Vidrighin George-Bogdan

CEN 2.3B

2nd year

**Heuristic approach (hill climbing) of the n archers on a k x k grid problem**

It is given a k x k grid that is configured with a pattern of walls. You are required to place n archers on this grid such that they cannot shoot each other. An archer can shoot up, down, left, right and also diagonally and its shoot can reach at most w locations in all directions, up to the grid edges.

For this problem the obvious choice was a backtracking approach but given the requirement to use a heuristic search algorithm, my choice was hill climbing.

The first step was to build a function that configured a board and the state randomly (**configureRandomly ( int board[][N], int\* state)** ). The N is defined as the size of the board (grid).

i.e: N = 8 , the board is 8x8

This function iterates through the column indices, gets a random row index and places an archer on the obtained place.

srand(time(0));  
  
 for (int i = 0; i < N; i++) {  
  
 state[i] = rand() % N;  
  
 board[state[i]][i] = 1;  
 }

Another function is **printBoard(int board[][N])** which is basic and self explanatory. It iterates through the indices of the board and prints the elements.

for (int i = 0; i < N; i++) {  
 cout << " ";  
 for (int j = 0; j < N; j++) {  
 cout << board[i][j] << " ";  
 }  
 cout <<endl;  
}

Like the printBoard function, we also have a **printState(int\* state)** function that iterates through the state indices and prints the state of the board.

for (int i = 0; i < N; i++) {  
 cout << " " << state[i] << " ";  
}  
cout << endl;

**compareStates(int\* state1, int\* state2)** is a function that compares two states and returns true if they are equal and false otherwise. It compares the states during an iteration of the state arrays.

for (int i = 0; i < N; i++) {  
 if (state1[i] != state2[i]) {  
 return false;  
 }  
}  
return true;

**fill(int board[][N], int value)** when using this function, using the parameter value, it fills the board with values. We iterate through the board and fill each element of the grid with a value.

for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
 board[i][j] = value;  
 }  
}

The **calculateObjective(int board[][N], int\* state)** function is a little more complicated, but basically it calculates the objective value of the state ( archers attacking eachother).

The simple idea of this function is : for each archer in a column, we check for other archers in the line of our current archer and if found, the variable attacking is incremented.

At each column i, the archer is placed at row state[i], and while we are iterating we are checking on various locations of the board (stated in the problem definition) if there are other archers placed such that the problem cannot work correctly.

for (int i = 0; i < N; i++) {

row = state[i], col = i - 1;

# Here we are checking to the left of the same row #

while (col >= 0 && board[row][col] != 1) {  
 col--;  
}  
if (col >= 0 && board[row][col] == 1) {  
 attacking++;  
}

## Here we are checking to the right of the same row ##

row = state[i], col = i + 1;  
while (col < N && board[row][col] != 1) {  
 col++;  
}  
if (col < N && board[row][col] == 1) {  
 attacking++;  
}

### Here we are checking diagonally to the left up ###

row = state[i] - 1, col = i - 1;  
while (col >= 0 && row >= 0 && board[row][col] != 1) {  
 col--;  
 row--;  
}  
if (col >= 0 && row >= 0 && board[row][col] == 1) {  
 attacking++;  
}

#### Here we are checking diagonally to the right down ####

row = state[i] + 1, col = i + 1;  
while (col < N && row < N && board[row][col] != 1) {  
 col++;  
 row++;  
}  
if (col < N && row < N && board[row][col] == 1) {  
 attacking++;  
}

##### Here we are checking diagonally to the left down #####

row = state[i] + 1, col = i - 1;  
while (col >= 0 && row < N && board[row][col] != 1) {  
 col--;  
 row++;  
}  
if (col >= 0 && row < N && board[row][col] == 1) {  
 attacking++;  
}

###### Here we are checking diagonally to the right up ######

row = state[i] - 1, col = i + 1;  
 while (col < N && row >= 0 && board[row][col] != 1) {  
 col++;  
 row--;  
 }  
 if (col < N && row >= 0 && board[row][col] == 1) {  
 attacking++;  
 }  
}

return (int)(attacking / 2);

Following up we have a function that generates a board configuration using a state **generateBoard(int board[][N], int\* state)**, we start by filling the board using the fill function with the value 0, and then iterating through the board, we give the value 1 to board[state[i][i]][i].

fill(board, 0);  
for (int i = 0; i < N; i++) {  
 board[state[i]][i] = 1;  
}

**copyState(int\* state1, int\* state2)** copies the contents of state2 to state1 when iterating through the state.

The **getNeighbour(int board[][N], int\* state)** is used to get the neighbour of the current state. Firstly we declare and initialize the optimal board and state with the current ones.

