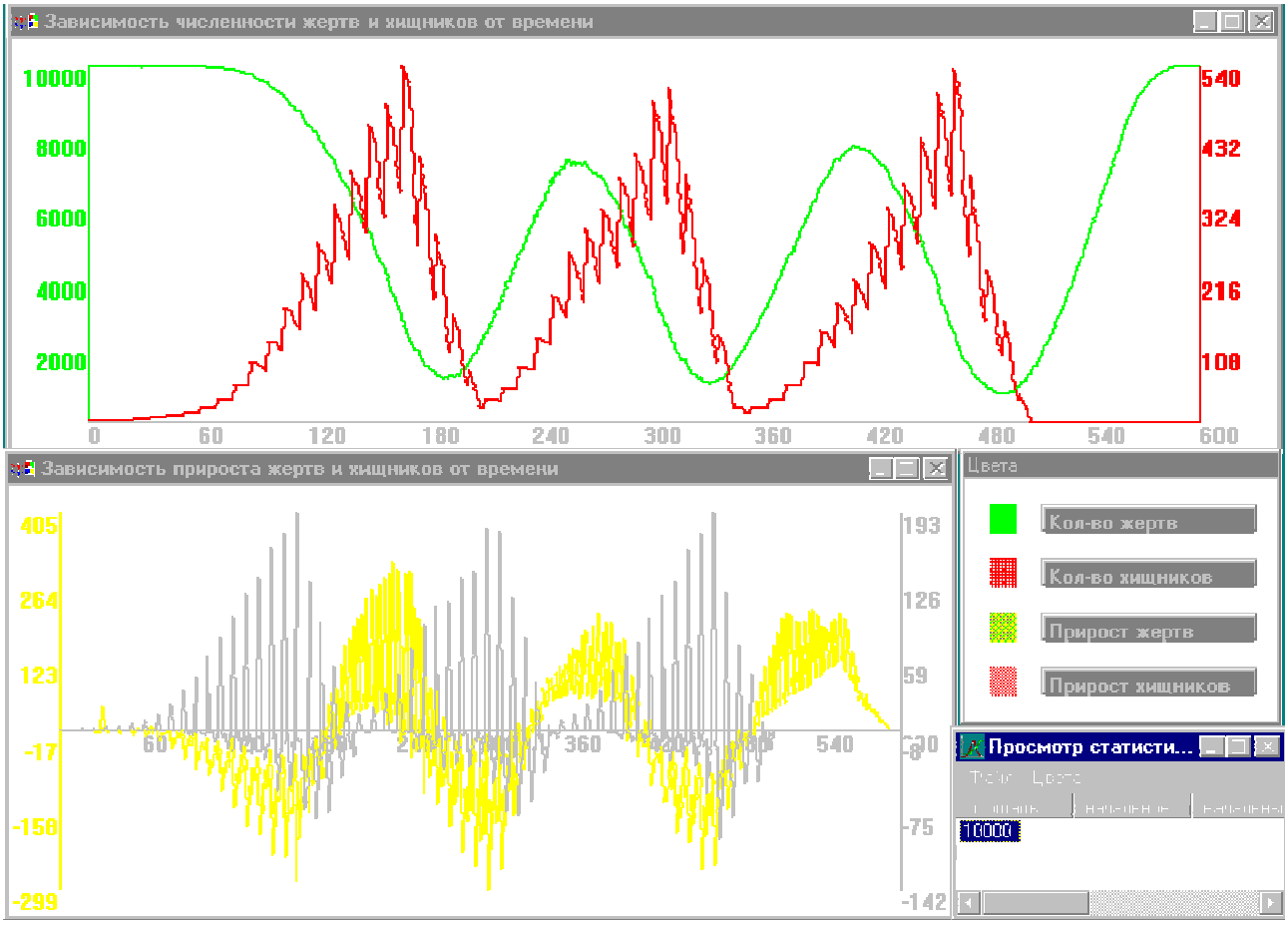


Декілька слів про сценарій, за яким живе така біосистема.

Починається перший такт еволюції. Спочатку на один крок переміщуються по черзі всі карасі. Ті з них, котрим підійшов строк, розмножуються. Далі починають полювання щуки. Наприкінці такту підводиться підсумок: виключаються щуки, що померли від голоду, і карасі, що їх з'їли щуки, і додаються риби, які народилися. Після цього починається наступний такт і т.д. В результаті відслідковується зміна у часі чисельності популяцій:

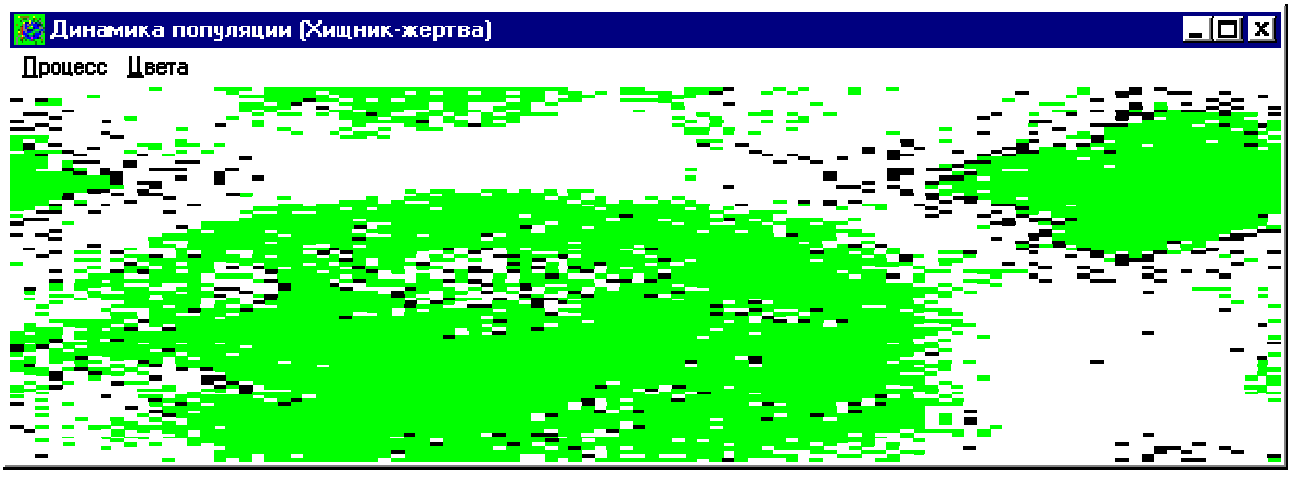
Таким є у загальних рисах алгоритм роботи програми кліткового автомату «Хижак–жертва». Змінюючи параметри, можна одержати цікаві результати:

1. Чисельності обох популяцій можуть коливатися з однаковими періодами T, але із зсувом фази приблизно на чверть періоду, причому, у цих випадках період коливань чисельності пропорційний квадратному кореневі з $T_к \cdot T_щ$.



