Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Факультет комп’ютерних наук

**КУРСОВА РОБОТА**

з дисципліни «Теорія алгоритмів»

Тема «Задача пошуку виходу з лабіринту»

Виконала студентка 2 курсу

групи КС-21

Дібцева Анна Миколаївна

Перевірив:   
доц. Щебенюк В.С. ..............

доц. Олешко О.І. ..............

ст. викл. Лисицький К.Є. ..............

Харків – 2019

**ЗМІСТ**

[ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ 3](#_Toc27373472)

[ВСТУП 4](#_Toc27373473)

[РОЗДІЛ 1 ОГЛЯД ЗАДАЧІ 5](#_Toc27373474)

[РОЗДІЛ 2 КЛАСИФІКАЦІЯ ЛАБІРИНТІВ 6](#_Toc27373475)

[2.1 Розмірність 6](#_Toc27373476)

[2.2 Тесселяція 8](#_Toc27373477)

[2.3 Маршрутизація 11](#_Toc27373478)

[РОЗДІЛ 3 ПОШУК ВИХОДУ 13](#_Toc27373479)

[3.1 Дотримання уздовж стін (Wall follower) 13](#_Toc27373480)

[3.2 Алгоритм ланцюгів 14](#_Toc27373481)

[3.3 Recursive backtracker 15](#_Toc27373482)

[3.4 Алгоритм Тремо 15](#_Toc27373483)

[3.5. Заповнювач глухих кутів (Dead end filler) 16](#_Toc27373484)

[3.6. Random mouse 16](#_Toc27373485)

[3.7 Хвильовий алгоритм 16](#_Toc27373486)

[ВИСНОВКИ 20](#_Toc27373487)

[ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 22](#_Toc27373488)

[ДОДАТОК А ПОВНА ВЕРСІЯ РЕАЛІЗАЦІЇ 23](#_Toc27373489)

# ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

3D - 3-dimensional (з англ. тривимірний),

2D - 2-dimensional (з англ. двовимірний)

# ВСТУП

Алгоритми пошуку шляху – невід’ємна частина розробки ігор. А також різних систем навігації, орієнтації і т.д.

На теперішній час завдання дістатися з одної точки в іншу, оминаючи перешкоди, має широке розповсюдження і застосовується у великій кількості областей нашого життя. Наприклад, за потреби дістатися у якесь місце, ми використовуємо GPS навігатори для генерацій найкоротшого шляху. Або, краще за всіх пройти гру, використовуючи карту з різними шляхами.

Алгори́тм — набір інструкцій, які описують порядок дій виконавця, щоб досягти результату розв’язання задачі за скінченну кількість дій; система правил виконання дискретного процесу, яка досягає поставленої мети за скінченний час [1].

Теорія алгоритмів – це наука, що вивчає загальні властивості і закономірності алгоритмів і різноманітні формальні моделі їх подання. До завдань теорії алгоритмів відносяться формальний доказ алгоритмічної нерозв'язності завдань, асимптотичний аналіз складності алгоритмів, класифікація алгоритмів відповідно до класів складності, розробка критеріїв порівняльної оцінки якості алгоритмів і т.д. [2].

Алгоритми класифікуються відповідно до типу розв'язуваної задачі: сортування, пошуку, обробки графів і т.п.

Тож у цій роботі буде розглянуто так званий «алгоритм пошуку» для реалізації знаходження виходу з лабіринту, класифікація лабіринтів і алгоритми їх проходження.

# РОЗДІЛ 1 ОГЛЯД ЗАДАЧІ

У цій роботі проводяться дослідження для визначення оптимального рішення задачі пошуку виходу з лабіринту. Розглядаються різні типи геометрії та маршрутизації лабіринту для можливості твори подальших маніпуляцій над об'єктом вивчення.

Виходячи з теми, об'єкт нашого дослідження це пошук шляху, який зв'язує 2 точки певного лабіринту.

Предметом дослідження даної курсової роботи є знаходження і аналіз шляху від входу до виходу із заданого лабіринту.

Виходячи з вищевикладеного, метою дослідження методу пошуку виходу з лабіринту є визначення оптимального алгоритму для знаходження шляху, який зв'язує 2 точки всередині конкретного лабіринту (в нашому випадку, між входом і виходом).,

Тож результаті ми маємо отримати аналіз різних різних типів лабіринтів і алгоритмів, котрі можна на них використовувати.

# РОЗДІЛ 2 КЛАСИФІКАЦІЯ ЛАБІРИНТІВ

Лабіринти в цілому (а значить, і алгоритми для їх створення) можна розбити по семи різних класифікацій: розмірності, тесселяции, маршрутизації [4].