int opBoard[N][N];  
int opState[N];  
  
# Here we just copy the state and generate the board using oprimal state and board #

copyState(opState, state);  
generateBoard(opBoard, opState);

## Then we initialize the optimal objective value and declare and initialize the temporary board and state and generate the neighbour board##

int opObjective = calculateObjective(opBoard, opState);

int NeighbourBoard[N][N];  
int NeighbourState[N];  
  
copyState(NeighbourState, state);  
generateBoard(NeighbourBoard, NeighbourState);

### After that we iterate through all possible neighbours, initializing the temporary neighbour with the current neighbour and calculate the objective value of the neighbour in the temp variable ###

for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
  
 if (j != state[i]) {

NeighbourState[i] = j;  
 NeighbourBoard[NeighbourState[i]][i] = 1;  
 NeighbourBoard[state[i]][i] = 0;

int temp = calculateObjective(NeighbourBoard, NeighbourState);

#### We compare temporary and optimal neighbour objectives and update accordingly ####

if (temp <= opObjective) {  
 opObjective = temp;  
 copyState(opState,NeighbourState);  
 generateBoard(opBoard,opState);  
 }

##### Then we are reseting to the original configuration for the next iteration #####

NeighbourBoard[NeighbourState[i]][i] = 0;  
 NeighbourState[i] = state[i];  
 NeighbourBoard[state[i]][i] = 1;

}

}

}

###### And finally copying the optimal board and state to the current board and state ######

copyState(state, opState);  
fill(board, 0);  
generateBoard(board, state);

The **hillClimbing(int board[][N], int\* state)** function is the main one of this problem and we start by declaring and initializing the neighbour with the current board and state.

int neighbourBoard[N][N] = {};  
int neighbourState[N];  
  
copyState(neighbourState, state);  
generateBoard(neighbourBoard, neighbourState);

do {

# We are copying the neighbour board and state to the current board and state and then getting the optimal neighbour #

copyState(state, neighbourState);  
generateBoard(board, state);

getNeighbour(neighbourBoard, neighbourState);

## We then compare states and if neighbour and current are equal, no optimal neighbour exists and we just print the board and break ##

if (compareStates(state, neighbourState)){

printBoard(board);  
 break;  
}

### If that’s not the case but their objectives are equal, we are approaching a shoulder of a local optimum and if that is the case we are going to jump to a random neighbour ###

else if (calculateObjective(board, state) == calculateObjective(neighbourBoard, neighbourState)){

neighbourState[rand() % N] = rand() % N;  
 generateBoard(neighbourBoard, neighbourState);  
}

}while(true);

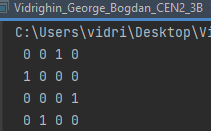
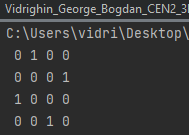
And finally in the main function all we do is initialize the state and the board, then randomly configure the board and call the hillClimbing function .

We change the size of the board by changing the value of N from the definition. And because of using random features, everytime we press run we most likely will get another result, well that also depends on the number of possible outcomes that is dictated by the size of the board.

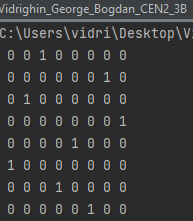
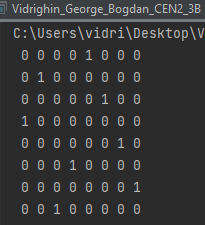
#define N 8

I am gonna do some tests on grids of different sizes and I will attach screenshots of these tests.

#define N 4

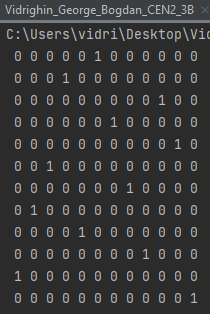
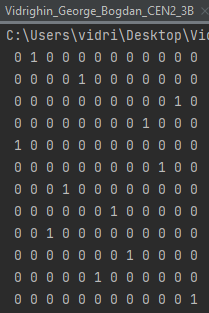


#define N 8

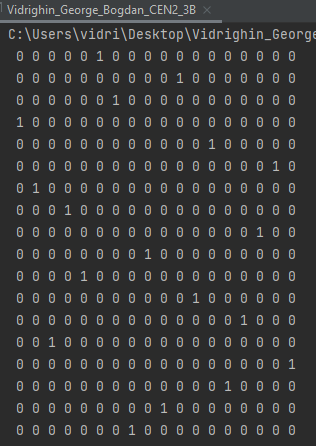
A screenshot of a computer

Description automatically generated with medium confidence

#define N 12



#define N 18



#define N 24

