**인공지능 과제2**

(Eight Puzzle Problem)

logo.gif

|  |  |
| --- | --- |
| 과목명 | 인공지능 |
| 담당교수 | 이수원 |
| 학과 | 소프트웨어학부 |
| 학년 | 4 |
| 학번 | 20152994 |
| 성명 | 이진영 |
| 제출일 | 2018.4.17 |



**목 차**

**1. 구현 방법**

**2. 소스 코드**

**3. 실행 결과**

**4. 랜덤 실행 결과**

**Eight Puzzle AI 프로그램**

20152994 이진영

**1. 구현 방법**

① 자료구조

- puzzle 정보를 저장하는 Puzzle 클래스와 tree 정보를 저장하는 Node클래스,

BestNode 클래스 사용

- BestNode 클래스는 Node 클래스에서 가중치 값인 f(n), g(n), h(n) 에 대한 정보만 추가

- open\_list, closed\_list는 C++의 deque 클래스를 이용해 구현

② 함수

- breadth\_first\_seach() : 함수 내부에서 open\_list를 Queue처럼 사용하면서 node를 방문하고

tree를 생성하면서 solution을 찾을 때까지 loop한다.

- depth\_first\_search() : 함수 내부에서 open\_list를 Stack처럼 사용하면서 node를 방문하고

tree를 생성하면서 solution을 찾을 때까지 loop한다.

- best\_first\_search() : 함수 내부에서 open\_list를 Queue처럼 사용하는데

open\_list에 node들이 추가될 때마다 가중치 f 값을 기준으로

오름차순으로 정렬하여 가장 낮은 f값을 가진 node부터

방문하면서 tree를 생성하고 solution을 찾을 때 까지 loop한다.

- 주어진 puzzle을 풀 수 있는지 여부를 확인하는 함수 getInvCount()와 isSolvable()은

온라인에서 찾아 사용함

(http://massivealgorithms.blogspot.kr/2015/06/how-to-check-if-instance-of-8-puzzle-is.html)