### 2.1 Розмірність

Розмірність по суті визначає, скільки вимірювань в просторі заповнює лабіринт. Існують наступні типи:

Двомірні: більшість лабіринтів мають цю розмірність, так як ми завжди можемо відобразити схему на аркуші паперу і рухатися по ньому, не перетинаючи ніяких інших коридорів. Для наочності в цій роботі будуть розглядатися лабіринти саме цієї розмірності.



Рисунок 2.1 – Двовимірний лабіринт [4]

Тривимірні: на відміну від двомірної моделі даний лабіринт має кілька рівнів; в ньому проходи можуть крім чотирьох сторін світу опускатися вниз і підніматися вгору. 3-dimensional (3D) -лабіринт часто візуалізують як масив з 2-dimensional (2D) -рівнем з позначками сходів «вгору» і «вниз».

Також існують чотиривимірні лабіринти і ще більш багатовимірні. Іноді їх візуалізують як 3D-лабіринти з «порталами», провідними крізь четвертий вимір, наприклад, портали в «минуле» і «майбутнє» [4].

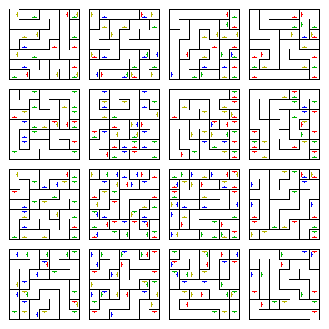


Рисунок 2.2 – Багатовимірний лабіринт [4]

Переплетення: лабіринти з переплетеннями - це, по суті ті ж самі двомірні (або, точніше 2,5-мірні) лабіринти, в яких, на відміну від звичайної двомірної схеми, проходи можуть накладатися один на одного. При відображенні зазвичай цілком очевидно, де знаходяться тупики та як один прохід знаходиться над іншим, тому що вони малюються спеціальним чином: верхні як би «накривають» нижні проходи і виглядають цілісно, в той час як нижні проходи показані в цій точці як розриви або тупики, хоча прохід там все ж існує. У реальному світі лабіринти, в яких є мости, що з'єднують одну частину лабіринту з іншого, частково є переплетеннями.

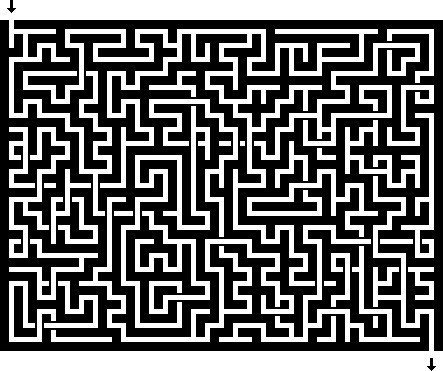


Рисунок 2.3 – Переплетений лабіринт [4]

## 2.2 Тесселяція

Тесселяція - це класифікація геометрії окремих осередків, з яких складається лабіринт [4]. Існуючі типи:

Ортогональний, на базі якого здійснюються дослідження даної курсової роботи. Це стандартна прямокутна сітка, в якій осередки мають проходи, що перетинаються під прямими кутами (рисунок 3.1).

Також існують дельта-лабіринти складаються із сполучених трикутників, і у кожного осередку може бути до трьох з'єднаних з нею проходів [4].



Рисунок 2.4 – Дельта-лабіринт [4]

Не менш цікаві і складні сигма-лабіринти, які складаються із сполучених шестикутників, при цьому у кожній клітинці може бути до 6 проходів.

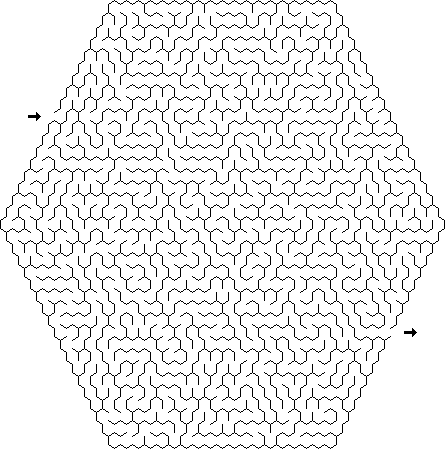


Рисунок 2.5 – Сигма-лабіринт [4]

Круглі тета-лабіринти складаються з концентричних кіл проходів, в яких початок або кінець знаходиться в центрі, а інший - на зовнішньому краї. Осередки зазвичай мають чотири можливих з'єднання проходів, але їх може бути і більше завдяки більшій кількості осередків в зовнішніх кільцях проходів [4].

