# 《人工智能》实验二报告

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名 | 俞家宝 | 学号 | 2021117338 |
| 实验地点 | 实验室321 | 实验日期 | 2023/10/23 |

**一、实验内容**

Building the following state space search algorithms in Prolog or Lisp or Java :

（1）depth-first or breadth-first ( iterative, not recursive )

（2）best-first search

* These search control algorithms should be applied to one of the following problems:

（1）The 8-puzzle (p. 89), and

（2）The Tile Problem ( p. 162, no. 5 ).

* Note that these problems also give the students a nice opportunity to consider various representations and their appropriateness for each problem.

1. **实验原理**
2. 使用宽度优先搜索方法解决该问题

广度优先是从初始状态一层一层向下找，直到找到目标为止。每遍历一层都将一层的所有节点加入OPEN表。

2. 使用启发式搜索

特点：重排OPEN表，选择最有希望的节点加以扩展，使用A\*算法

用来加速搜索过程的有关问题领域的特征信息。

包括：

用于决定要扩展的下一个节点的信息；

在扩展一个节点时，用于决定要生成哪一个或哪几个后继节点的信息;

用于决定某些应该从搜索树中抛弃或修剪的节点的信息；

使用启发式信息指导的搜索过程称为启发式搜索.

估价函数：f(n)=d(n)+w(n)

:::

在本人使用的估价函数是搜索代价+曼哈顿距离

搜索代价：每向深一层搜索则搜索代价自增1

曼哈顿函数：每个数字距离目标位置的曼哈顿距离之和

1. **实验过程以及结果分析**

目录：

[宽度优先搜索](#宽度优先搜索)

[启发式搜索](#启发式搜索)

[**宽度优先搜索运行结果**](#宽度优先搜索运行结果)

[**启发式搜索运行结果**](#启发式搜索运行结果)

1. **Breadth-first宽度优先搜索**

import java.util.\*;  
*/\*\*  
 \* @author 俞家宝  
 \* @version 1.0  
 \* @className BreadthFirstAlgorithms  
 \* @dataTime 2023/10/23 20:49  
 \*/*public class BreadthFirstAlgorithms {  
 public static class PuzzleState {  
 int[][] puzzle;  
 PuzzleState parent;  
 PuzzleState(int[][] puzzle, PuzzleState parent) {  
 this.parent = parent;  
 this.puzzle = puzzle;  
 }  
 }  
 public static int[][] *goal* = {{1, 2, 3}, {8, 0, 4}, {7, 6, 5}};  
 public static int[][] *initBoard* = { {2, 8, 3}, {1, 6, 4}, {7, 0, 5}};  
 public static void main(String[] args) {  
 PuzzleState solution = *search*(*initBoard*, *goal*);  
 if (solution == null) {  
 System.*out*.println("无解");  
 } else {  
 List<PuzzleState> path = new ArrayList<>();  
 PuzzleState current = solution;  
 while (current != null) {  
 path.add(current);  
 current = current.parent;  
 }  
 Collections.*reverse*(path);  
 StringBuilder solvePath = new StringBuilder();  
 for (int i = 0; i < path.size() - 1; i++) {  
 solvePath.append(*getPoint*(path.get(i).puzzle, path.get(i + 1).puzzle)).append("->");  
 }  
 solvePath.delete(solvePath.length() - 2, solvePath.length());  
 for (PuzzleState node : path) {  
 *outPut*(node.puzzle);  
 System.*out*.println();  
 }  
 System.*out*.println(solvePath);  
 }  
 }

//执行宽度优先搜索  
 public static PuzzleState search(int[][] initial, int[][] goal) {  
 Queue<PuzzleState> open = new ArrayDeque<>();  
 open.add(new PuzzleState(initial, null));  
 Set<String> visited = new HashSet<>();  
 while (!open.isEmpty()) {  
 PuzzleState current = open.poll();  
 if (Arrays.*deepEquals*(current.puzzle, goal)) {  
 return current; // 找到解  
 }  
 visited.add(Arrays.*deepToString*(current.puzzle));  
 int[] dx = {-1, 1, 0, 0};  
 int[] dy = {0, 0, -1, 1};  
 for (int i = 0; i < 4; i++) {  
 int x = -1;  
 int y = -1;  
 // 找到空白块的位置  
 for (int j = 0; j < 3; j++) {  
 for (int k = 0; k < 3; k++) {  
 if (current.puzzle[j][k] == 0) {  
 x = j;  
 y = k;  
 break;  
 }  
 }  
 }  
 int newX = x + dx[i];  
 int newY = y + dy[i];  
 if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {  
 int[][] newState = new int[3][3];  
 for (int j = 0; j < 3; j++) {  
 newState[j] = Arrays.*copyOf*(current.puzzle[j], 3);  
 }  
 // 交换空白块和相邻的块  
 int temp = newState[x][y];  
 newState[x][y] = newState[newX][newY];  
 newState[newX][newY] = temp;  
 if (!visited.contains(Arrays.*deepToString*(newState))) {  
 open.add(new PuzzleState(newState, current));  
 }  
 }  
 }  
 }  
 return null; // 无解  
 }  
 static void outPut(int[][] state) {  
 for (int[] ints : state) {  
 for (int anInt : ints) {  
 System.*out*.print(anInt + " ");  
 }  
 System.*out*.println();  
 }  
 }