2. Досить часто коливання можуть збиватися і тоді їхні періоди змінюються у широких межах.
3. У деяких випадках щуки, виявившись волею долі далеко від карасів, усі гинуть. Тоді чисельність карасів зростає, аж поки вони не займуть весь ставок. Зазначимо, що така імовірнісна модель дає майже ті самі результати, що й відома модель Вольтерра–Лотки, записана у вигляді рівнянь.

Типовий результат роботи кліткового автомату «Хижак–жертва» зображено на малюнку нижче:



§14.

Розмивання берегової лінії

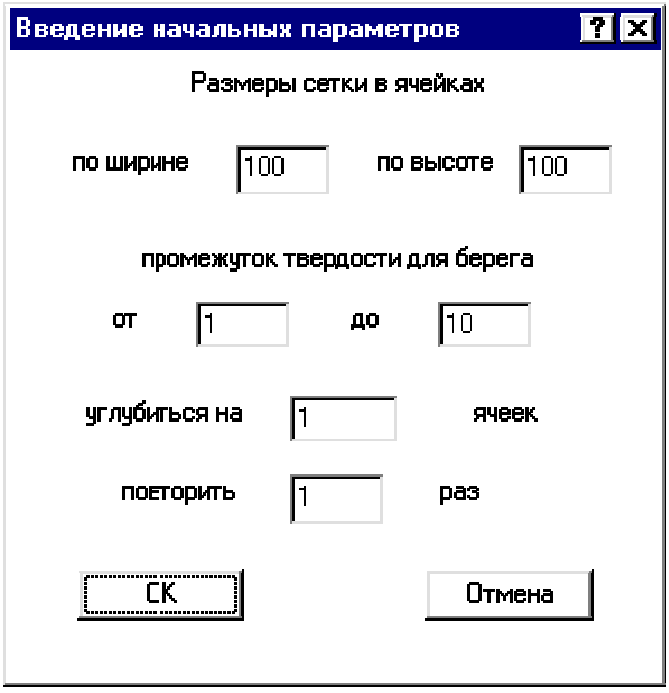
«Спочатку зітворив Бог небо й землю. Земля ж була безвидною й пустою, і темрява над безоднею, і Дух Божий носився над водою» (Біблія).

Ця живописна картина, намальована у перших віршах Книги Буття, послугує нам відправною точкою для дослідницької експедиції під назвою “Узбережжя”.

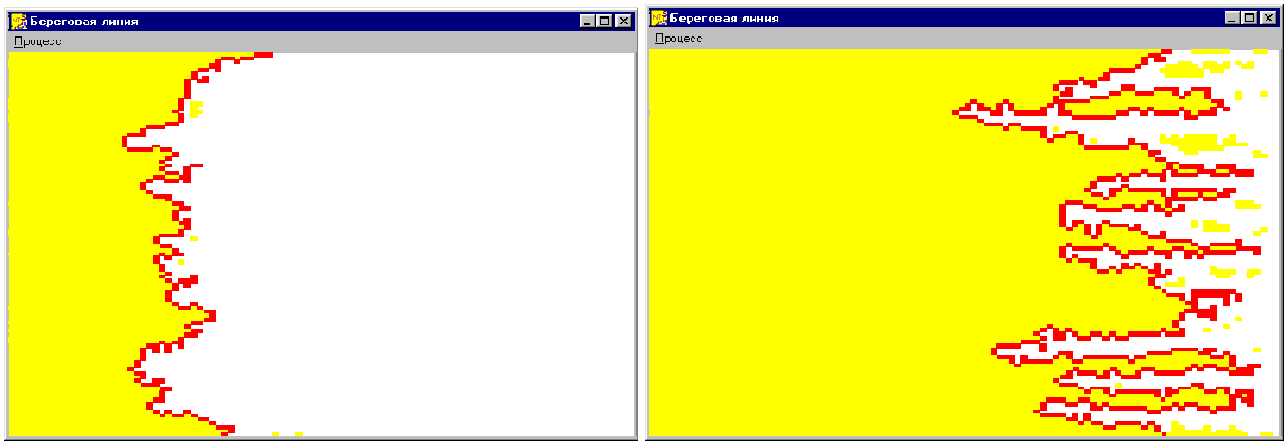
Наш ковчег кидає якір неподалік від первісного і поки що ідеально рівного берега: ми припускаємо тут, що Господь, який щойно закінчив свої труди, не попіклувався про створення посіченої берегової лінії у її сучасному варіанті. Він, напевно, мав право вважати себе творцем законів, а не форм і, не прагнучи різноманітності, прокреслив уздовж лінійки пряму, поклавши по один бік від неї сушу, а по другий залишивши воду. Дослідники, які знаходяться на ковчезі, є свідками історичного моменту. На лінії горизонту з’являються перші хвилі. Безжалісний вітер кидає їх у різні сторони, але вони неухильно наближаються до берега. Спостерігачі на судні помічають, що будь-яка хвиля на кожному кроці свого руху випадково обирає один з трьох можливих напрямів до берега. Такий характер руху хвиль зберігається до моменту їх зіткнення з берегом. Хвиля, що ударяє елемент берега, або видаляє його, якщо сила її удару достатня для руйнування даної берегової породи, або, відбиваючись, продовжує рух за тим самим законом.

Окреслимо алгоритм роботи програми, яка моделює цей процес. Із самого початку для кожного елемента первісного берега обирається ціле випадкове число з деякого наперед заданого інтервалу. Воно є критерієм твердості і задає кількість хвиль (ударів), необхідних для руйнування даного берегового елементу. Таким чином, усі можливі значення умовної твердості берегових порід містяться між границями зазначеного інтервалу. При кожному ударі хвилі з критерію твердості відповідного елемента берега віднімається одиниця. Якщо після цього критерій стає рівним нулеві, хвиля видаляє даний елемент берега і,

виконавши свою місію, зникає. Замість «відпрацьованої» хвилі на лінії горизонту одразу ж з'являється нова. Якщо критерій не дорівнює нулеві після віднімання, хвиля знову обирає один з трьох випадкових напрямів руху. Програма має дозволяти вводити будь-які розумні межі інтервалу для критеріїв твердості окремих елементів суші, а також у широких межах варіювати розміри цих елементів – комірок сітки.

