인공지능

과제 3 CSP

백은옥 교수님  
전민철, 조현진, 최승혁 조교님

2013011424  
성예닮  
컴퓨터공학부

1. 환경

운영체제: Windows 10  
언어: Java 8  
개발 IDE: IntelliJ IDEA

2. 자료구조 및 알고리즘

Standard CSP:  
board[i] = j: i행, j열에 여왕이 놓여있다.

ForwardCheckingCSP, ArcConsistencyCSP:  
board[i][j] = 1, i행, j열에 여왕이 놓여있다.  
board[i][j]의 값이 -1인 경우 ForwardChecking 혹은 ConsistencyCheck에 의해 지워진 자리이다.  
board[i][j[의 값이 0인 경우, 여왕을 배치할 수 있는 자리이다.

StandardCSP:  
여왕을 배치할 때 마다 Promissing을 호출한다.  
제약조건을 만족한다면 다음 행에 여왕 배치를 준비하고,  
만족하지 못한다면 현재 행의 다음열에 여왕을 배치하고 제약 조건을 검사한다.

Promising:  
제약조건을 만족하는지 검사한다.

ForwardCheckCSP:  
여왕을 배치할 때 마다 ForwardChecking을 통해 다른 행에 유망하지 않은 자리들을 제거한다.  
ForwardChecking을 어기지 않는 위치에 배치한 후, Promissing을 호출하여 제약 조건을 만족하는지 검사한다.  
제약조건을 만족한다면 다음 행의 유망한 자리에 여왕을 배치하고,  
만족하지 못한다면 현재 행의 다음 유망한 자리에 여왕을 배치한다.

ForwardChecking:  
배치한 여왕의 행과 열을 기준으로 다른 행에 대해 제약조건을 확인한다.  
유망하지 않은 자리라면 -1을 더 이상 퀸을 배치할 수 없도록 미리 제거한다.

ArcConsistencyCSP:  
여왕을 배치할 때 마다 ForwardChecking을 통해 다른 행에 유망하지 않은 자리들을 제거한다.  
다음으로 모든 행에 대해 Arc Consistency를 검사한다.  
현재 상태가 모든 Arc에 대해 Consistency를 만족한다면. 현재 상태에 여왕을 배치하고,  
다음 행에 여왕을 배치할 준비를 한다.  
하나의 행이라도 모두 유망하지 않다면 이전 상태를 기억해두었다가 되돌아간다.

ConsistencyCheck:  
모든 행에 대해 Arc Consistency를 검사한다.

3. 1-2분 이내에 측정 가능한 최대 N과 측정 시간에 대한 분석

N == 29 (3회 평균)  
Standard CSP: 11.619  
CSP with forward checking: 5.767  
CSP with arc consistency: 4.926

N == 30 (3회 평균)  
Standard CSP: 384.39  
CSP with forward checking: 184.632  
CSP with arc consistency: 144.769

기본적으로 N이 29인 경우까지 길지 않은 시간 내에 정답을 찾아낼 수 있다.  
하지만 N이 29 미만이더라도 위쪽의 여왕의 위치가(첫 행에 가까울수록) 뒤쪽인 경우는(마지막 열에 가까울 경우) 시간이 좀 더 오래 걸린다.  
이번 과제의 작성한 코드에서는 재귀 구현의 편의를 위해 배열 복사를 자주 하게 되는데, C계열 언어처럼 Garbage collector를 직접 구현하거나, 메모리 할당 해제를 직접 할 수 없어 N이 증가할수록 더 많은 메모리를 점유하게 되고, 이 부분에서 시간이 더 오래 걸릴 수 있다..

**4 코드 및 상세 설명**

**대부분의 경우 주석에 많은 설명을 포함하고 있어 주석을 포함한 코드를 복사하였습니다.**

**Standard CSP**

*/\*\*  
 \* 인자로 받은 정수형 배열에 담긴 퀸들의 위치를 가지고,  
 \* 현재 퀸들의 배치가 적합한지 확인한다.  
 \** ***@param*** *board 검사할 체스판의 상태이다.  
 \** ***@param*** *level 몇 번째 레벨까지 검사할지를 정해준다.  
 \** ***@return*** *유망하다면 true, 그렇지 않다면 false를 반환한다.  
 \*/*static boolean standardPromising(int[] board, int level) {  
 for (int i = 0; i < level + 1; i++) {  
 for (int j = i + 1; j < level + 1; j++) {  
 // 같은 열 검사  
 if (board[i] == board[j]) {  
 return false;  
 } else if (j - i == Math.*abs*(board[j] - board[i])) {  
 // 대각선 검사  
 return false;  
 }  
 }  
 }  
 return true;  
}

