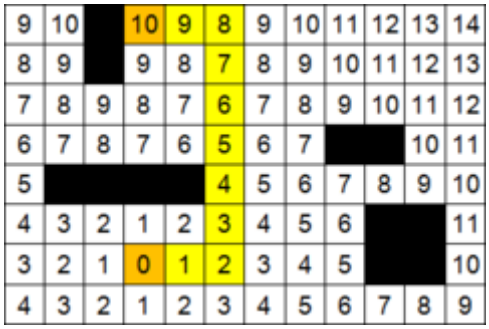
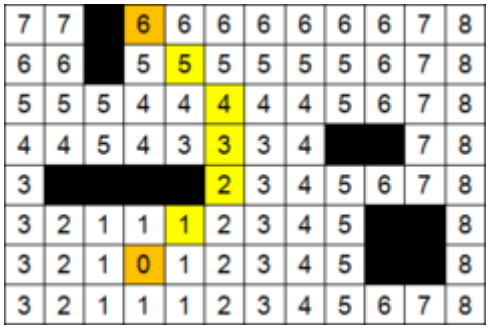
Хвильовий алгоритм

Матеріал з Вікіпедії — вільної енциклопедії.

Хвильовий алгоритм (Алгоритм Лі) МОРОЗ Л*Х— алгоритм, що дозволяє знайти мінімальний шлях в графі з ребрами одиничної довжини. Заснований на алгоритмі пошуку в ширину. Застосовується для знаходження найкоротшого шляху в графі, в загальному випадку знаходить лише його довжину.



Результат роботи алгоритму (ортогональний шлях)



Результат роботи алгоритму (ортогонально-діагональний шлях)

Зміст

Суть алгоритму
Різновиди
Практичне застосування
Опис алгоритму
Приклад реалізації
Див. також
Література
Посилання

Суть алгоритму

На двовимірній карті (матриці), що складається з «прохідних» і «непрохідних» комірок, позначена комірка старту і комірка фінішу. Мета алгоритму - прокласти найкоротший шлях від комірки старту до комірки фінішу, якщо це, звичайно, можливо. Від старту у всі напрями поширюється хвиля, причому кожна пройдена хвилею комірка позначається як «пройдена». Хвиля, у свою чергу, не може проходити через комірки помічені як «пройдені» або «непрохідні». Хвиля рухається, поки не досягне точки фінішу або поки не залишиться непройдених комірок. Якщо хвиля пройшла всі доступні комірки, але так і не досягла точки фінішу, значить, шлях від старту до фінішу прокласти неможливо. Після досягнення хвилею фінішу, прокладається шлях в зворотному напрямі (від фінішу до старту) і зберігається в масиві.

Різновиди

Заповнення по матриці в 4 напрямках (околиця фон Неймана), він же алгоритм Лі — точніший, але менш швидкий метод.

```
    i+1
i+1  i  i+1
    i+1
```

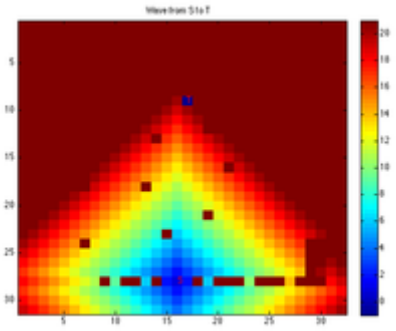
Заповнення по матриці в 8 напрямків (околиця Мура) — більш швидкий, але менш точний метод. Шлях по діагоналі приблизно в 1.4 рази довший, ніж по горизонталі й вертикалі, саме тому комірки, розташовані по діагоналі, мають коефіцієнт +3, а по горизонталі й вертикалі — коефіцієнт +2.

```
i+3 i+2 i+3
i+2  i  i+2
i+3 i+2 i+3
```

Існує також алгоритм A* (A-star) — направлений хвильовий алгоритм.

Практичне застосування

Хвильовий алгоритм - один з основних при автоматизованому трасуванні (розводці) друкованих плат. Він має досить велику кількість різноманітних модифікацій, що дозволяють покращити знайдений розв'язок з точки зору затрат пам'яті та швидкодії. Також одне з характерних застосувань хвильового алгоритму — пошук найкоротшої відстані на карті в стратегічних комп'ютерних іграх.



Опис алгоритму

Алгоритм складається з кількох етапів, серед яких основними можна виділити наступні: підготовчий етап, етап поширення хвилі та етап побудови шляху.