//得到空白快的位置

static List<Integer> getEmptyLocation(int[][] puzzle) {  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 if (puzzle[i][j] == 0) {  
 return new ArrayList<>(Arrays.*asList*(i, j));  
 }  
 }  
 }  
 return new ArrayList<>(Arrays.*asList*(0, 0));  
 }

//得到每一步的交换方向

static String getPoint(int[][] current, int[][] next) {  
 int[] dx = {-1, 1, 0, 0};  
 int[] dy = {0, 0, -1, 1};  
 for (int i = 0; i < 4; i++) {  
 List<Integer> currentEmptyLocation = *getEmptyLocation*(current);  
 List<Integer> nextEmptyLocation = *getEmptyLocation*(next);  
 int newX = currentEmptyLocation.get(0) + dx[i];  
 int newY = currentEmptyLocation.get(1) + dy[i];  
 if (newX == nextEmptyLocation.get(0) && newY == nextEmptyLocation.get(1)) {  
 switch (i) {  
 case 0: {  
 return "UP";  
 }  
 case 1: {  
 return "DOWN";  
 }  
 case 2: {  
 return "LEFT";  
 }  
 case 3: {  
 return "RIGHT";  
 }  
 }  
 }  
 }  
 return null;  
 }  
}

1. **Best-first****启发式搜索**

import java.util.\*;  
*/\*\*  
 \* @author 俞家宝  
 \* @version 1.0  
 \* @className* BestFirstAlgorithms *\* @dataTime 2023/10/23 18:15  
 \*/*public class BestFirstAlgorithms {  
 public static class PuzzleState {  
 int[][] puzzle;  
 PuzzleState parent;  
 int cost;  
 PuzzleState(int[][] puzzle, PuzzleState parent, int cost) {  
 this.parent = parent;  
 this.puzzle = puzzle;  
 this.cost = cost;  
 }  
 }  
 public static int[][] *goal* = {{1, 2, 3}, {8, 0, 4}, {7, 6, 5}};  
 public static int[][] *initBoard* = {{2, 8, 3}, {1, 6, 4}, {7, 0, 5}};  
 public static void main(String[] args) {  
 PuzzleState solution = *search*(*initBoard*, *goal*);  
 if (solution == null) {  
 System.*out*.println("无解");  
 } else {  
 List<PuzzleState> path = new ArrayList<>();  
 PuzzleState current = solution;  
 while (current != null) {  
 path.add(current);  
 current = current.parent;  
 }  
 Collections.*reverse*(path);  
 StringBuilder solvePath = new StringBuilder();  
 for (int i = 0; i < path.size() - 1; i++) {  
 solvePath.append(*getPoint*(path.get(i).puzzle, path.get(i + 1).puzzle)).append("->");  
 }  
 solvePath.delete(solvePath.length() - 2, solvePath.length());  
 for (PuzzleState node : path) {  
 *outPut*(node.puzzle);  
 System.*out*.println();  
 }  
 System.*out*.println(solvePath);  
 }  
 }