- getInvCount()에서 inversion값은 puzzle의 숫자들을 일렬로 나열했을 때

각 숫자보다 작은 숫자가 다음 숫자들 중에 있을 때마다 1개씩 증가하고

작성한 프로그램에서는 inversion값이 홀수이면 solvable

**2. 소스코드**

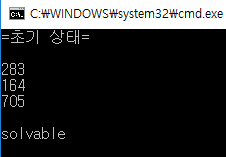
① <search.cpp>

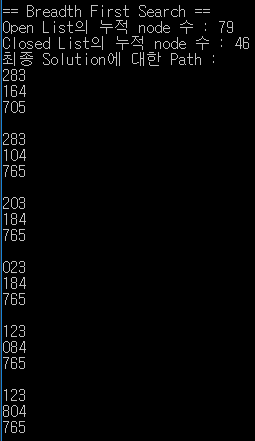
|  |
| --- |
| #include <iostream>  #include <stdio.h>  #include <deque>  #include <iterator>  #include <algorithm>  #include "puzzle.h"  using namespace std;  deque<Node> open\_list;  deque<Node> closed\_list;  deque<BestNode> open\_lb;  deque<BestNode> closed\_lb;  unsigned int open\_num = 0; // open\_list의 누적 node 수  unsigned int closed\_num = 0; // closed\_list의 누적 node 수  void breadth\_first\_search(Node\* root);  void depth\_first\_search(Node\* root, int bound);  void best\_first\_search(BestNode\* root);  void printPath(Node\* term);  void printPathB(BestNode\* term);  void printState(int puzzle[3][3]);  bool compare(const BestNode& node1, const BestNode& node2); // best-first-search를 수행할때 sort()함수에 사용되는 비교함수  int getInvCount(int arr[]); // 주어진 puzzle의 inversion을 구하는 함수  bool isSolvable(int puzzle[3][3]); // 주어진 puzzle이 solvable한지 확인하는 함수  void main() {  int ex\_num[3][3] = { {2,8,3},{1,6,4},{7,0,5} }; // 교재의 Problem Instance 생성  Puzzle\* ex\_puzzle = new Puzzle(ex\_num);  Node\* root = new Node(ex\_puzzle, 0);  printf("=초기 상태=\n\n");  printState(ex\_num);  if (isSolvable(ex\_num)) { // 풀 수 있는 퍼즐인지 확인  printf("solvable\n\n");  breadth\_first\_search(root);  printf("== Breadth First Search ==\n");  printf("Open List의 누적 node 수 : %d\n", open\_num);  printf("Closed List의 누적 node 수 : %d\n", closed\_num);  printf("최종 Solution에 대한 Path : \n");  printPath(&closed\_list.front());  open\_list.clear();  closed\_list.clear();  open\_num = 0;  closed\_num = 0;  depth\_first\_search(root, 5);  printf("== Depth First Search ==\n");  printf("Open List의 누적 node 수 : %d\n", open\_num);  printf("Closed List의 누적 node 수 : %d\n", closed\_num);  printf("최종 Solution에 대한 Path : \n");  printPath(&closed\_list.front());  open\_list.clear();  closed\_list.clear();  open\_num = 0;  closed\_num = 0;  BestNode\* root\_best = new BestNode(ex\_puzzle, 0);  best\_first\_search(root\_best);  printf("== Best First Search ==\n");  printf("Open List의 누적 node 수 : %d\n", open\_num);  printf("Closed List의 누적 node 수 : %d\n", closed\_num);  printf("최종 Solution에 대한 Path : \n");  printPathB(&closed\_lb.back());  }  else  printf("not solvable\n\n");  }  void breadth\_first\_search(Node\* root) {  Node\* temp = root;  open\_list.push\_back(\*root); // deque type인 open\_list를 Queue처럼 사용  open\_num++;  while ((temp->puzzle.isSolution() == false) && (!open\_list.empty())) {  closed\_num++;  closed\_list.push\_front(\*temp);  open\_list.pop\_front(); // 확인한 node는 open\_list에서 삭제  char dir[4];  int possible = temp->getPossibleMove(dir);  closed\_list.front().next = new Node\*[possible];  for (int i = 0; i < possible; i++) {  temp->puzzle.do\_move(dir[i]);  Node\* child = new Node(&temp->puzzle ,temp->depth + 1);  temp->puzzle.undo\_move();  child->parent = &closed\_list.front();  closed\_list.front().next[i] = child;  open\_list.push\_back(\*child);  open\_num++;  delete child;    }  temp = &open\_list.front();  }  if (temp->puzzle.isSolution() == true) {  closed\_list.push\_front(\*temp);  closed\_num++;  }  }  void depth\_first\_search(Node\* root, int bound) {  Node\* temp = root;  open\_list.push\_front(\*root); // deque type인 open\_list를 Stack처럼 사용  open\_num++;  while ((temp->puzzle.isSolution() == false) && (!open\_list.empty())) {  closed\_num++;  closed\_list.push\_front(\*temp);  open\_list.pop\_front(); // 확인한 node는 open\_list에서 삭제  char dir[4];  int possible = temp->getPossibleMove(dir);  closed\_list.front().next = new Node\*[possible];  if (closed\_list.front().depth < bound) {  for (int i = possible - 1; i >= 0; i--) {  closed\_list.front().puzzle.do\_move(dir[i]);  Node\* child = new Node(&closed\_list.front().puzzle, closed\_list.front().depth+1);  closed\_list.front().puzzle.undo\_move();  child->parent = &closed\_list.front();  closed\_list.front().next[possible - i - 1] = child;  open\_list.push\_front(\*child);  open\_num++;  }  }  if(!open\_list.empty())  temp = &open\_list.front();  }  if (temp->puzzle.isSolution() == true) {  closed\_list.push\_front(\*temp);  closed\_num++;  }  }  void best\_first\_search(BestNode\* root) {  BestNode\* temp = root;  open\_lb.push\_back(\*root); // deque type인 open\_list를 Queue처럼 사용  open\_num++;  while ((temp->puzzle.isSolution() == false) && (!open\_lb.empty())) {  closed\_num++;  closed\_lb.push\_back(\*temp);  open\_lb.pop\_front(); // 확인한 node는 open\_list에서 삭제  char dir[4];  int possible = temp->getPossibleMove(dir);  closed\_lb.front().next = new BestNode\*[possible];  for (int i = 0; i < possible; i++) {  temp->puzzle.do\_move(dir[i]);  BestNode\* child = new BestNode(&temp->puzzle, temp->depth + 1);  temp->puzzle.undo\_move();  child->parent = &closed\_lb.back();  closed\_lb.front().next[i] = child;  open\_lb.push\_back(\*child);  open\_num++;  }  sort(open\_lb.begin(), open\_lb.end(), compare);    temp = &open\_lb.front();  }  if (temp->puzzle.isSolution() == true) {  closed\_lb.push\_back(\*temp);  closed\_num++;  }  }  void printPath(Node\* term) {  int tmp = term->depth;  term->mark = true;    for (int i = 0; i < tmp; i++) {  term = term->parent;  term->mark = true;  }  deque<Node>::iterator it;  it = closed\_list.end();  for (it = closed\_list.end()-1; it >= closed\_list.begin(); it--) {  if (it->mark == true) {  for (int i = 0; i < 3; i++) {  for (int j = 0; j < 3; j++) {  printf("%d", it->puzzle.info[i][j]);  }  printf("\n");  }  printf("\n");  }  if (it == closed\_list.begin())  break;  }  }  void printPathB(BestNode\* term) {  int tmp = term->depth;  term->mark = true;  for (int i = 0; i < tmp; i++) {  term = term->parent;  term->mark = true;  }  deque<BestNode>::iterator it;  for (it = closed\_lb.begin(); it < closed\_lb.end(); it++) {  if (it->mark == true) {  for (int i = 0; i < 3; i++) {  for (int j = 0; j < 3; j++) {  printf("%d", it->puzzle.info[i][j]);  }  printf("\n");  }  printf("\n");  }  }  }  void printState(int puzzle[3][3])  {  for (int i = 0; i < 3; i++) {  for (int j = 0; j < 3; j++) {  printf("%d", puzzle[i][j]);  }  printf("\n");  }  printf("\n");  }  bool compare(const BestNode& node1, const BestNode& node2) {  return node1.f < node2.f;  }  // A utility function to count inversions in given array 'arr[]'  int getInvCount(int arr[])  {  int inv\_count = 0;  for (int i = 0; i < 9 - 1; i++)  for (int j = i + 1; j < 9; j++)  // Value 0 is used for empty space  if (arr[j] && arr[i] && arr[i] > arr[j])  inv\_count++;  return inv\_count;  }  // This function returns true if given 8 puzzle is solvable.  bool isSolvable(int puzzle[3][3])  {  // Count inversions in given 8 puzzle  int invCount = getInvCount((int \*)puzzle);  // return true if inversion count is even.  return (invCount % 2 == 1);  } |