На підготовчому етапі проводиться аналіз комірок, що присутні на карті, визначаються зайняті комірки. Решта комірок позначаються як вільні, їм присвоюється вага "0". В лічильник кроків K записується 1.

Етап поширення хвилі полягає у знаходженні точки фінішу шляхом перебору сусідніх комірок та присвоєння їм відповідної ваги. В першу чергу перевіряються всі комірки, суміжні з початковою. Якщо комірка має вагу "0", їй присвоюється значення з лічильника кроків. Комірки з іншою вагою (заповнені на попередніх кроках), а також комірки, відмічені як зайняті, залишаються без змін. Якщо одна з комірок є точкою фінішу, алгоритм переходить на наступний етап – побудову шляху. В іншому випадку лічильник кроків збільшується на одиницю і описаний для початкової точки алгоритм повторюється для всіх точок з вагою K-1. Якщо кінцева точка не знайдена, і для всіх точок з вагою K-1 в околі не лишилось вільних комірок, то вважається, що шлях не існує.

На етапі побудови шляху (згортання хвилі) починаємо рух з кінцевої точки. Для цієї точки обирається комірка з її околу (вага цієї комірки буде K). Тоді для цієї комірки знову шукається суміжна з вагою K-1. Таким чином продовжується доти, поки не знайдено комірку з вагою 1, в околі якої знаходиться початкова точка (вага початкової точки 0). Таким чином маємо побудований шлях, який також відносить до множини шляхів мінімальної довжини.