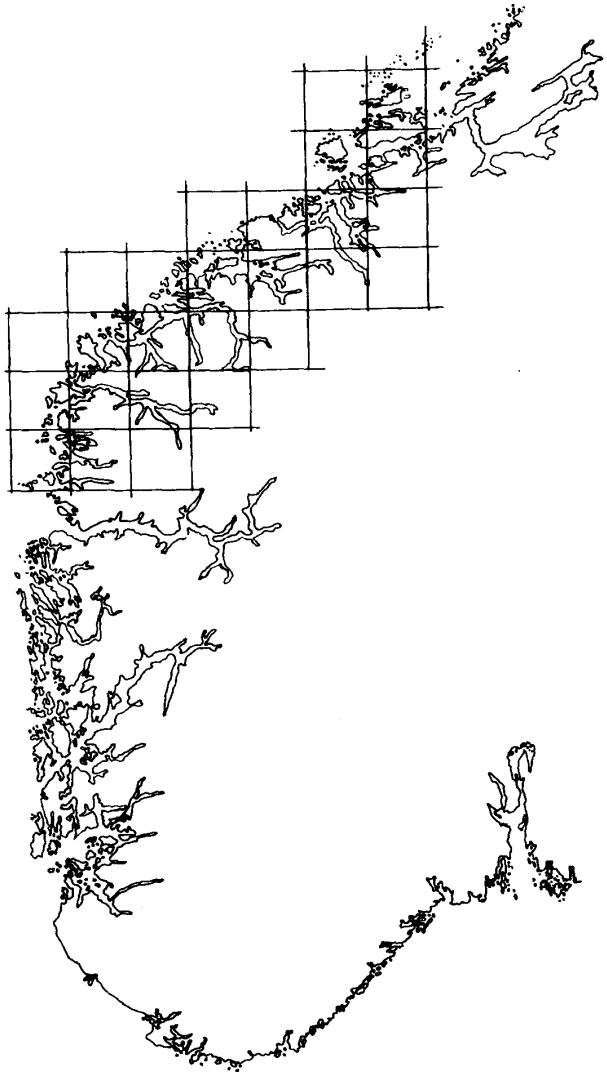


Змінюючи межі критеріїв, користувач може одержувати на екрані зображення різноманітних за виглядом берегових ліній – від піщаних плесів до глибоких скельних фіордів (див. малюнок нижче). Зокрема, у наших експериментах було встановлено, що задовільні обриси фіордів виникають, коли мінімальним критерієм брати 1, а максимальний критерій брати однаковим для сіток різних розмірів (наприклад, 100).



Досліджуваний об’єкт, яким у розглядуваному випадку є берегова лінія, а точніше – її довжина, є фрактальним. У якості фракталів ми можемо розглядати також і острови, і вимиту частину берега (тоді острови відіграватимуть роль порожнин).

Зазначимо, що розглянута модель являє собою *модель Агрегації з Обмеженням Дифузії (АОД)*. Суть її полягає в тому, що модельна частинка, яка дифундує у досліджуване середовище, блукає у ньому, аж поки не зіткнеться з існуючим там первісним тілом – агрегатом. Як тільки це відбувається, частинка приєднується до агрегату, доповнюючи його. В результаті росте новоутворення – кластер, структура якого і є предметом обговорення у даній моделі. Щоправда, у нашому випадку використано інверсну агрегацію: блукаюча частинка при стиканні з агрегатом не приєднується до нього, а, навпаки, видаляє елемент агрегату.



То ж повернемося до наших дослідників – пасажирів ковчега, – яких ми залишили у момент споглядання початку творення берегів і повернемо їх у сьогодення. Не дуже важко зрозуміти, чому обриси берегових ліній на сучасній географічній карті світу мало схожі на первісні: за довгий час робота моря далася взнаки. Якнайбільший інтерес для учасників експедиції «Узбережжя» на сучасній карті являє узбережжя Норвегії: воно настільки вигадливе, що можна тільки дивуватися на цей витвір природи. На шляху до цієї країни, вони вирішують поки що за допомогою карти одержати відповідь на питання: «Якою є довжина берегової лінії Норвегії?»

У масштабі карти добре видно глибокі фіорди на західному узбережжі. Більш дрібні деталі обрисів узбережжя на північному сході від південної кінцівки розрізнюються вже гірше. Проте, розглядаючи більш докладні карти для плавання вздовж цієї ділянки, можна переконатися, що берегова лінія тут має практично такий самий вигляд, як і на західному узбережжі (щоправда, у зменшеному масштабі).

Перш, ніж відповісти на головне питання, учасникам експедиції необхідно вирішити ще декілька проблем: чи варто включати до берегової лінії острови? як бути з річками? у якому місці фіорд перестає бути фіордом і де саме він переходить у річку? Припустимо, що вони зможуть знайти задовільні відповіді на всі питання такого роду. Та все одно найбільш принципове утруднення залишиться. Справа у тому, що коли вони нададуть розхилові циркуля довільного розміру, який відповідатиме d км, і підрахують кількість кроків $N(d)$, які потрібно зробити, щоб пройти циркулем уздовж карти з кінця в кінець все узбережжя, то у такому разі за довжину берегової лінії треба буде прийняти величину

$$L=N(d)-d.$$

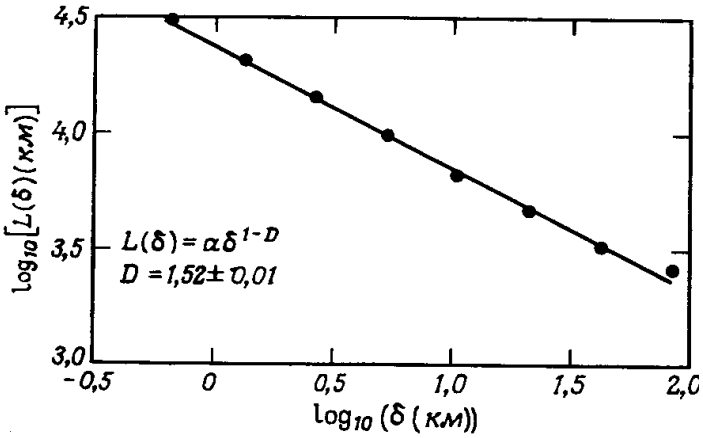
Деякі учасники експедиції вважають таку оцінку неточною і пропонують зменшити довжину d одиничного відрізку, а далі повторити все спочатку. Інші заперечують, говорячи, що в такому разі у довжину берегової лінії хоч і ввійдуть найбільш глибокі фіорди, але південно-східне узбережжя, як і раніше, буде долатися за кілька кроків. Стає зрозумілим, що для ще більш точного підрахунку можуть статися у нагоді такі карти, якими користуються сусіди для розв’язання питань про те, де повинен проходити кордон між їхніми земельними ділянками. Отже, при вирішенні питань такого роду уточнення можна вносити нескінченно, як це робили у відомому фільмі герої Фернанделя і Тото. Крім того, при користуванні циркулем неодмінно виникатимуть проблеми з островами та річками. Але існує принципове обмеження знизу – мінімальний розхил циркуля.

Крім обходу за допомогою циркуля, існує альтернативний спосіб: вкрити

карту сіткою з комірками розміром dxd кожна. Кількість $N(d)$ таких комірок, необхідних для покриття берегової лінії на карті, приблизно дорівнює кількості кроків для обходу берегової лінії на цій самій карті за допомогою циркуля з розкриттям d. Зменшення d призведе до збільшення кількості комірок, які укладатимуться у довжину берегової лінії.