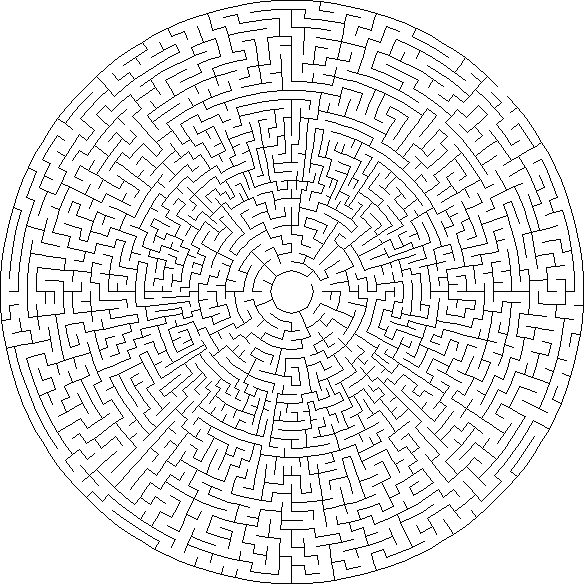


Рисунок 2.6 – Тета-лабіринт [4]

І іпсилон-лабіринт, який представляє собою з'єднані з сусідами восьмиугольники або квадрати.

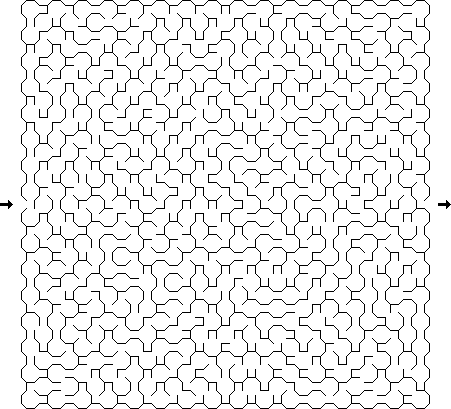


Рисунок 2.7 – Іпсилон-лабіринт [4]

Crack-лабіринт - це аморфний лабіринт без постійної тесселяции, в якому стіни і проходи розташовані під випадковими кутами.

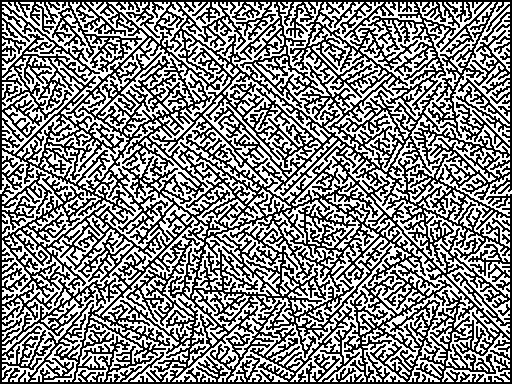


Рисунок 2.8 – Crack-лабІринт [4]

Фрактальний лабіринт - це лабіринт, що складається з безлічі менших лабіринтів. Є також фрактальний лабіринт з вкладених осередків - це лабіринт, в кожному осередку якого розміщені інші лабіринти, і цей процес може повторюватися кілька разів. І нескінченно рекурсивний фрактальний лабіринт - це справжній фрактал, в якому вміст лабіринту копіює себе, створюючи по суті нескінченно великий лабіринт.

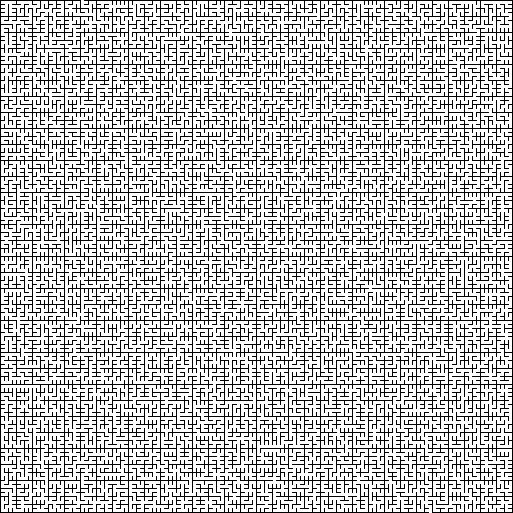


Рисунок 2.9 – Фрактальный лабіринт [4]

## 2.3 Маршрутизація

Класифікація за маршрутами є важливим аспектом в генерації лабіринтів. Він пов'язаний з типами проходів в межах геометрії (див. 3.1.2)

Ідеальним (рисунок 3.1) або лабіринтом з одиночним з'єднанням (simply-connected Maze) називають лабіринт без петель, замкнутих ланцюгів і недосяжних областей. У ньому з кожної точки існує єдиний шлях до будь-якої іншої точки, тобто має тільки одне рішення. З точки зору програмування лабіринт такого виду можна описати як дерево, сполучна безліч осередків або вершин. Саме лабіринт ідеальної маршрутизації буде використовуватися для визначення оптимального вирішення проблеми пошуку вихід з лабіринту.

