­­МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

­­­



Контрольна робота

з курсу «Методи та системи штучного інтелекту»

для студентів базового напрямку 6.08.04 "Комп’ютерні науки"

(заочна форма навчання)

Варіант 10

Виконав студент гр. КНз-3

Чалий Михайло

­­

Львів 2015

## Мета роботи

Ознайомитися з алгоритмами пошуку оптимального рішення на графах за допомогою манхетенновської відстані на прикладі гри п’ятнашки.

## Завдання

1. Розробити алгоритм проходження гри
2. Отримати індивідуальне завдання.
3. Виконати програму і записати результати

## Реалізація

Лістінг Board.java

**import** java.util.ArrayList;  
**import** java.util.Arrays;  
**import** java.util.List;  
  
*/\*\*  
 \* Created by Mike on 13.07.2014.  
 \*/***public class** Board {  
 **private final int**[][] **blocks**;  
 **private final int N**;  
  
 **private final int zi**;  
 **private final int zj**;  
 **private final int manhattan**;  
  
 *// construct a board from an N-by-N array of blocks  
 // (where blocks[i][j] = block in row i, column j)* **public** Board(**int**[][] blocks) {  
 **this**(blocks, **true**);  
 }  
  
 **private** Board(**int**[][] blocks, **boolean** copy) {  
  
 **this**.**N** = blocks.**length**;  
  
 **if** (copy) {  
 **this**.**blocks** = **new int**[**N**][**N**];  
 } **else** {  
 **this**.**blocks** = blocks;  
 }  
  
 **int** manhattanCandidate = 0;  
 **int** ziCandidate = -1;  
 **int** zjCandidate = -1;  
 **for** (**int** i = 0; i < **N**; i++) {  
 **if** (copy) {  
 **this**.**blocks**[i] = Arrays.*copyOf*(blocks[i], **N**);  
 }  
  
 **for** (**int** j = 0; j < **N**; j++) {  
  
 **int** actual = blocks[i][j];  
 **if** (actual != 0) {  
 **int** ie = (actual - 1) / **N**;  
 **int** je = (actual - 1) % **N**;  
  
 manhattanCandidate += Math.*abs*(ie - i) + Math.*abs*(je - j);  
 } **else** {  
 ziCandidate = i;  
 zjCandidate = j;  
 }  
 }  
 }  
  
 **this**.**manhattan** = manhattanCandidate;  
 **this**.**zi** = ziCandidate;  
 **this**.**zj** = zjCandidate;  
 }  
  
 *// board dimension N* **public int** dimension() {  
 **return N**;  
 }  
  
 *// number of blocks out of place* **public int** hamming() {  
 **int** hamming = 0;  
 **for** (**int** i = 0; i < **N**; i++) {  
 **for** (**int** j = 0; j < **N**; j++) {  
 **int** expected = (i \* **N**) + j + 1;  
 **if** (i == **N** - 1 && j == **N** - 1) {  
 expected = 0;  
 }  
 **int** actual = **blocks**[i][j];  
  
 **if** (expected != actual && !(i == **N** - 1 && j == **N** - 1)) {  
 hamming++;  
 }  
 }  
 }  
  
 **return** hamming;  
 }  
  
 *// sum of Manhattan distances between blocks and goal* **public int** manhattan() {  
 **return manhattan**;  
 }  
  
 *// is this board the goal board?* **public boolean** isGoal() {  
 **return manhattan** == 0;  
 }  
  
 *// a board obtained by exchanging two adjacent blocks in the same row* **public** Board twin() {  
  
 **int**[][] twinBlocks = **new int**[**N**][**N**];  
 **boolean** hasSwap = **false**;  
  
 **for** (**int** i = 0; i < **N**; i++) {  
  
 **if** (**blocks**[i][0] != 0 && **blocks**[i][1] != 0 && !hasSwap) {  
  
 twinBlocks[i][0] = **blocks**[i][1];  
 twinBlocks[i][1] = **blocks**[i][0];  
 **for** (**int** j = 2; j < **N**; j++) {  
 twinBlocks[i][j] = **blocks**[i][j];  
 }  
 hasSwap = **true**;  
  