② <puzzle.h>

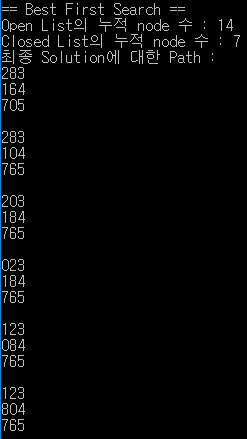
|  |
| --- |
| const int solution[3][3] = { { 1,2,3 },  { 8,0,4 },  { 7,6,5 } };  class Position {  public:  int row; // 행  int col; // 열  };  class Puzzle {  public:  int info[3][3]; // 퍼즐에 존재하는 숫자 저장, 숫자 0은 빈 칸을 의미  Position empty\_idx; // 퍼즐에서 빈 칸의 num배열에서의 index 저장  bool didMove; // 수행된 move에 대해 undo가 한번만 수행될 수 있도록 move여부 정보 저장  char preDir; // 수행된 move의 정보를 저장하여 undo 할때 어떤방향으로 가야할지 알려줌  public:  Puzzle() {}  Puzzle(int number[][3]) { // Puzzle의 생성자 (배열형태의 puzzle 정보를 받아 객체 생성)  for (int i = 0; i < 3; i++)  for (int j = 0; j < 3; j++)  info[i][j] = number[i][j];  getEmptyIdx();  didMove = false;  preDir = '\0';  }  bool isSolution() { // Puzzle이 solution에 도달했는지 여부 확인  for (int i = 0; i < 3; i++)  for (int j = 0; j < 3; j++)  if (solution[i][j] != info[i][j])  return false;  return true;  }  bool isSame(Puzzle\* compare) { // Puzzle이 같은지 확인  for (int i = 0; i < 3; i++)  for (int j = 0; j < 3; j++)  if (compare->info[i][j] != info[i][j])  return false;  return true;  }  Position getEmptyIdx() { // 비어있는 칸의 위치 리턴  for (int i = 0; i < 3; i++) {  for (int j = 0; j < 3; j++) {  if (info[i][j] == 0) {  empty\_idx.row = i;  empty\_idx.col = j;  }  }  }  return empty\_idx;  }  void do\_move(char dir) {  switch (dir)  {  case 'l':  move\_left();  break;  case 'u':  move\_up();  break;  case 'd':  move\_down();  break;  case 'r':  move\_right();  break;  default:  break;  }  }  void move\_left() {  if (empty\_idx.col - 1 < 0) {  didMove = false;  preDir = '\0';  } else {  info[empty\_idx.row][empty\_idx.col] =  info[empty\_idx.row][empty\_idx.col - 1];  empty\_idx.col--;  info[empty\_idx.row][empty\_idx.col] = 0;  didMove = true;  preDir = 'l';  }  }  void move\_right() {  if (empty\_idx.col + 1 > 2) {  didMove = false;  preDir = '\0';  } else {  info[empty\_idx.row][empty\_idx.col] =  info[empty\_idx.row][empty\_idx.col + 1];  empty\_idx.col++;  info[empty\_idx.row][empty\_idx.col] = 0;  didMove = true;  preDir = 'r';  }  }  void move\_up() {  if (empty\_idx.row - 1 < 0) {  didMove = false;  preDir = '\0';  } else {  info[empty\_idx.row][empty\_idx.col] =  info[empty\_idx.row-1][empty\_idx.col];  empty\_idx.row--;  info[empty\_idx.row][empty\_idx.col] = 0;  didMove = true;  preDir = 'u';  }  }  void move\_down() {  if (empty\_idx.row + 1 > 2) {  didMove = false;  preDir = '\0';  } else {  info[empty\_idx.row][empty\_idx.col] =  info[empty\_idx.row + 1][empty\_idx.col];  empty\_idx.row++;  info[empty\_idx.row][empty\_idx.col] = 0;  didMove = true;  preDir = 'd';  }  }  void undo\_move() {  if (didMove == true) {  switch (preDir)  {  case 'l':  move\_right();  didMove = false;  preDir = '\0';  break;  case 'r':  move\_left();  didMove = false;  preDir = '\0';  break;  case 'u':  move\_down();  didMove = false;  preDir = '\0';  break;  case 'd':  move\_up();  didMove = false;  preDir = '\0';  break;  default:  break;  }  }  }  };  class Node { // Puzzle과 Tree 정보를 저장하기 위한 Node 클래스 정의  public:  Puzzle puzzle;  int depth;  Node\* parent;  Node\*\* next;  bool mark;  public:  Node() {}  Node(Puzzle\* p, int depth) { // Puzzle과 depth 정보를 받아 Node를 생성  for (int i = 0; i < 3; i++)  for (int j = 0; j < 3; j++)  puzzle.info[i][j] = p->info[i][j];  puzzle.didMove = p->didMove;  puzzle.empty\_idx = p->empty\_idx;  puzzle.preDir = p->preDir;  parent = NULL;  next = NULL;  this->depth = depth;  mark = false;  }  int getPossibleMove(char dir[4]) { // possible로 빈칸이 움직일 수 있는 개수 리턴  int possible = 0; // dir로 움직일 수 있는 방향 리턴  if (parent == NULL) {  puzzle.move\_left();  if (puzzle.didMove == true) {  dir[possible] = 'l';  possible++;  }  puzzle.undo\_move();  puzzle.move\_up();  if (puzzle.didMove == true) {  dir[possible] = 'u';  possible++;  }  puzzle.undo\_move();  puzzle.move\_right();  if (puzzle.didMove == true) {  dir[possible] = 'r';  possible++;  }  puzzle.undo\_move();  puzzle.move\_down();  if (puzzle.didMove == true) {  dir[possible] = 'd';  possible++;  }  puzzle.undo\_move();  } else { //parent Node와 동일한 Puzzle을 생성하지 않도록 동일성 확인  puzzle.move\_left();  if (!puzzle.isSame(&parent->puzzle)) {  if (puzzle.didMove == true) {  dir[possible] = 'l';  possible++;  }  }  puzzle.undo\_move();  puzzle.move\_up();  if (!puzzle.isSame(&parent->puzzle)) {  if (puzzle.didMove == true) {  dir[possible] = 'u';  possible++;  }  }  puzzle.undo\_move();  puzzle.move\_right();  if (!puzzle.isSame(&parent->puzzle)) {  if (puzzle.didMove == true) {  dir[possible] = 'r';  possible++;  }  }  puzzle.undo\_move();  puzzle.move\_down();  if (!puzzle.isSame(&parent->puzzle)) {  if (puzzle.didMove == true) {  dir[possible] = 'd';  possible++;  }  }  puzzle.undo\_move();  }  return possible;  }  };  class BestNode { // Best-first-search에 사용할 Node 클래스 정의  public:  Puzzle puzzle;  int depth;  BestNode\* parent;  BestNode\*\* next;  bool mark;  int f, g, h; // f(n), g(n), h(n) 을 저장하는 변수  public:  BestNode() {}  BestNode(Puzzle\* p, int depth) {  h = 0;  for (int i = 0; i < 3; i++) {  for (int j = 0; j < 3; j++) {  puzzle.info[i][j] = p->info[i][j];  if (!(i == 1 && j == 1)) {  if (puzzle.info[i][j] != solution[i][j]) h++; // h(n) == 틀려있는 타일 개수  }  }  }  puzzle.didMove = p->didMove;  puzzle.empty\_idx = p->empty\_idx;  puzzle.preDir = p->preDir;  parent = NULL;  next = NULL;  this->depth = depth;  g = depth; // g(n) == depth  f = g + h; // f(n) = g(n) + h(n)  mark = false;  }  int getPossibleMove(char dir[4]) {  int possible = 0;  if (parent == NULL) {  puzzle.move\_left();  if (puzzle.didMove == true) {  dir[possible] = 'l';  possible++;  }  puzzle.undo\_move();  puzzle.move\_up();  if (puzzle.didMove == true) {  dir[possible] = 'u';  possible++;  }  puzzle.undo\_move();  puzzle.move\_right();  if (puzzle.didMove == true) {  dir[possible] = 'r';  possible++;  }  puzzle.undo\_move();  puzzle.move\_down();  if (puzzle.didMove == true) {  dir[possible] = 'd';  possible++;  }  puzzle.undo\_move();  }  else {  puzzle.move\_left();  if (!puzzle.isSame(&parent->puzzle)) {  if (puzzle.didMove == true) {  dir[possible] = 'l';  possible++;  }  }  puzzle.undo\_move();  puzzle.move\_up();  if (!puzzle.isSame(&parent->puzzle)) {  if (puzzle.didMove == true) {  dir[possible] = 'u';  possible++;  }  }  puzzle.undo\_move();  puzzle.move\_right();  if (!puzzle.isSame(&parent->puzzle)) {  if (puzzle.didMove == true) {  dir[possible] = 'r';  possible++;  }  }  puzzle.undo\_move();  puzzle.move\_down();  if (!puzzle.isSame(&parent->puzzle)) {  if (puzzle.didMove == true) {  dir[possible] = 'd';  possible++;  }  }  puzzle.undo\_move();  }  return possible;  }  }; |