Якщо берегова лінія Норвегії має цілком певну довжину L_N, то можна сподіватися, що кількість кроків циркуля або кількість квадратних комірок $N(d)$ для покриття берегової лінії на карті, буде обернено пропорційною d, а довжина $L(d)=N(d)–d$ при зменшенні d прямуватиме до постійної величини L_N. Проте, цим сподіванням не судилося здійснитися і, отже, вони є марними.

З малюнка видно, що при зменшенні розміру d кроку виміряна довжина L зростає.



Графік на цьому малюнку виконано у подвійному логарифмічному масштабі, і він показує, що при зменшенні d виміряна довжина берегової лінії аж ніяк не прямує до постійного значення. Навпаки, виміряна довжина чудово вписується у наближену формулу

$$L(d)=ad^{1-D}.$$

Для звичайної кривої можна було б чекати, що за достатньо малих d $a=L_N$ і показник D=1. Але для берегової лінії Норвегії, як це видно з графіка, D=1,52.

Берегова лінія – це фрактал з фрактальною розмірністю D_f. Якби учасники експедиції визначали довжину берегової лінії Великобританії, то вони дійшли того самого висновку, але для фрактальної розмірності одержали б D=1,3.

Практична робота (обчислювальний експеримент) з використанням програми моделювання розмивання берегової лінії ведеться за таким планом:

1. На квадратній сітці розмірами NxN комірок наочно будується морське узбережжя, зафарбовується берегова лінія і програмно на екран виводиться кількість комірок сітки, що покривають берегову лінію, тобто її довжина L.

2. До звітної таблиці заносяться значення L та N.

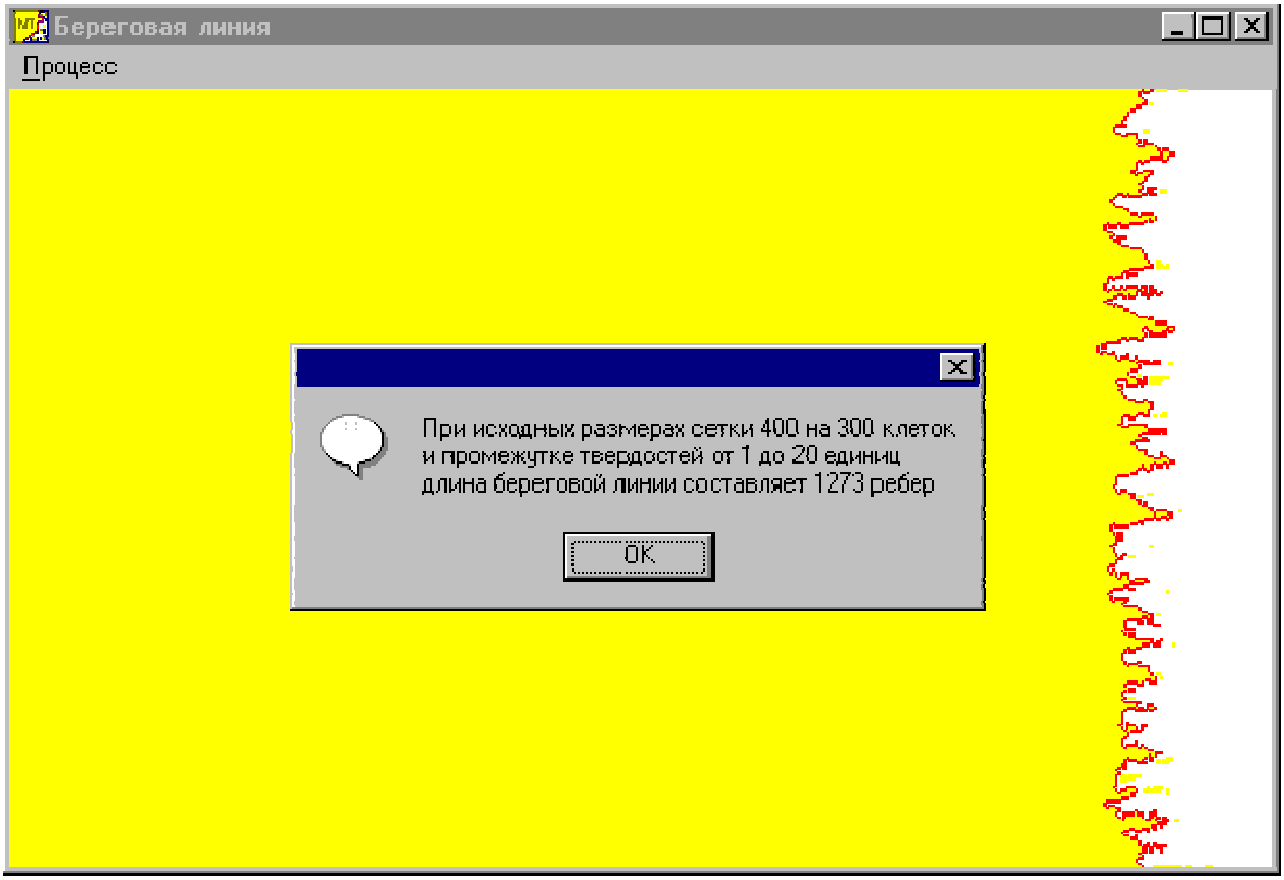
3. п.п. 1-2 виконувалися для N=100, 90, 80, 70, 60, 50 (для кожного N по декілька разів з наступним усередненням).

4. За даними таблиці будується графік залежності $L=L(N)$ у подвійному логарифмічному масштабі (на горизонтальній вісі відкладалися ln N, а на вертикальній – ln L. Нахил цього графіка характеризує фрактальну розмірність D_f.

5. Фрактальна розмірність обчислюється згідно виразу

$$D_f=\ln L/\ln N.$$

Типовий результат роботи програми зображено на малюнку нижче:



§15.

Електроліз на плоскому катоді

Якщо у попередній моделі зробити декілька незначних змін, то можна перетворити її на модель електролізу, яка здатна демонструвати процес електролітичного відновлення металу. Під час роботи відповідної програми роль хвиль виконуватимуть іони металу. Вони також зазнаватимуть випадкових відхилень при своєму русі до катоду електролітичної вани. Катод у вигляді тонкої лінії розташовано на нижньому краї екрану. Як тільки черговий іон торкатиметься або катоду, або частинки металу, яка вже осіла на нього, він, відновлюючись, приєднується до агрегату, а замість нього з протилежного боку з'являється новий, що дифундує у вану. При цьому на кожному кроці слід перевіряти наявність частинок у безпосередній близькості від нього. Якщо їх нема, іон рухатиметься далі, якщо ж сусіда визначено – агрегує з ним.

