VILNIAUS UNIVERSITETAS

MATEMATIKOS IR INFORMATIKOS FAKULTETAS

INFORMATIKOS KATEDRA

GYTIS KARČIAUSKAS

III kursas 2 grupė

UŽDUOTIS 11

**Kelio labirinte paieška su grįžimu atgal**

v2.0

2011

**Užduotis**

**Duota**: Labirintas, turintis m eilučių ir n stulpelių bei pradinis taškas (i,j).

**Rasti**: Kelią išeiti iš labirinto naudojant paiešką su grįžimu (backtracking).

Realizuoti paieškos su grįžimu algoritmą išėjimui iš labirinto rasti. Ištirti šio algoritmo sudėtingumą:

1. teoriškai,

2. praktiškai kaip priklauso nuo n, m ir pertvarų skaičiaus.

Labirintą galima vaizduoti dviem stačiakampėm m x n matricomis: vienoje vienetu žymime, jei yra

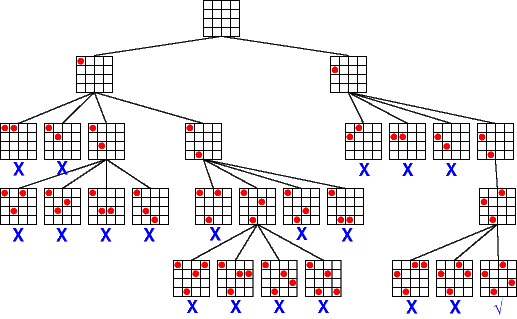
horizontali pertvara ir nuliu, jei jos nėra; kitoje analogiškai žymime vertikalias pertvaras. Galimi ir kiti vaizdavimo būdai.

Šiai užduočiai naudojamas *backtracking* algoritmas yra glaudžiai susijęs su *brute-force* paieškos technika. *Brute-force* paieška yra problemų sprendimo būdas, kuri ieško sprendimo trivialiai generuodamas visus galimus sprendimo variantus ir tikrindamas, ar tas sprendinys atitinka reikiamus kriterijus.

*Backtracking* yra sklandesnė šio būdo versija. Užuot generavęs visus įmanomus variantus, *backtracking* tiesiog suranda visus galimus, kurie atitinka tam tikrus reikalavimus. Jei algoritmas negali surasti reikiamo sprendimo vienu keliu, jis gali grįžti ir bandyti kitą derinį. Praktiškai jis gali išbandyti kiekvieną galimybę.

**Pavyzdys**

Lengviausias būdas pavaizduoti *backtracking* algoritmą yra jį implementuoti kaip medį:



Tarkime ieškome lentoje 4x4 tokio valdovių išdėstymo, kad viena kitos nekirstų. Pirmąją valdovę dėsim į poziciją 1,1. Bandom pastatyti antrą valdovę. Kadangi valdovės gali vaikščioti horizontaliai, vertikaliai, skersai ir išilgai, pirmi du variantai netinka, tad antrąja valdovę dedame į trečią eilutę, (3, 2) poziciją. Nuo šios pozicijos generuojame visus galimus derinius, juos tikriname ir atmetame. Algoritmas sukasi tol, kol randam reikiamą sprendimą (pažymėta varnele).

**Algoritmo kintamieji:**

Algoritmui pateiksim matricą, kurią programiškai sudarys dvimatis masyvas. Programa pati pasiskaičiuos matricos didumą pagal masyvų skaičių masyve bei pirmojo masyvo elementų skaičių.

Matricoje naudojami specialūs ženklai:

* 0 – siena;
* 1 – takelis;
* 2 – pabaigos taškas;
* 3 – pradžios taškas;

*Backtracking‘ui* sekti buvo pasirinktas masyvas robot[], kuriame bus saugomos šakų išsiskyrimo koordinatės bei judėjimo kryptis direction. Jos yra tokios:

* 0 – į kairę;
* 1 – į viršų;
* 2 – į dešinę;
* 3 – į apačią.

Robotas sienas atpažins iš simbolių, esančių ekrane, tad grįžimui atgal nuspręsta naudoti praeito takelio užspalvinimą sienų simboliu. Kada spalvinti, o kada ne – nuspręs režimo kintamasis mode.

Sekti pačio roboto esamą poziciją, yra naudojami curr\_y ir curr\_x kintamieji, sienoms atpažinti – left, top, right bottom kintamieji.

**Algoritmo ištrauka:**

# find current robot position

for i in range(len(matrix)):

for ii in range(len(matrix[0])):

if (window.inch(i, ii) == 79):

curr\_x = ii

curr\_y = i

if ((curr\_y + 1) == len(matrix)): # start position

bottom = 0

direction = 1

elif (window.inch(curr\_y + 1, curr\_x) == 32):

bottom = 1

elif (window.inch(curr\_y + 1, curr\_x) == 88):

bottom = 0

window.addstr(len(matrix) + 5, 0, 'BOTTOM: ' + str(bottom))

if (window.inch(curr\_y, curr\_x + 1) == 88): # if next action is wall

right = 0

elif (window.inch(curr\_y, curr\_x + 1) == 32):

right = 1

window.addstr(len(matrix) + 4, 0, 'RIGHT: ' + str(right))

if (window.inch(curr\_y, curr\_x - 1) == 88):

left = 0

elif (window.inch(curr\_y, curr\_x - 1) == 32):

left = 1

window.addstr(len(matrix) + 2, 0, 'LEFT: ' + str(left))