*/\*\*  
 \* Standard CSP를 이용하여 N-여왕 문제를 해결한다.  
 \** ***@param*** *board doStandardCSP를 호출할 당시의 체스판의 상태이다.  
 \** ***@param*** *level 재귀 호출의 깊이, 몇번째 행인지를 나타낸다.  
 \** ***@return*** *level에서의 유망한 board의 상태를 반환한다.  
 \*/*public static int[] doStandardCSP(int[] board, int level) {  
 if (level >= board.length) {  
 return board;  
 }  
  
 int[] returnBoard = new int[board.length];  
  
 for (int i = 0; i < board.length; i++) {  
 board[level] = i;  
 // 깊이를 하나 증가시켜 재귀 호출  
 if (*standardPromising*(board, level)) {  
 returnBoard = *doStandardCSP*(board, level + 1);  
  
 // 정답을 찾는다면, 해당 정답을 반환한다.  
 if (null != returnBoard) {  
 return returnBoard;  
 }  
 }  
 }  
 // 정답이 아닌 경우 null을 반환한다.  
 return null;  
}

CSP with Forward Checking

*/\*\*  
 \* 인자로 받은 정수형 배열에 담긴 퀸들의 위치를 가지고,  
 \* 현재 퀸들의 배치가 적합한지 확인한다.  
 \** ***@param*** *board 검사할 체스판의 상태이다.  
 \* -1: 유망하지 않은 위치  
 \* 0: 유망한 위치  
 \* 1: 이미 퀸을 배정한 위치  
 \** ***@param*** *row 몇 번째 행을 검사할지를 정해준다.  
 \** ***@return*** *유망하다면 true, 그렇지 않다면 false  
 \*/*public static boolean forwardCheckPromising(int[][] board, int row) {  
 for (int i = 0; i < board.length; i++) {  
 if (0 == board[row][i]) {  
 return true;  
 }  
 }  
 return false;  
}

*/\*\*  
 \* 인자로 받은 board의 row, column의 위치를 기준으로 forward checking을 한다.  
 \** ***@param*** *board forward checking을 할 체스판의 상태  
 \** ***@param*** *row forward checking을 할 열의 위치  
 \** ***@param*** *column forward checking을 할 열의 위치  
 \*/*public static void forwardCheck(int[][] board, int row, int column) {  
 for (int i = 0; i < board.length; i++) {  
 if (0 == board[row][i]) {  
 board[row][i] = -1;  
 }  
 }  
  
 for (int i = 0; i < board.length; i++) {  
 if (0 == board[i][column]) {  
 board[i][column] = -1;  
 }  
 }  
  
 // 입력된 행보다 위의 행들에 대해서 forward checking  
 for (int i = 0; i < row; i++) {  
 if (-1 < (column - row + i)  
 && board.length > (column - row + i)  
 && board[i][column - row + i] != 1)  
 {  
 board[i][column - row + i] = -1;  
 }  
  
 if (-1 < (column + row - i)  
 && board.length > (column + row - i)  
 && board[i][column + row - i] != 1)  
 {  
 board[i][column + row - i] = -1;  
 }  
 }  
  
 // 입력된 행보다 아래의 행들에 대해서 forward checking  
 for (int i = row + 1; i < board.length; i++) {  
 if (-1 < (column - row + i - 1)  
 && board.length > (column - row + i)  
 && board[i][column - row + i] != 1)  
 {  
 board[i][column - row + i] = -1;  
 }  
  
 if (-1 < (column + row - i - 1)  
 && board.length > (column + row - i)  
 && board[i][column + row - i] != 1)  
 {  
 board[i][column + row - i] = -1;  
 }  
 }  
}

*/\*\*  
 \** ***@param*** *board 현재 board의 상태  
 \** ***@param*** *level forward checking을 이용한 CSP를 수행할 단계, 행의 위치이다.  
 \** ***@return*** *level에서의 유망한 board를 반환한다. 그렇지 않다면 null을 반환한다.  
 \*/*public static int[][] doForwardCheckCSP(int[][] board, int level) {  
 if (level >= board.length) {  
 return board;  
 }  
  
 if (!*forwardCheckPromising*(board, level)) {  
 return null;  
 }  
  
 int[][] newBoard = new int[board.length][board.length];  
 *copyArray*(board, newBoard);  
  
 for (int i = 0; i < board.length; i++) {  
 if (0 == newBoard[level][i]) {  
 newBoard[level][i] = 1;  
 *forwardCheck*(newBoard, level, i);  
 newBoard = *doForwardCheckCSP*(newBoard, level + 1);  
  
 if (null != newBoard) {  
 return newBoard;  
 }  
 newBoard = new int[board.length][board.length];  
 *copyArray*(board, newBoard);  
 newBoard[level][i] = -1;  
 }  
 }  
 return null;  
}

CSP with arc consistency

*/\*\*  
 \* forward checking 이후 모든 간선에 대해 consistency를 확인한다.  
 \* N-여왕 문제의 경우, 모든 상태는 서로 연결되어 있기 때문에  
 \* level 이후의 consistency만 고려하면 된다.  
 \** ***@param*** *board  
 \** ***@param*** *level  
 \** ***@return*** *\*/*public static int[][] doArcConsistencyCSP(int[][] board, int level) {  
 if (level >= board.length) {  
 return board;  
 }  
  
 int[][] newBoard = new int[board.length][board.length];  
 *copyArray*(board, newBoard);  
  
 for (int i = 0; i < newBoard.length; i++) {  
 if (-1 == newBoard[level][i]) {  
 continue;  
 }  
 // [level][i]의 consistency를 확인하기 위해  
 // forward checking을 선행한다.  
 *forwardCheck*(newBoard, level, i);  
  
 if (*consistencyCheck*(newBoard, level + 1)) {  
 // [level][i]가 유망하고,  
 // 모든 행에 대해 Consistency를 만족한다면 여왕을 배치한다.  
 newBoard[level][i] = 1;  
 newBoard = *doArcConsistencyCSP*(newBoard, level + 1);  
  
 // 유망한 상태라면 해당 체스판을 반환한다  
 if (null != newBoard) {  
 return newBoard;  
 } else {  
 // 그렇지 않다면 이전 체스판으로 복귀한다.  
 newBoard = new int[board.length][board.length];  
 *copyArray*(board, newBoard);  
 }  
 } else {  
 *copyArray*(board, newBoard);  
 newBoard[level][i] = -1;  
 }  
 }  
 return null;  
}

*/\*\*  
 \* level 이전의 행은 이미 consistency 하다. 따라서 이후의 level만 확인한다.  
 \** ***@param*** *level  
 \** ***@param*** *board  
 \** ***@return*** *level 이후부터의 모든 간선이 Consistency하면 true를 반환한다.  
 \*/*public static boolean consistencyCheck(int[][] board, int level) {  
 for (int i = level; i < board.length; i++) {  
 boolean isConsistency = false;  
 for (int j = 0; j < board.length; j++) {  
 if (0 == board[level][j]) {  
 isConsistency = isConsistency | true;  
 break;  
 }  
 }  
  
 if (!isConsistency) {  
 return false;  
 }  
 }  
 return true;  
}

**5 소스코드를 컴파일하는 방법**

**1) java 파일을 다운로드 받아 실행한다.  
컴퓨터의 환경이 encoding=UTF-8, line separator=UNIX(LF)인 경우 소스 코드를 올바르게 컴파일 하고 출력 결과를 올바르게 확인할 수 있다.**

**2) IntelliJ IDEA 프로젝트이기 때문에, assignmen2 폴더를 다운로드하여 IntelliJ에서 Import 한다.  
File -> New -> Project From Existing sources -> 다운로드 받은 경로 입력 후 OK  
-> Create project from existing sources -> Project name 입력 후 모두 Overwrite**

**6 기타 사항**

1) 출력 파일을 생성할 절대 경로에 스페이스가 포함되어 있다면 전체 경로를 “”(큰따옴표)로 감싸주어야 한다.  
예시) java -jar 2013011424.jar 4 "C:\Users\Yedarm Seong”

2) 출력 파일의 개행 문자가 CRLF가 아닌 LF기 때문에, Windows 메모장에서는 개행이 되지 않는다. UNIX 개행문자를 지원하는 에디터(Sublime text, atom 등)으로 출력 파일을 확인해야 한다.

3) 체스판의 한 행의 크기(여왕의 개수)가 3이하인 경우 정답이 없기 때문에 알고리즘 수행 후 No solution을 출력한다.

4) JVM의 Heap영역 메모리 할당을 크게 하면, 수행 속도가 조금 개선된다.