 } **else** {  
 **for** (**int** j = 0; j < **N**; j++) {  
 twinBlocks[i][j] = **blocks**[i][j];  
 }  
 }  
 }  
  
 **return new** Board(twinBlocks, **false**);  
 }  
  
 *// does this board equal y?* **public boolean** equals(Object other) {  
  
 **if** (other == **this**) **return true**;  
 **if** (other == **null**) **return false**;  
 **if** (other.getClass() != **this**.getClass()) **return false**;  
  
 Board that = (Board) other;  
  
 **if** (that.**N** != **this**.**N**) {  
 **return false**;  
 }  
  
 **for** (**int** i = 0; i < **N**; i++) {  
 **for** (**int** j = 0; j < **N**; j++) {  
 **if** (that.**blocks**[i][j] != **this**.**blocks**[i][j]) {  
 **return false**;  
 }  
 }  
 }  
  
 **return true**;  
 }  
  
 *// all neighboring boards* **public** Iterable<Board> neighbors() {  
  
 List<Board> neighbors = **new** ArrayList<>();  
 **int**[][] moves = {{**zi** - 1, **zj**}, {**zi**, **zj** - 1}, {**zi**, **zj** + 1}, {**zi** + 1, **zj**}};  
  
 **for** (**int**[] m : moves) {  
 **int** i = m[0];  
 **int** j = m[1];  
  
 **if** (i < 0 || i >= **N** || j < 0 || j >= **N**) {  
 **continue**;  
 }  
  
 **int**[][] neighborBlocks = *copyBlocks*(**blocks**);  
  
 neighborBlocks[**zi**][**zj**] = **blocks**[i][j];  
 neighborBlocks[i][j] = 0;  
  
 neighbors.add(**new** Board(neighborBlocks, **false**));  
 }  
  
 **return** neighbors;  
 }  
  
 **private static int**[][] copyBlocks(**int**[][] blocks) {  
 **int** N = blocks.**length**;  
 **int**[][] copy = **new int**[N][N];  
  
 **for** (**int** i = 0; i < N; i++) {  
 copy[i] = Arrays.*copyOf*(blocks[i], N);  
 }  
  
 **return** copy;  
 }  
  
  
 *// string representation of the board (in the output format specified below)* **public** String toString() {  
 StringBuilder s = **new** StringBuilder();  
 s.append(**N** + **"\n"**);  
 **for** (**int** i = 0; i < **N**; i++) {  
 **for** (**int** j = 0; j < **N**; j++) {  
 s.append(String.*format*(**"%2d "**, **blocks**[i][j]));  
 }  
 s.append(**"\n"**);  
 }  
 **return** s.toString();  
  
 }  
  
}

Лістінг Solver.java

**import** java.util.ArrayList;  
**import** java.util.Comparator;  
**import** java.util.List;  
  
*/\*\*  
 \* Created by Mike on 13.07.2014.  
 \*/***public class** Solver {  
  
 **private final int moves**;  
 **private final** List<Board> **solution**;  
 **private final boolean solvable**;  
  
 **private static class** Step {  
 **private int move**;  
 **private final** Board **board**;  
 **private** Step **previous**;  
 **private final int manhattanPriority**;  
  
 **private** Step(**int** move, Board board, Step previous) {  
 **this**.**move** = move;  
 **this**.**board** = board;  
 **this**.**previous** = previous;  
 **this**.**manhattanPriority** = move + board.manhattan();  
 }  
  
 **int** manhattanPriority() {  
 **return manhattanPriority**;  
 }  
  
 **public** Board getBoard() {  
 **return board**;  
 }  
  
 **public** Step getPrevious() {  
 **return previous**;  
 }  
  
 @Override  
 **public** String toString() {  
 **return ">> moves "** + **move** + **"\r\n"** + **"priority="** + **manhattanPriority** + **"\r\n"** + **board**;  
 }  
 }  
  