Плетеним або лабіринтом з багаторазовими сполуками (purely multiply connected Maze) називають такий лабіринт, де не існує тупиків. В даному лабіринті використовують замикаються і повертаються один до одного проходи. Якісний плетений лабіринт набагато складніше ідеального лабіринту того ж розміру, тому що витрачається більше часу на ходьбу колами замість попадання в тупики і повернення до попередньої розвилці.

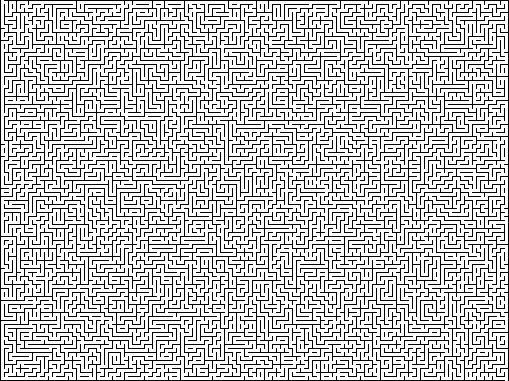


Рисунок 2.10 – Плетеный лабіринт [4]

У одномаршрутні лабіринті розвилки взагалі відсутні. Він містить єдиний довгий, але в’юнкий прохід, який змінює напрямок на всьому протязі лабіринту. Складність полягає лише в тому, щоб не повернути випадково в зворотному напрямку і не повернутися до входу. Однак, в програмному коді ця проблема легко усувається запам'ятовуванням попередньої пройденої клітини або ж, у разі послідовного перевірки оточуючих осередків на наявність проходу, при відсутності тупиків, ця проблема взагалі не виникає.



Рисунок 2.11 – Одномаршрутній лабіринт [4]

# РОЗДІЛ 3 ПОШУК ВИХОДУ

Існує безліч способів вирішення лабіринтів, і кожен з них має власні характеристики. У цьому розділі розглядаються конкретні алгоритми пошуку виходу з лабіринту, їх переваги та недоліки.

## 3.1 Дотримання уздовж стін (Wall follower)

Дотримання уздовж стін (Wall follower) або правило лівої чи правої руки - це простий алгоритм вирішення лабіринтів. Пріоритетом для нього є проходить лабіринт об'єкт («ви»), він завжди дуже швидкий і не використовує додаткової пам'яті, тому що не використовує стек і не зберігає будь-які дані про проходження, а лише змінює значення елементів матриці в залежності від виконання умов. До обробки ці елементи можуть приймати одне з наступних ДВОХ значень:

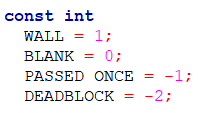


Рисунок 3.1 – Возможные значения ячеек

Починаємо йти по проходах і при досягненні розвилки завжди повертаємо направо (або завжди наліво). Щоб застосувати таке рішення лабіринту в реальному світі, потрібно покласти руку на праву (або ліву) стіну і постійно тримати її на стіні в процесі проходження лабіринту. При бажанні можна позначати вже відвідані комірки та комірки, відвідані двічі. В кінці можна повернутися назад за рішенням, слідуючи тільки по осередках, відвіданих один раз.

Дотримання уздовж стіни в 3D-лабіринті можна реалізувати детермінованим способом, спроектувавши 3D-проходи на 2D-площину, тобто прикинувшись, що провідні вгору проходи насправді ведуть на північний захід, а провідні вниз ведуть на південний схід, а потім застосувати звичайні правила проходження вздовж стін.

Цей метод не обов'язково знайде найкоротший рішення. Але його можна застосовувати навіть на лабіринтах з багаторазовими сполуками, де відсутні тупики, тому що проходячи вдруге по осередках, алгоритм позначає їх як тупикові і не буде «ходити» колами на одному місці.

## 3.2 Алгоритм ланцюгів

Алгоритм ланцюгів (Chain algorithm) вирішує лабіринт, сприймаючи його як безліч лабіринтів меншого розміру (подібно ланкам ланцюга) і вирішуючи їх по порядку.