if (window.inch(curr\_y - 1, curr\_x) == 32): # if empty

top = 1

elif (window.inch(curr\_y - 1, curr\_x) == 88):

top = 0

window.addstr(len(matrix) + 3, 0, 'TOP: ' + str(top))

key = window.getch()

if ( mode == 1):

backtrace = robot[-1]

if ((curr\_y == backtrace[0]) and (curr\_x == backtrace[1])):

if ((left + right + top + bottom) == 1):

robot.pop()

else:

mode = 0

direction = backtrace[2]

if (mode == 1):

window.addch(curr\_y, curr\_x, 'X')

else:

window.addch(curr\_y, curr\_x, ' ')

if (direction == 0):

if ((left == 1) and (top == 0) and (bottom == 0)):

curr\_x -= 1

elif((left == 0) and (top == 1) and (bottom == 0)):

curr\_y -= 1

direction = 1

elif((left == 0) and (top == 0) and (bottom == 0)): # backtrack

mode = 1

direction = 2

elif((left == 0) and (top == 0) and (bottom == 1)): # backtrack

mode = 1

direction = 3

elif (direction == 1):

if ((top == 1) and (left == 0) and (right == 0)):

curr\_y -= 1

elif ((left == 1) and (top == 1) and (right == 1)):

robot.append([curr\_y, curr\_x, direction])

curr\_x -= 1

direction = 0

elif ((left == 1) and (top == 0) and (right == 0)):

curr\_x -= 1

direction = 0

elif ((top == 0) and (left == 0) and (right == 1) and (bottom == 1)):

curr\_x += 1

direction = 2

elif(( top == 0) and (left == 0) and (right == 1) and (bottom == 0)):

mode = 1

direction = 2

elif ((top == 1) and (left == 0)):

curr\_y -= 1

direction = 1

elif ((bottom == 1) and (top == 0) and (left == 0) and (right == 0)):

mode = 1

direction = 3

elif ((left == 1) and (top == 1) and (right == 0)):

robot.append([curr\_y, curr\_x, direction])

direction = 1

curr\_x -= 1

elif (direction == 2):

if ((top == 0) and (right == 1) and (bottom == 0)):

curr\_x += 1

elif ((top == 1) and ((right == 1) or (bottom == 1))):

robot.append([curr\_y, curr\_x, dir])

curr\_y -= 1

direction = 1

elif((top == 0) and (right == 0) and (left == 0) and (mode == 1)):

direction = 3

curr\_y += 1

elif ((top == 1) and (right == 0) and (bottom == 0)):

direction = 1

curr\_y -= 1

elif (direction == 3):

if ((bottom == 1) and (left == 0) and (right == 0)):

curr\_y += 1

elif ((bottom == 1) and (left == 1) and (right == 0)):

if (left == 1):

if ((curr\_x) == robot[-1][1]):

curr\_y += 1

elif ((bottom == 0) and (left == 0) and (right == 0)):

mode = 1

direction = 1

elif ((bottom == 0) and (left == 0) and (right == 1)):

mode = 1

direction = 2

window.addch(curr\_y, curr\_x, 'O')

**Programos naudojimas, parametrai**

$ python backtracking.py

**Eksperimentai**

Su matrica:

[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 1, 1, 0, 0, 0, 1, **2**, 0],

[0, 0, 0, 1, 0, 0, 0, 1, 0, 0],

[0, 0, 0, 1, 0, 0, 1, 1, 0, 0],

[0, 1, 1, 1, 0, 0, 0, 1, 0, 0],

[0, 1, 0, 1, 0, 1, 0, 1, 0, 0],

[0, 1, 0, 0, 0, 1, 0, 1, 0, 0],

[0, 1, 1, 1, 1, 1, 1, 1, 0, 0],

[0, 0, 0, 0, 0, 1, 0, 0, 0, 0],

[0, 0, 0, 0, 0, **3**, 0, 0, 0, 0]]

Ėjimų skaičius: 28;

Pertvarų skaičius: 70

Su matrica:

[[0, 0, 0, 0, 0, 0, 0],

[0, 0, **2**, 1, 0, 0, 0],

[0, 0, 0, 1, 0, 0, 0],

[0, 0, 0, 1, 0, 1, 0],

[0, 1, 1, 1, 1, 1, 0],

[0, 0, 0, 0, 0, **3**, 0]]

Ėjimų skaičius: 18;

Pertvarų skaičius: 28

**Uždavinio sudėtingumas**

Nustatyti uždavinio sudėtingumą yra komplikuota, kadangi labirinto matrica gali susidėti iš daug sienų ir mažai takų roboto ėjimui (tarkim 10x10 matrica, kurioje bus 97 sienos), ir atvirkščiai – daug takų ir mažai sienų. Roboto nueitas kelias iki galutinio taško taip pat gali būti labai įvairus.

**Išvados ir pastebėjimai**

Šis algoritmas panašus į *brute-force* algoritmą, kadangi tinkamą sprendinį gali tiek pirmu bandymu, tiek paskutiniu. Būtent dėl šios savybės yra komplikuota suskaičiuoti algoritmo sudėtingumą.

Taip pat algoritmas gali būti sumažintas iki mimimum, jei robotas visada žiūrės į kairę, paskui į viršų, dešnėn ir į apačią. Esant paskutinėj būsenoj, reiktų robotą apsukti ir ieškoti vėl kairiausio įėjimo. Tačiau šis būdas neturi backtracking‘o...

Pati programa gali būti patobulinta panaudojant funkcijas (taip sumažinant kodo eilučių skaičių), roboto aplinkos stebėjimą įrašant į matricą.