//计算曼哈顿距离  
 static int heuristic(int[][] puzzle, int[][] goal) {  
 int h = 0;  
 for (int i = 0; i < puzzle.length; i++) {  
 for (int j = 0; j < puzzle[i].length; j++) {  
 int value = puzzle[i][j];  
 for (int x = 0; x < goal.length; x++) {  
 for (int y = 0; y < goal[x].length; y++) {  
 if (value == goal[x][y] && value != 0) {  
 h += Math.*abs*(i - x) + Math.*abs*(j - y);  
 }  
 }  
 }  
 }  
 }  
 return h;  
 }  
//进行启发式搜索A\*  
 public static PuzzleState search(int[][] initial, int[][] goal) {  
 PriorityQueue<PuzzleState> open = new PriorityQueue<>(Comparator.*comparingInt*(n -> n.cost + *heuristic*(n.puzzle, goal)));  
 open.add(new PuzzleState(initial, null, 0));  
 Set<String> visited = new HashSet<>();  
 while (!open.isEmpty()) {  
 PuzzleState current = open.poll();  
 if (Arrays.*deepEquals*(current.puzzle, goal)) {  
 return current; // 找到解  
 }  
 visited.add(Arrays.*deepToString*(current.puzzle));  
 int[] dx = {-1, 1, 0, 0};  
 int[] dy = {0, 0, -1, 1};  
 for (int i = 0; i < 4; i++) {  
 int x = -1;  
 int y = -1;  
 // 找到空白块的位置  
 for (int j = 0; j < 3; j++) {  
 for (int k = 0; k < 3; k++) {  
 if (current.puzzle[j][k] == 0) {  
 x = j;  
 y = k;  
 break;  
 }  
 }  
 }  
 int newX = x + dx[i];  
 int newY = y + dy[i];  
 if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {  
 int[][] newState = new int[3][3];  
 for (int j = 0; j < 3; j++) {  
 newState[j] = Arrays.*copyOf*(current.puzzle[j], 3);  
 }  
 // 交换空白块和相邻的块  
 int temp = newState[x][y];  
 newState[x][y] = newState[newX][newY];  
 newState[newX][newY] = temp;  
 if (!visited.contains(Arrays.*deepToString*(newState))) {  
 open.add(new PuzzleState(newState, current, current.cost + 1));  
 }  
 }  
 }  
 }  
 return null; // 无解  
 }

//输出九宫格  
 static void outPut(int[][] puzzle) {  
 for (int[] ints : puzzle) {  
 for (int anInt : ints) {  
 System.*out*.print(anInt + " ");  
 }  
 System.*out*.println();  
 }  
 }

//得到空白快位置  
 static List<Integer> getEmptyLocation(int[][] puzzle) {  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 if (puzzle[i][j] == 0) {  
 return new ArrayList<>(Arrays.*asList*(i, j));  
 }  
 }  
 }  
 return new ArrayList<>(Arrays.*asList*(0, 0));  
 }  
//得到每一步的交换方向  
 static String getPoint(int[][] current, int[][] next) {  
 int[] dx = {-1, 1, 0, 0};  
 int[] dy = {0, 0, -1, 1};  
 for (int i = 0; i < 4; i++) {  
 List<Integer> currentEmptyLocation = *getEmptyLocation*(current);  
 List<Integer> nextEmptyLocation = *getEmptyLocation*(next);  
 int newX = currentEmptyLocation.get(0) + dx[i];  
 int newY = currentEmptyLocation.get(1) + dy[i];  
 if (newX == nextEmptyLocation.get(0) && newY == nextEmptyLocation.get(1)) {  
 switch (i) {  
 case 0: {  
 return "UP";  
 }  
 case 1: {  
 return "DOWN";  
 }  
 case 2: {  
 return "LEFT";  
 }  
 case 3: {  
 return "RIGHT";  
 }  
 }  
 }  
 }  
 return null;  
 }  
}

**运行结果截图：**

1. **宽度优先搜索运行结果：**

**电子计算机

中度可信度描述已自动生成--------->电子计算机

中度可信度描述已自动生成**

1. **启发式搜索运行结果****：**

**屏幕上有字

描述已自动生成--->屏幕上有字

描述已自动生成--->屏幕上有字

描述已自动生成--->屏幕上有字

描述已自动生成--->屏幕上有字

描述已自动生成--->屏幕上有字

描述已自动生成**

**屏幕上有字

描述已自动生成**

1. **实验总结**

**实现启发式搜索和宽度优先搜索之后，比较明显发现，相比于宽度优先搜索，启发式搜索：**

更高的效率：使用启发式函数来估计从起点到目标的最短路径成本，这使得它能够更快地找到最优解。相比之下，宽度优先搜索不使用这种估计，因此可能会探索大量不必要的状态，尤其在问题空间很大时，效率差。

更好的内存利用：通常使用较少的内存，因为它只保持一个较小的优先队列，而宽度优先搜索需要存储所有当前层级的状态，这在问题空间较大时会导致内存问题。

可用于有权重的图：可以轻松处理有权重的图，因为它考虑了路径成本。这在很多实际问题中非常有用，比如寻找最短路径时，每条边可能有不同的成本。

适应性更强：启发式函数允许算法根据问题的特性进行自适应调整。这意味着你可以根据具体问题调整启发式函数，以改善搜索性能。宽度优先搜索则没有这种适应性。

可解决更广泛的问题：由于启发式搜索灵活性，它可以应用于各种不同类型的问题，包括图搜索、路径规划、游戏搜索以及其他优化问题。相比之下，宽度优先搜索更适合特定类型的问题，如无权重图的最短路径。

**以下是搜索过程当中的队列情况：**

**宽度优先：**

**图片包含 文本

描述已自动生成**

**启发式：**

**图片包含 背景图案

描述已自动生成**

**其中启发式只进行了6次移动尝试，而宽度优先进行了34次才找到最终解**