У параметри передаються місця початку і кінця, і алгоритм завжди знайде шлях від між цими точками, якщо він існує. При цьому рішення схильне бути розумно коротким, якщо навіть не найкоротшим. Це означає, що таким способом можна вирішувати лабіринти, в яких невідомо точне розташування кінця.

По суті це алгоритм проходження вздовж стін зі способом перестрибування між островами. Починаємо з малювання прямої лінії (або хоча б лінії без самоперетинів) від початку до кінця, дозволяючи їй при необхідності перетинати стіни. Потім просто слідуємо по лінії від початку до кінця. Якщо ми натикаємося на стіну, то не можемо пройти через неї, а значить повинні обходити. Відправляємо двох наступних уздовж стін «роботів» по кожному з напрямків уздовж стіни, на яку натрапили. Якщо робот знову перетнеться з указующей лінією в тій точці, де вона ближче до виходу, тоді зупиняємось і слідуємо за цією стіні, поки не доберемося до неї самі. Продовжуємо слідувати по лінії і повторювати процес, поки не досягнемо кінця. Якщо обидва робота повернуться до їх вихідним локаціях і напрямками, то подальші точки вздовж прямої недосяжні і лабіринт нерозв'язних.

## 3.3 Recursive backtracker

Алгоритм recursive backtracker знаходить рішення, але воно не обов'язково буде найкоротшим. Прірітетом для нього є проходить лабіринт об'єкт, він швидкий для всіх типів лабіринтів і використовує стек розміром аж до розміру лабіринту.

Він дуже простий в реалізації: якщо ми знаходимося біля стіни (або в поміченої лінією області), то повертаємо «невдача», інакше, якщо ми знаходимося в кінці, повертаємо «успіх», інакше, рекурсивно пробуємо рухатися у всіх чотирьох напрямках. Малюємо лінію, коли намагаємося пройти в новому напрямку і видаляємо її, якщо повернуто значення «невдача»; після досягнення стану «успіх» у нас буде нанесено лініями єдине рішення. При поверненні назад (backtracking) краще позначати простір особливим значенням відвіданих місць, щоб ми не відвідували їх знову, приходячи з іншого напрямку. Цей метод завжди знаходить рішення, якщо воно існує, але воно не обов'язково буде найкоротшим.

## 3.4 Алгоритм Тремо

Цей метод вирішення лабіринтів розроблений для використання людиною всередині лабіринту. Він схожий на recursive backtracker (див. 3.3) і знаходить рішення для всіх лабіринтів: при русі по проходу ми малюємо лінію за собою, позначати наш шлях. При попаданні в глухий кут повертаємося назад і повертаємося тим же шляхом, яким прийшли, позначаючи його вдруге, що буде означати, що він не має розгалужень в подальшому. Дійшовши до розвилки, на якій ще не були, випадковим чином вибираємо новий прохід. Якщо ми проходимо по новому проходу і доходимо до розвилки, яку відвідували раніше, то вважаємо її тупиком і повертаємося тим же шляхом, яким прийшли. Варто звернути увагу на цей етап, тому що саме завдяки йому користувач не ходить колами. Всі проходи будуть або порожніми, тобто ми їх ще не відвідували, поміченими один раз, тобто ми там були рівно один раз, або поміченими двічі, тобто ми рухалися по ним і змушені були повертатися в зворотному напрямку. Коли ми нарешті досягнемо рішення, то помічені один раз шляху становитимуть прямий шлях до самого початку. Якщо у лабіринту немає рішення, то ми опинимося на початку, а всі проходи будуть позначені двічі.

## 3.5. Заповнювач глухих кутів (Dead end filler)

Це простий алгоритм вирішення лабіринтів, що не вимагає додаткової пам'яті. Ми просто досліджуємо лабіринт і заповнюємо кожен глухий кут, заливаючи прохід в зворотному порядку від безвиході, поки не досягнемо розвилки. В кінці у нас залишиться одне рішення, або кілька рішень, якщо у лабіринту їх більше одного.

Алгоритм завжди знаходить для ідеальних лабіринтів одне унікальне рішення. Однак є неефективним лабіринтах з сильним плетінням, і даремний у всіх лабіринтах без тупиків.

## 3.6. Random mouse

Це неефективний метод вирішення лабіринту, що полягає у випадковому переміщенні. Чи не робляться повороти на 180 градусів, якщо без них можна обійтися. Алгоритм повільний і не гарантує завершення або рішення лабіринту, а після досягнення кінця буде настільки ж складно повернутися до початку, зате він простий і не вимагає додаткової пам'яті для реалізації.

## 3.7 Хвильовий алгоритм

Дії хвильового алгоритму чимось нагадують розходяться кола на воді. Навколо стартової точки, якщо вона розташована не на краю матриці, завжди знаходяться ще чотири точки. Нам необхідно перевірити, чи вільні вони для проходження? Інструкція комп'ютера буде звучати так: «пройди прямо і перевір, чи вільна клітина попереду? Якщо так, то перейди туди і зміни її одиницею. Якщо немає, то пройди назад і перевір, чи вільна клітина позаду? Якщо так - перейди туди і познач її одиницею. Якщо немає, перевір, чи вільна клітина справа? Якщо так, то перейди туди і познач її одиницею. Якщо немає, перевір клітку зліва ».

Якщо навколо точки або, в даному випадку, старту нашої подорожі є вільні клітини, вони будуть позначені «одиницею». В наступному кроці ми повинні будемо помітити «двійкою» все вільні клітини, навколо клітин з «одиницею», зрозуміло, окрім вже помічених. Клітини з «двійкою» знайдуть власне завдання - помітять вільний простір «трійкою» і т.д. Таким чином, хвиля «розбігається» від центру, огинаючи перешкоди і досягає фінішу - виходу з лабіринту.

Розглянемо реалізацію рішення проблеми пошуку виходу з лабіринту на прикладі хвильового алгоритму, застосованого на двомірному ортогональному лабіринті.

На вхід програма отримує текстовий файл, з якого завантажує матрицю лабіринту, заповнену умовними цифровими позначеннями:

* 0 – прохід;
* 1 – стіна;
* 2 – стартова точка.

Рухатися можна тільки вправо, вліво, вгору і вниз (можна ходити ще й по діагоналі але це не буде).

Починаючи зі стартовою точки починається дослідження шляхів лабіринту, поступово розширюючи радіус дії. Це організовується за допомогою циклу, на кожній ітерації якого, знаходяться осередки з поточним найвищим значенням, для яких викликається функція checkNeighbors (x, y) (рисунок 3.2) для перевірки сусідніх клітин на наявність продовження шляху далі (рисунок 3.1).

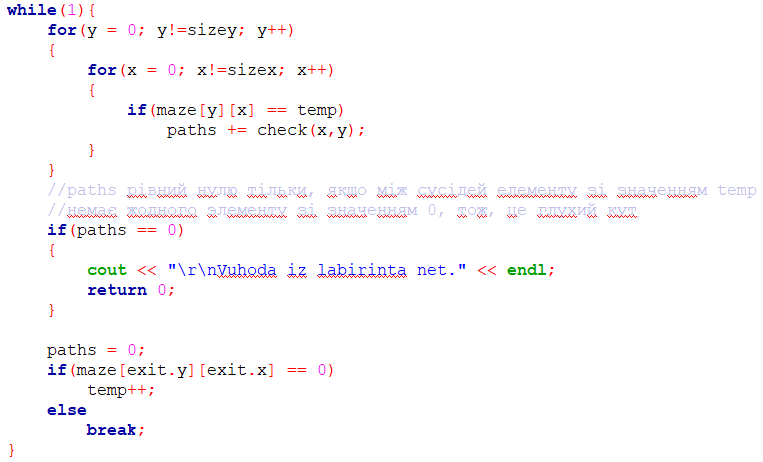


Рисунок 3.2 – Цикл пошуку та перевірки стану шляху

Функція checkNeighbors (x, y) (рисунок 3.2) приймає на вхід координати x і y (номер стовпчика і рядка) осередки лабіринту. Проводиться перевірка сусідніх клітин на наявність проходу. Якщо прохід знайдений, то значення змінюється з 0 на поточне значення temp змінної, що позначає рівень дослідженості лабіринту.

Функція возвращяет кількість cnt проходів, для з подальшої обробки:

* Якщо *cnt* = 0, товихіду з лабіринту не існує;
* Якщо *cnt* > 0, то цикл продовжує свою роботу, збильшивши рівень проходження лабіринту *temp*.