Урізноманітнити дослідження можна, зменшуючи розмір катода до зовсім короткого відрізку.

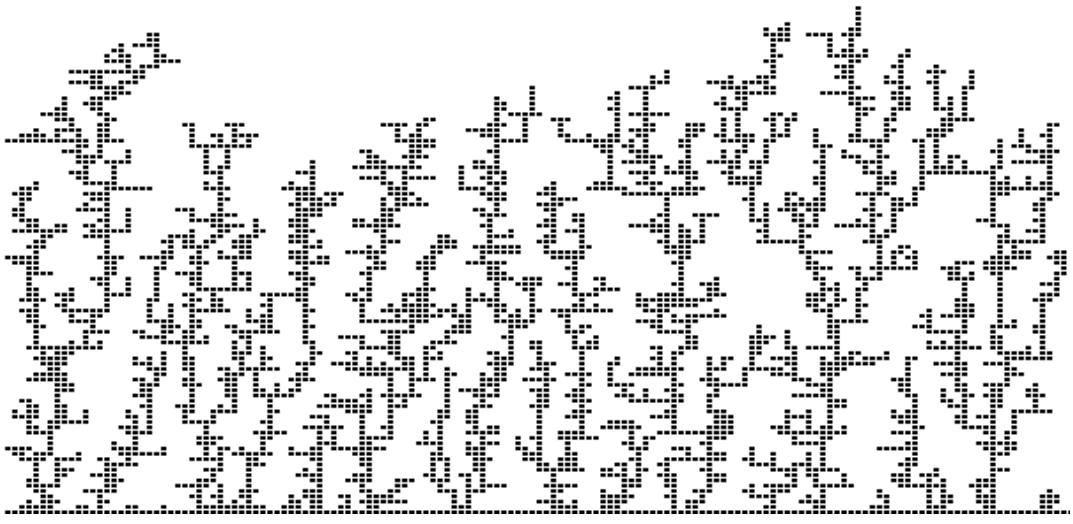
Для визначення фрактальної розмірності даного об’єкту скористаємося виразом

$$D_f = \ln N / \ln L,$$

тут N – кількість частинок у кластері, L – розмір сітки.

Для цих кластерів одержуємо D_f=1,7. Отже, вони є фракталами.

На малюнку зображено типовий кластер, утворений на сітці 100x100:



§16.

Електроліз на точковому катоді

Розглянемо електролітичну вану з точковим катодом, розташованим у центрі кола, яке відіграватиме роль анода. Якщо напруженість електричного поля буде достатньо малою, то кожен окремий іон можна розглядати як частинку, яка дифундує у просторі між електродами. Дійсно, силова дія катодного потенціалу повинна обмежуватися деяким радіусом: іони, що наблизилися до катоду ближче за цей радіус, екранують катод від решти іонів. Із зменшенням обраного потенціалу величина радіусу взаємодії може ставати достатньо малою. Такий рух триває до тих пір, поки іон не торкнеться катоду – первісного агрегату (затравки) – і не агрегує з ним.

Будь-яка спроба розв'язати задачу «в лоб» виявляється майже безнадійною внаслідок дуже малої ймовірності агрегації для частинки, яка знаходиться на помітній відстані від катоду і здійснює випадкові блукання. Це принципове утруднення долає метод *ренорм-групи.*

Ідея методу полягає у такому. Нехай у початковий момент часу спостережувана електролітична вана знаходиться на такому віддаленні від експериментатора, що видається дуже малою і містить лише невелику кількість розрізнюваних і практично точкових дільниць. За кожен такт часу іон, що безладно рухається у вані, долає відстань від однієї дільниці до іншої у випадковому напрямі. Оскільки спостережуваних дільниць мало, ймовірність агрегації велика і невдовзі цей іон приєднується до кластеру. У міру зростання кластеру ми немов би наближаємося до вани і збільшуємо масштаб її подання так, щоб завжди залишалося не дуже багато периферійних елементів, у яких може блукати ще не агрегований іон.

У практичній реалізації це означає, що навколо початкового кластеру – катода обирається прямокутне вікно, верхня сторона якого дещо вища за найвищий елемент кластеру, нижня сторона – нижча за найнижчий елемент, ліва сторона – лівіша найлівішого елемента, а права – правіша найправішого. У ви-

падкових точках цього вікна з'являються дифундовані іони, які рухаються після появи у випадкових напрямах, відбиваючись від стінок вікна. Самого ж вікна не видно. Якщо приєднаний іон виявляється вище попередньої найвищої точки кластеру, верхня сторона вікна відсувається на крок вгору. Так само поводять себе решта три сторони.

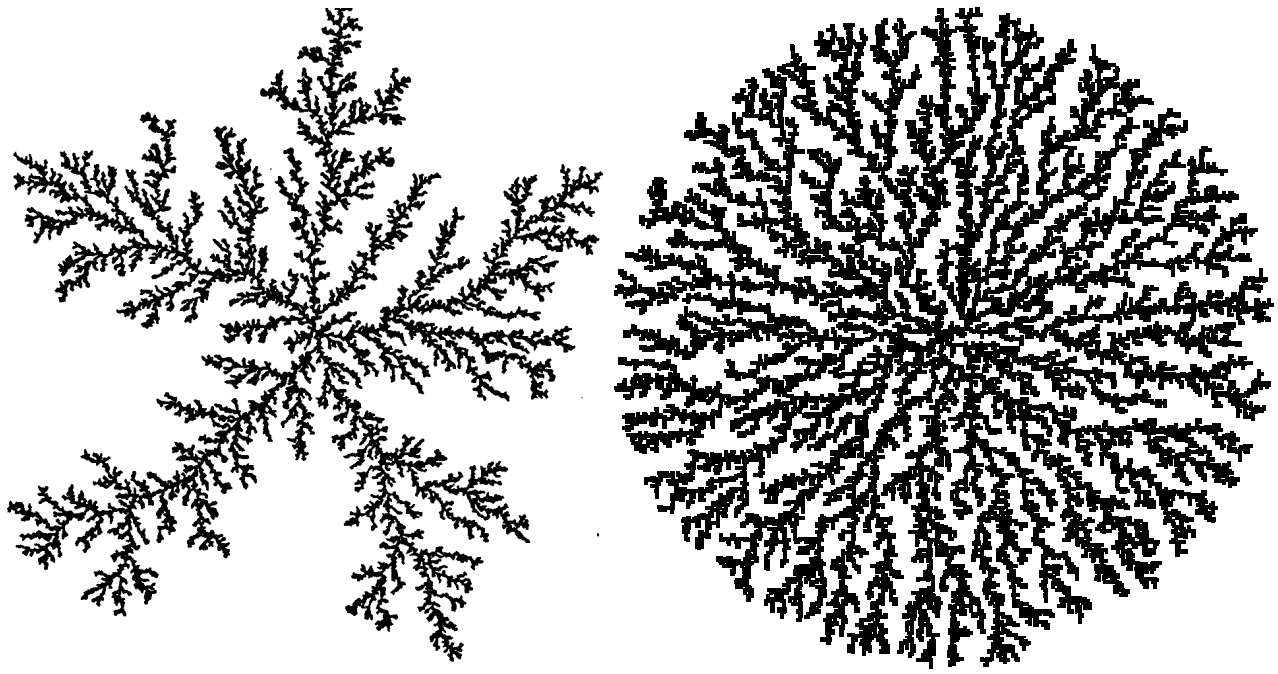
Фрактальна розмірність утворюваних тут об'єктів визначається через функцію залежності $N=N(R)$, де N – кількість частинок у колі радіуса R із центром у затравці (первісному агрегаті):