Приклад реалізації

Також однією з задач, з якими легко справляється хвильовий алгоритм є пошук виходу з лабіринту. Нижче наведено приклад реалізації алгоритму пошуку виходу з лабіринту на мові C++.

```
#include "conio.h"      // Для функції getch()
#include <string.h>

struct screen_point{
 unsigned char chr;
 unsigned char attr;    // Це все що потрібно для виводу
};                      // на екран.
typedef struct screen_point screen_line[80];
screen_line * scr;
char movecost[10][10]={
   {0,0,0,0,0,0,0,0,0,0},
   {0,1,6,6,6,6,6,1,1,0},
   {0,1,0,0,0,0,6,0,0,0},
```

```
   {0,1,0,1,1,1,1,1,1,0},
   {0,1,0,1,1,0,0,0,1,0}, // Це і є лабіринт
   {0,1,0,1,0,0,1,0,1,0}, // 0 - стіна
   {0,1,0,1,0,1,1,0,1,0}, // будь-яке інше число -
   {0,1,0,0,0,0,0,0,1,0}, // ступінь прохідності
   {0,1,8,1,1,1,1,1,1,0}, // 1 - найкраща прохідність
   {0,0,0,0,0,0,0,0,0,0}
                        };
unsigned char fillmap[10][10];  // Розмір рівний розміру лабіринту!!!
                                // якщо шлях може бути довшим за
                                // 255 варто замінити byte->word
struct{
 signed char x,y;               // Координати в лабіринті
}buf[256];                      // Чим більший лабіринт, тим більшим повинен
                                // бути цей масив
unsigned char bufp,bufe;        // Індекси в buf

int sx,sy,tx,ty;                // Початкові та кінцеві координати шляху

/*
  Ця частина займається виводом на екран і
                не має жодного відношення до алгоритму
*/
void clrscr(){                  // Очистити екран
 int i;
 for(i=0;i<80*25;i++)((short*)scr)[i]=0x0720;
}

// Надрукувати рядок str в координатах (x,y) кольором attr
void writestr(int x,int y,char str[],char attr){
int i;
for(i=0;str[i]!=0;i++,x++){scr[y][x].chr=str[i];scr[y][x].attr=attr;}
}

// Малюємо початковий малюнок лабіринту
void draw_maze(){
 int i,j;
 for(j=0;j<10;j++)for(i=0;i<10;i++){
  scr[j][i*2  ].attr=16*(7-movecost[j][i])+7+8*((i+j)&1);
  scr[j][i*2+1].attr=16*(7-movecost[j][i])+7+8*((i+j)&1);
 }
 scr[sy][sx*2].chr='[';scr[sy][sx*2+1].chr=']';
 scr[ty][tx*2].chr='<';scr[ty][tx*2+1].chr='>';
 scr[1][40].attr=16*(7-1);writestr(45,1,"Пусте місце",7);
 scr[3][40].attr=16*(7-0);writestr(45,3,"Стіна",7);
 scr[5][40].attr=16*(7-6);writestr(45,5,"Болото",7);
 writestr(40,7,"[]    Початкова точка",7);
 writestr(40,9,"<>    Ціль шляху",7);
}

/*
  Реалізація самого алгоритму
*/

/* Ця функція перевіряє чи є запропонований шлях в точку фінішу більш коротким,
   ніж знайдені раніше, і якщо так, то запам'ятовує точку в buf.      */
void push(int x,int y,int n){
 if(fillmap[y][x]<=n)return;            // Якщо новий шлях не коротший, його відкидаємо
 fillmap[y][x]=n;                       // Запам'ятовуємо нову довжину шляху
 buf[bufe].x=x;
 buf[bufe].y=y;                         // Запам'ятовуємо точку
 bufe++;
 scr[y][x*2  ].chr=n/10+48;
                                        // Промальовка та очікування натиснення клавіші
 scr[y][x*2+1].chr=(n%10)+48;
 getch();   //
}
/* Тут береться чергова точка з buf і повертається 1,
  якщо buf пустий, повертається 0           */
int pop(int *x,int *y){
 if(bufp==bufe)return 0;
 *x=buf[bufp].x;
 *y=buf[bufp].y;
 bufp++;
 return 1;
}

void fill(int sx,int sy,int tx,int ty){
 int x,y,n,t;
// Спочатку fillmap заповнюється max значенням
 _fmemset(fillmap,0xFF,sizeof(fillmap));
 bufp=bufe=0;          // Обнуляємо буфери
 push(sx,sy,0);
```

```
  while(pop(&x,&y)){    // Цикл виконується, доки є точки в буфері
   if((x==tx)&&(y==ty)){
    writestr(0,20,"Знайдено шлях довжиною     ",15);
    scr[20][19].chr=n/10+48;
    scr[20][20].chr=(n%10)+48;
    break;
   }
 // n рівне довжині шляху до будь-якої сусідньої комірки
   n=fillmap[y][x]+movecost[y][x];
 // Перебір 4-х сусідніх комірок
   if(movecost[y+1][x  ])push(x  ,y+1,n);
   if(movecost[y-1][x  ])push(x  ,y-1,n);
   if(movecost[y  ][x+1])push(x+1,y  ,n);
   if(movecost[y  ][x-1])push(x-1,y  ,n);
  }

  // Якщо перебрано всю карту і не знайдено жодного шляху
  if(fillmap[ty][tx]==0xFF){
   writestr(0,20,"Шляху не існує!!!",15);
   return;
  }  else
  writestr(0,20,"Заливку завершено. Пройдемо по шляху!!!",15);

  x=tx;y=ty;n=0xFF;               // Ми почали заливку з (sx,sy), отже
                                  // по шляху доведеться йти з (tx,ty), у зворотньому напрямі
  while((x!=sx)||(y!=sy)){        // Доки не прийшли в (sx,sy)
   scr[y][x*2].attr=2*16;scr[y][x*2+1].attr=2*16;  // Малювання
                                                   // Шукається сусідня комірка,
   if(fillmap[y+1][x  ]<n){tx=x  ;ty=y+1;t=fillmap[y+1][x  ];}    // що містить
   if(fillmap[y-1][x  ]<n){tx=x  ;ty=y-1;t=fillmap[y-1][x  ];}    // мінімальне значення
   if(fillmap[y  ][x+1]<n){tx=x+1;ty=y  ;t=fillmap[y  ][x+1];}
   if(fillmap[y  ][x-1]<n){tx=x-1;ty=y  ;t=fillmap[y  ][x-1];}
   x=tx;y=ty;n=t;                  // Переходимо на знайдену комірку

   getch();                        // Очікуємо натиснення клавіші
  }

}

void main(){
 int i;
 sx=1;sy=1;   // Задання початкової точки
 tx=3;ty=3;   // Задання цілі шляху

 scr=(screen_line*)0xB8000;
 clrscr();
 draw_maze();
 getch();

 fill(sx,sy,tx,ty);     // Виклик функції пошуку шляху

}
```

Див. також

- Алгоритм канального трасування

Література

- *Wai-Kai Chen*. The circuits and filters handbook. — 2, 2003. — 2961 с. — ISBN 0849309123.
- Frank Rubin. The Lee path connection algorithm. — IEEE Transactions on Computers, 1974.

Посилання

- Доведення коректності алгоритму (http://algolist.manual.ru/maths/graphs/shortpath/wave.php) (рос.)