**3. 실행결과**

****

****

****

****

**4. 랜덤 실행결과** (Depth-first bound는25)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 초기  상태 | 목적  상태 | OpenList의 누적 노드수 | | | CloseList의 누적 노드수 | | | Solution의 Path Length | | |
| Breadth | Depth | Best | Breadth | Depth | Best | Breadth | Depth | Best |
| 문제1 | 365  718  402 | 123  804  765 | 465758 | 410059 | 14553 | 263165 | 410049 | 8309 | 21 | 23 | 21 |
| 문제2 | 345  268  710 | “ | 109605 | 935687 | 2798 | 64868 | 935673 | 1613 | 18 | 24 | 18 |
| 문제3 | 156  438  207 | “ | 2117901 | 2282535 | 46701 | 1270544 | 2282526 | 26611 | 23 | 25 | 23 |
| 문제4 | 450  786  312 | “ | 812056 | 767082 | 20538 | 470739 | 767074 | 11774 | 22 | 24 | 22 |
| 문제5 | 723  415  068 | “ | 38816 | 129374 | 1209 | 23321 | 129365 | 685 | 16 | 24 | 16 |
| 문제6 | 320  745  681 | “ | 279758 | 1224075 | 8889 | 162953 | 1224066 | 5120 | 20 | 22 | 20 |
| 문제7 | 364  275  081 | “ | 829661 | 648294 | 25687 | 482208 | 648285 | 14836 | 22 | 24 | 22 |
| 문제8 | 564  380  172 | “ | 1471682 | 157783 | 44324 | 840203 | 157769 | 25617 | 23 | 25 | 23 |
| 문제9 | 482  376  501 | “ | 1476497 | 733768 | 46612 | 841674 | 733757 | 26922 | 23 | 25 | 23 |
| 문제10 | 846  721  350 | “ | 673114 | 271271 | 27058 | 378477 | 271259 | 15538 | 22 | 24 | 22 |
| 평균 | - | - | 827485 | 755993 | 23837 | 479816 | 755983 | 13703 | 21 | 24 | 21 |