$$D_f=\ln N/\ln R.$$

Якщо сітка квадратна, то R можна розглядати як радіус вписаного у неї кола і, отже, $R=L$ (L – розмір сторони сітки). Тоді $D_f=\ln N/\ln L$.

Генеруючи кластери на сітках різних розмірів і підраховуючи кількість частинок у кластері (програма автоматизує підрахунок), можна обчислити фрактальну розмірність утворених об'єктів. За нашими даними, для кластерів такого типу фрактальна розмірність приблизно становить $D_f=1{,}7$.

Якщо, зберігаючи блукання, збільшувати ймовірність просування іону у напрямі катоду, то можна імітувати зміну катодного потенціалу і одержувати кластери різного виду:

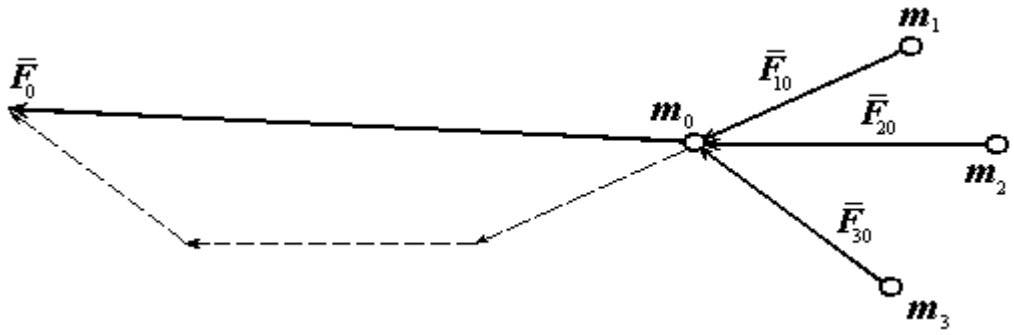


§17.

Задача багатьох тіл

У будь-якому курсі моделювання чільне місце посідають динамічні моделі, що описуються законами Ньютона, модельними потенціалами тощо. Відомо, що велику кількість динамічних задач засобами елементарної математики аналітично розв'язати неможливо. Проте чисельне розв'язання – методом скінчених різниць – цілком доступне і не викликає утруднень. Р. Фейнман, працюючи з молодшими студентами, для розв'язання динамічних задач цим методом на своїх лекціях використовував таблиці, які «є, звичайно, просто зручною формою запису результатів, отриманих з рівнянь, і фактично повністю замінюють їх». Однією з задач, яку він пропонував студентам, була задача багатьох тіл, реалізацію якої у електронних таблицях ми детально проаналізуємо з метою створення алгоритму роботи відповідної програми.

Нехай наша система складається з чотирьох тіл з масами m_0, m_1, m_2 та m_3 відповідно:



Згідно закону всесвітнього тяжіння, сила, що діє на тіло з масою m_i з боку всіх інших, дорівнює векторній сумі парних взаємодій:

$$\boldsymbol{F}_i = \sum_{\substack{j \\ (i \neq j)}} \boldsymbol{F}_{ji} = \sum_{\substack{j \\ (i \neq j)}} G \frac{m_j m_i}{r_{ji}^3} \boldsymbol{r}_{ji} = G m_i \sum_{\substack{j \\ (i \neq j)}} \frac{m_j}{r_{ji}^3} \boldsymbol{r}_{ji} \quad (1)$$

Ця сила, згідно другого закона Ньютона, надає тілу прискорення:

$$\boldsymbol{F}_i = m_i \boldsymbol{a}_i \quad (2)$$

Прирівнюючи формули (1) та (2), отримуємо формулу для визначення прискорення:

$$m_i \boldsymbol{a}_i = G m_i \sum_{\substack{j \\ (i \neq j)}} \frac{m_j}{r_{ji}^3} \boldsymbol{r}_{ji} \Rightarrow \boldsymbol{a}_i = G \sum_{\substack{j \\ (i \neq j)}} \frac{m_j}{r_{ji}^3} \boldsymbol{r}_{ji} \quad (3)$$

Зауважимо, що у нашому випадку ми, взагалі кажучи, не можемо користуватися класичними формулами для визначення швидкості та координати, бо, згідно (3), прискорення залежить від координати, тобто *рух тіла під дією сили тяжіння не є рівноприскореним.* Як можна подолати цю перешкоду? Скористаємося чисельним методом – розіб'ємо весь час руху тіла на дуже малі проміжки і будемо вважати, що на кожному з цих елементарних проміжків прискорення є постійним.

Нехай на початку руху i-те тіло має координати (x_{i0}, y_{i0}), прискорення $\boldsymbol{a}_{i0}$ та швидкість $\boldsymbol{v}_{i0}$. Наприкінці першого проміжку часу тіло набуде прискорення $\boldsymbol{a}_{i1}$ за (3); його швидкість обчислюватиметься за формулою:

$$\boldsymbol{v}_{i1} = \boldsymbol{v}_{i0} + \boldsymbol{a}_{i0} \Delta t \quad (4)$$

Знаючи швидкість, ми можемо обчислити нові координати тіла:

$$\begin{aligned} x_{i1} &= x_{i0} + \upsilon_{i1x} \Delta t \\ y_{i1} &= y_{i0} + \upsilon_{i1y} \Delta t \end{aligned} \quad (5)$$

Змінюючи i, визначаємо прискорення, координати та швидкості всіх інших тіл наприкінці першого проміжку часу. Повторюючи цю процедуру, ми врешті-решт одержимо їх координати, дискретизовані проміжком часу Δt, що дасть нам змогу побудувати графіки їх руху. Для тестування візьмемо спочатку лише два тіла, а далі вдосконалюватимемо нашу модель, поступово вводячи до розгляду інші тіла.