 *// find a solution to the initial board (using the A\* algorithm)* **public** Solver(Board initial) {  
  
 Comparator<Step> manhattanComparator = **new** Comparator<Step>() {  
 @Override  
 **public int** compare(Step s1, Step s2) {  
 **if** (s1.manhattanPriority() == s2.manhattanPriority()) {  
 **return** s1.**board**.manhattan() - s2.**board**.manhattan();  
 }  
 **return** s1.manhattanPriority() - s2.manhattanPriority();  
 }  
 };  
  
 **boolean** isSolvable = **true**;  
 MinPQ<Step> queue = **new** MinPQ<>(manhattanComparator);  
 MinPQ<Step> twinQueue = **new** MinPQ<>(manhattanComparator);  
  
 queue.insert(**new** Step(0, initial, **null**));  
 twinQueue.insert(**new** Step(0, initial.twin(), **null**));  
  
 Step currentStep = **null**;  
  
 **while** (**true**) {  
 Step currentTwinStep = twinQueue.delMin();  
 **if** (currentTwinStep.getBoard().isGoal()) {  
 isSolvable = **false**;  
 **break**;  
 }  
  
 currentStep = queue.delMin();  
 Board current = currentStep.getBoard();  
  
 **if** (current.isGoal()) {  
 **break**;  
 }  
  
 addNeighbours(twinQueue, currentTwinStep);  
 addNeighbours(queue, currentStep);  
 }  
  
 **this**.**solvable** = isSolvable;  
 **if** (isSolvable) {  
 **this**.**solution** = **new** ArrayList<>();  
 Step tmp = currentStep;  
 **while** (tmp != **null**) {  
 **solution**.add(0, tmp.**board**);  
 tmp = tmp.**previous**;  
 }  
 **this**.**moves** = **solution**.size() - 1;  
 } **else** {  
 **this**.**moves** = -1;  
 **this**.**solution** = **null**;  
 }  
  
 }  
  
 **private void** addNeighbours(MinPQ<Step> queue, Step currentStep) {  
 Step previousStep = currentStep.getPrevious();  
 **boolean** grannyFound = **false**;  
 **for** (Board neighbour : currentStep.**board**.neighbors()) {  
 **if** (previousStep == **null** || grannyFound || !neighbour.equals(previousStep.**board**)) {  
 queue.insert(**new** Step(currentStep.**move** + 1, neighbour, currentStep));  
 } **else** {  
 grannyFound = **true**;  
 }  
 }  
 }  
  
 *// is the initial board solvable?* **public boolean** isSolvable() {  
 **return solvable**;  
 }  
  
 *// min number of moves to solve initial board; -1 if no solution* **public int** moves() {  
 **return moves**;  
 }  
  
 *// sequence of boards in a shortest solution; null if no solution* **public** Iterable<Board> solution() {  
 **return solution**;  
 }  
  
 *// solve a slider puzzle (given below)* **public static void** main(String[] args) {  
  
 *// create initial board from file* In in = **new** In(args[0]);  
 **int** N = in.readInt();  
 **int**[][] blocks = **new int**[N][N];  
 **for** (**int** i = 0; i < N; i++)  
 **for** (**int** j = 0; j < N; j++)  
 blocks[i][j] = in.readInt();  
 Board initial = **new** Board(blocks);  
  
 *// solve the puzzle* Solver solver = **new** Solver(initial);  
  
 *// print solution to standard output* **if** (!solver.isSolvable())  
 StdOut.*println*(**"No solution possible"**);  
 **else** {  
 StdOut.*println*(**"Minimum number of moves = "** + solver.moves());  
 **for** (Board board : solver.solution())  
 StdOut.*println*(board);  
 }  
 }  
  
}

## Результат

## Висновки

Ознайомився з алгоритмами пошуку оптимального рішення на графах за допомогою манхетенновської відстані на прикладі гри п’ятнашки.