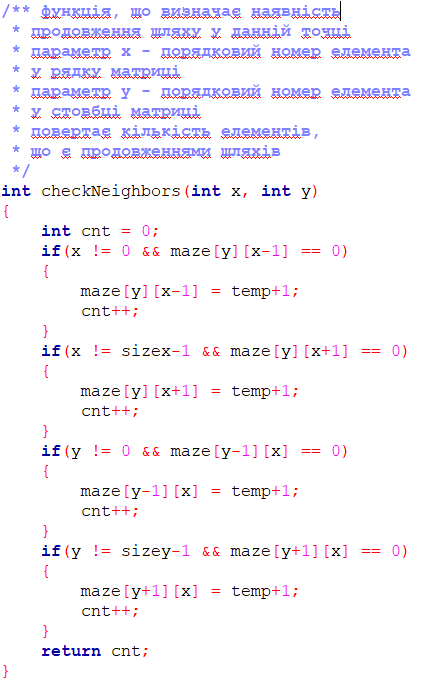


Рисунок 3.3 – Функція *checkNeighbors(x, y)*

А найкоротший шлях знаходиться дуже просто. Потрібно на кожному кроці, почавши відлік від фінішу, шукати клітини, помічені все меншим і меншим числом. Від «дев'ятки» шукаємо «вісімку», від неї - «сімку» і т.д. У 95% випадків таким чином знаходиться найкоротший шлях. Але, на жаль, не завжди. Якщо набір перешкод має складну форму, то «хвиля» може «зациклитися». Тому, програмі потрібно встановити якусь межу для кількості ітерацій [5].

Тепер маємо повноцінну реалізацію Хвильового алгоритму поиска Вихід з лабірінту. Повний код реализации наведено у Додатках А.

# ВИСНОВКИ

На сьогодні, в нашому сучасному житті велику частину займають технології. Серед яких популярні онлайн і офлайн ігри, навігатори, розвиваються системи проектування доріг, залізничних колій і розташування систем водоканалу і т.п.

Дуже велику важність у цих областях займає саме система програмування шляхів з додаванням особливих факторів в залежності від області використання.  
 Але якщо їх узагальнити, то ми отримаємо ті ж самі лабіринти (2D, 3D але це все є саме лабіринти) з шляхами та стінами.

В цій курсовій роботі були розглянуті різні типи лабіринтів, їх геометрія та маршрутизація.

Були також наведені різні алгоритми пошуку виходу з лабіринту, з зауваженнями щодо їх складності, індивідуальними рисами, затратним або навпаки, з економним використовуванням пам’яті комп’ютера, і ефективності виконування поставленої задачі.

Більш детально ознайомились з хвильовим алгоритмом, використаним на ідеальному ортогональному лабіринті.

В результаті програми ми отримали проаналізований лабіринт, з визначеним найкоротшим шляхом від стартової точки до виходу з лабіринту.

Виходячи с процесу написання коду програми, труднощів, котрі з’являлися у процесі та результату можна зробити висновок, що хвильовий алгоритм є простим і надіним методом вирішення проблеми визначення найкоротшого шляху між двома точками лабіринту. Але він має й суттєвий недолік – він вимагає великого об’єму пам’яті і не найкращу швидкість роботи. Вона, авжеж, прийнята для покрокових іграшок, але не потягне швидкі динамічні ігри.

Корисною альтернативою могло б слугувати алгоритм заповнювання глухих кутів (Dead end filler). На відміну від хвильового алгоритму, він має велику швидкість обробки даних і не потребує додаткової пам’яті.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Алгоритм – Вікіпедія. // https://uk.wikipedia.org/wiki/Алгоритм. Дата звернення: 15.12.2019.
2. Теория алгоритмов – Википедия // https://ru.wikipedia.org/wiki/ Теория\_алгоритмов. Дата звернення: 15.12.2019.
3. Лабиринт – Википедия // https://ru.wikipedia.org/wiki/Лабиринт Дата звернення: 15.12.2019.
4. Лабиринты: классификация, генерирование, поиск // https://habr.com/ru/ post/445378/ Дата звернення: 02.12.2019.
5. Программируем выход из лабиринта // https://www.kv.by/content/ 338320-programmiruem-vykhod-iz-labirinta Дата звернення: 06.12.2019.

# ДОДАТОК А ПОВНА ВЕРСІЯ РЕАЛІЗАЦІЇ

Лістинг 1 – реалізація хвильового пошуку виходу з лабіринту

#include <iostream>

#include <clocale>

#include <stdio.h>

#include <io.h>

#define MEN 2

#define LF 10

#define SPACE 32

#define ORIGINAL\_MAZE 0

#define MAZE\_WITH\_PATH 1

using namespace std;

struct Point

{

int x;

int y;

};

Point exit;

int\*\* maze;

int sizex = 0, sizey = 0;

int load\_maze(){

int i, j, filesize = 0, x = 0, y = 0;

char\* cc;

FILE \*F;

F = fopen("maze.txt","rt");

if( F==NULL )

return 0;

filesize = filelength(fileno(F));

cc = new char[filesize];

for(i=0;i!=filesize;i++)

cc[i] = getc(F);

fclose(F);

i = 0;

while(cc[i] != LF)

{

if(cc[i] != SPACE)

sizex++;

i++;

}

sizey = filesize/(sizex\*2);

maze = new int\*[sizey];

while(y != sizey)

maze[y++] = new int[sizex];

i=0;

for(y = 0;y!=sizey;y++)

{

for(j=0;j!=sizex;j++)

{

maze[y][x] = cc[i]-48;

x++;

i+=2;

}

x=0;

}

return 1;

}

int temp = MEN;

int checkNeighbors(int x, int y){

int cnt = 0;

if(x != 0 && maze[y][x-1] == 0)

{

maze[y][x-1] = temp+1;

cnt++;

}

if(x != sizex-1 && maze[y][x+1] == 0)

{

maze[y][x+1] = temp+1;

cnt++;

}

if(y != 0 && maze[y-1][x] == 0)

{

maze[y-1][x] = temp+1;

cnt++;

}

if(y != sizey-1 && maze[y+1][x] == 0)

{

maze[y+1][x] = temp+1;

cnt++;

}

return cnt;

}

int search\_exit(){

int x, y;

exit.x = 0;

exit.y = 0;

y = 0;

for(x = 0; x!=sizex; x++)

if(maze[y][x] == 0)

{

exit.x = x;

exit.y = y;

return 1;

}

y = sizey-1;

for(x = 0; x!=sizex; x++)

if(maze[y][x] == 0)

{

exit.x = x;

exit.y = y;

return 1;

}

x = 0;

for(y = 0; y!=sizey; y++)

if(maze[y][x] == 0)

{

exit.x = x;

exit.y = y;

return 1;

}

x = sizex-1;

for(y = 0; y!=sizey; y++)

if(maze[y][x] == 0)

{

exit.x = x;

exit.y = y;

return 1;

}

return 0;

}

void print\_maze(char mode){

int x,y;

for(y = 0; y!=sizey; y++)

{

for(x = 0; x!=sizex; x++)

{

if(maze[y][x] == MEN)

cout << "M" << "";

else

if(mode == 1 && maze[y][x] == 0)

cout << "\*" << "";

else

if(maze[y][x] > MEN)

cout << "0" << "";

else

cout << maze[y][x] << "";

}

cout << endl;

}

}

int main(){

setlocale(LC\_CTYPE, "rus");

int x, y, paths = 0, i = 1, steps, cnt = 0;

if(load\_maze() == 0)

{

cout << "File maze.txt not found." << endl;

return 0;

}

cout << "Labirint:\r\n" << "(" << sizex << "x" << sizey << ")\r\n" << endl;

print\_maze(ORIGINAL\_MAZE);

cout << "\r\nM - Chelovek\r\n1 - Stena\r\n0 - Prohod" << endl;

if(search\_exit() == 0)

{

cout << "\r\nVuhoda iz labirinta net." << endl;

return 0;

}

cout << "\r\nKoordinatu Vuhoda:\r\nX = " << exit.x << "; Y = " << exit.y << "\r\n" << endl;

while(1){

for(y = 0; y!=sizey; y++)

{

for(x = 0; x!=sizex; x++)

{

if(maze[y][x] == temp)

paths += check(x,y);

}

}

//paths рівний нулю тільки, якщо між сусідей елементу зі значенням temp

//немає жодного элементу зі значенням 0, тож, це глухий кут

if(paths == 0)

{

cout << "\r\nVuhoda iz labirinta net." << endl;

return 0;

}

paths = 0;

if(maze[exit.y][exit.x] == 0)

temp++;

else

break;

}

steps = maze[exit.y][exit.x];

maze[exit.y][exit.x] = 0;

x = exit.x;

y = exit.y;

while(1)

{

if(x != 0 && maze[y][x-1] == steps-i){

if(maze[y][x-1] == 2)

break;

maze[y][x-1] = 0;

x--;

}

if(x != sizex-1 && maze[y][x+1] == steps-i){

if(maze[y][x+1] == 2)

break;

maze[y][x+1] = 0;

x++;

}

if(y != 0 && maze[y-1][x] == steps-i){

if(maze[y-1][x] == 2)

break;

maze[y-1][x] = 0;

y--;

}

if(y != sizey-1 && maze[y+1][x] == steps-i){

if(maze[y+1][x] == 2)

break;

maze[y+1][x] = 0;

y++;

}

i++;

}

cout << "\r\nSimvol '\*' - proydennuy element:\r\n" << endl;

print\_maze(MAZE\_WITH\_PATH);

cout << "\r\nLabirint proyden za " << steps - 2 << " hodov " << endl;

return 1;

}