Отже, ми можемо записати остаточний

АЛГОРИТМ РОБОТИ З МОДЕЛЛЮ:

1. Створимо електронну таблицю за таким зразком:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	a_{1x}	a_{1y}	a_{2x}	a_{2y}	υ_{1x}	υ_{1y}	υ_{2x}	υ_{2y}	x_1	y_1	x_2	y_2	*Дано:*	
2													G=	
3													Δt=	
4													m_1=	
5													m_2=	
6													υ_{1x0}=	

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
7													υ_{1y0}=	
8													υ_{2x0}=	
9													υ_{2y0}=	
10													x_{10}=	
11													y_{10}=	
12													x_{20}=	
13													y_{20}=	
14														

2. Занесемо у другий рядок початкові дані:

комірки	***формули / числа***
A2	=$N$2*$N$5*(K2-I2)/СТЕПЕНЬ(КОРЕНЬ((K2-I2)^2+(L2-J2)^2);3)
B2	=$N$2*$N$5*(L2-J2)/СТЕПЕНЬ(КОРЕНЬ((K2-I2)^2+(L2-J2)^2);3)
C2	=$N$2*$N$4*(I2-K2)/СТЕПЕНЬ(КОРЕНЬ((K2-I2)^2+(L2-J2)^2);3)
D2	=$N$2*$N$4*(J2-L2)/СТЕПЕНЬ(КОРЕНЬ((K2-I2)^2+(L2-J2)^2);3)
E2	=$N$6
F2	=$N$7
G2	=$N$8
H2	=$N$9
I2	=$N$10
J2	=$N$11
K2	=$N$12
L2	=$N$13

В комірки E2-L2 ми переносимо значення із стовпчика «Дано», а в A2-D2 заносимо формули для обчислення проекцій прискорення тіл згідно (3).

3. У третій рядок в комірки A3-D3 скопіюємо вміст комірок A2-D2, а комірки E3-L3 модифікуємо відповідно до формул (4) та (5):

комірки	***формули / числа***
E3	=E2+A2*$N$3
F3	=F2+B2*$N$3
G3	=G2+C2*$N$3
H3	=H2+D2*$N$3
I3	=I2+E3*$N$3
J3	=J2+F3*$N$3
K3	=K2+G3*$N$3
L3	=L2+H3*$N$3

4. Скопіювати третій рядок у наступні рядки (їх кількість добиратимемо експериментально).

Далі ми можемо розпочати

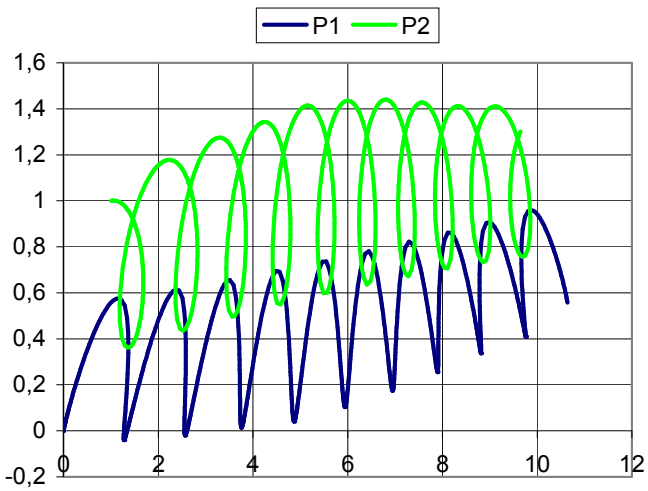
ОБЧИСЛЮВАЛЬНИЙ ЕКСПЕРИМЕНТ:

Введемо такі початкові дані: G=1; Δt=0,01; m_1=10; m_2=8; υ_{1x0}=0; υ_{1y0}=0; υ_{2x0}=2,2; υ_{2y0}=0,1; x_{10}=0; y_{10}=0; x_{20}=1; y_{20}=1. Легко побачити, що їх вибір нічим не зумовлений, проте для тестування моделі цього цілком достатньо. Зауважимо, що при виборі слід звернути увагу на час дискретизації – він не повинен бути занадто великим (при цьому алгоритм, що включає в себе різницеву схему, втратить стійкість – чим більше проміжок, тим менш достовірні результати), проте і занадто малим його робити не слід – нам буде заважати не лише величезна кількість рядків, які нам треба буде скопіювати, а й постійно зростаюча в операціях сумування похибка округлення.

D8 = =$N$2*$N$4*(J8-L8)/СТЕПЕНЬ(КОРЕНЬ((K8-I8)^2+(L8-J8)^2);3)

	D	E	F	G	H	I	J	K	L	M	
1	a_{2y}	υ_{1x}	υ_{1y}	υ_{2x}	υ_{2y}	x_1	y_1	x_2	y_2	*Дано:*	
2	-3,53553	0	0	2,2	0,1	0	0	1	1	G=	1
3	-3,42346	0,02828	0,02828	2,16464	0,06464	0,00028	0,00028	1,02165	1,000646	Δt=	0,01
4	-3,31908	0,05625	0,05567	2,12969	0,03041	0,00085	0,00084	1,04294	1,000951	m_1 =	10
5	-3,2217	0,08391	0,08222	2,09511	-0,0028	0,00168	0,00166	1,06389	1,000923	m_2 =	8
6	-3,13071	0,11131	0,108	2,06086	-0,035	0,0028	0,00274	1,0845	1,000573	υ_{1x0} =	0
7	-3,04558	0,13846	0,13304	2,02692	-0,0663	0,00418	0,00407	1,10477	0,99991	υ_{1y0} =	0
8	-2,9658	0,16539	0,15741	1,99326	-0,0968	0,00584	0,00565	1,1247	0,998942	υ_{2x0} =	2,2
9	-2,89095	0,19212	0,18113	1,95986	-0,1264	0,00776	0,00746	1,1443	0,997678	υ_{2y0} =	0,1
10	-2,82065	0,21866	0,20426	1,92667	-0,1553	0,00994	0,0095	1,16357	0,996125	x_{10} =	0
11	-2,75453	0,24505	0,22683	1,89369	-0,1835	0,01239	0,01177	1,18251	0,994289	y_{10} =	0
12	-2,6923	0,27129	0,24886	1,86089	-0,2111	0,01511	0,01426	1,20112	0,992178	x_{20} =	1
13	-2,63366	0,29741	0,2704	1,82824	-0,238	0,01808	0,01696	1,2194	0,989798	y_{20} =	1
14	-2,57836	0,32343	0,29147	1,79571	-0,2643	0,02132	0,01988	1,23736	0,987155		
15	-2,52616	0,34936	0,3121	1,7633	-0,2901	0,02481	0,023	1,25499	0,984254		
16	-2,47686	0,37522	0,33231	1,73097	-0,3154	0,02856	0,02632	1,2723	0,9811		
17	-2,43026	0,40103	0,35212	1,69871	-0,3402	0,03257	0,02984	1,28929	0,977698		

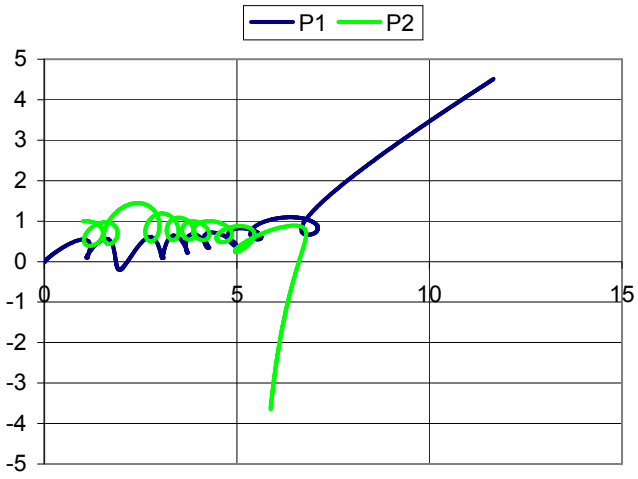
На малюнку – графік руху тіл, отриманий при таких початкових даних:



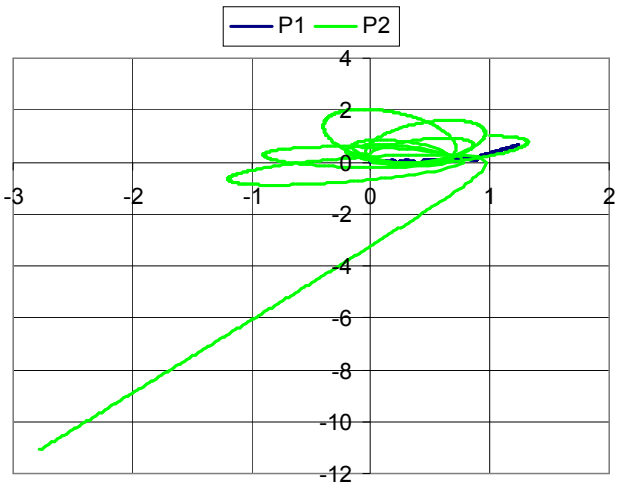
Проведемо

АНАЛІЗ РЕЗУЛЬТАТІВ ОБЧИСЛЮВАЛЬНОГО ЕКСПЕРИМЕНТУ:

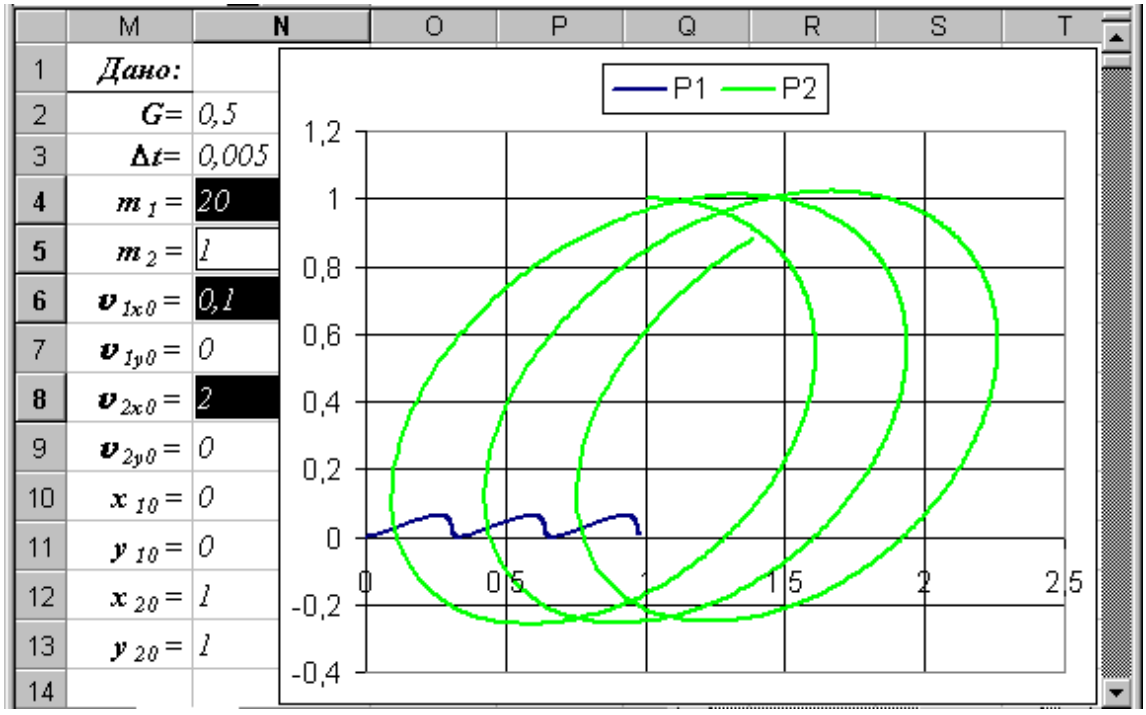
З малюнка видно, що тіла дійсно рухаються згідно закону всесвітнього тяжіння: хоча траєкторія їх спільного руху і звивиста, проте можна побачити, що у моменти зближення прискорення, відповідно до закону всесвітнього тяжіння, збільшується, що, у свою чергу, призводить до різкого збільшення швидкості. Тіла «розлітаються», але із збільшенням відстані прискорення зменшується до 0, аж поки не змінює свій знак. Цей факт означає, що тіла повинні знову зближуватися і т.д.



а)



б)



в)

Поставимо питання:

1. А що буде, якщо тіла занадто зблизяться? Якщо їх маси співрозмірні, то вони на великій швидкості віддалятимуться одне від одного у нескінченність (мал. а), якщо ж ми маємо справу з системою планета-супутник, то супутник передасть свій імпульс планеті і вирветься з її «гравітаційних обіймів» (мал. б).

2. Які початкові умови треба задати, щоб тіла рухались у одному напрямку? Для відповіді на це питання скористаємося законом збереження імпульсу: задамо початкові дані такими, щоб виконувалося, наприклад, співвідношення:

$$m_1 \upsilon_{1x0} = m_2 \upsilon_{2x0}.$$

Результати підтверджують нашу здогадку (мал. в).

Незважаючи на довільність вихідних даних, під час тестування ми впевнилися у *якісній відповідності* нашої моделі припущенням, покладеним у її основу (якими у нашому випадку виступають закони динаміки). Завдяки використанню зручного середовища для моделювання, у обчислювальному експерименті ми з легкістю змінювали параметри моделі, миттєво отримуючи результати. Нарешті, наявність зручного засобу візуалізації результатів моделювання дозволило нам провести аналіз експерименту і скорегувати початкові дані у відповідності до тих результатів, які ми хотіли отримати. І все це ми зробили *без написання програми*!

Тепер, коли ми «відчули» модель, можна приступити до програмної реалізації задачі багатьох тіл, а саме – створити програму для моделювання взаємного руху тіл, кількість, координати та швидкості яких задаються у файлі проекціями на координатні вісі: