```
11          for(int m = 0; m < N; m++) {
12              if(((k + m) == (i + j) || (k - m) == (i - j))) {
13                  if(board[k][m] == 1) {
14                      return true; // being attacked
15                  }
16              }
17          }
18      }
19
20      return false; // not being attacked
21  }
```

Breaking this down a little more we have the diagonal check. Any square (k, m) will be diagonal to the square (i, j) if $k + m$ is equal to $i + j$ or $k - m$ is equal to $i - j$.

5.4 Minimax

Minimax is a way to make a decision such that you minimise the possible loss for a worst case scenario. It was originally formulated for two player, zero sum game theory, but has been extended to many more complex games and general decision making in the presence of uncertainty. We'll look at the idea in relation to two player turned based board games, such as chess, go, checkers, connect four, and any other game where players take turns to make a move.

It is a relatively simple algorithm considering the value gained by using it (and optimsations of it) in such games. It requires a function of the game called an *evaluation function*, which is an approximation for how the game is going. For example, for the game of chess, a basic evaluation function might be how many pieces each players has left on the board, and if you have more than your opponent, then consider yourself winning. This isn't too far away from how most new players actually evaluate the game either! The top chess programs use significantly more complex evaluation functions than these though.

Generally when trying to decide on a move in a game as complex as chess, it is important to limit the amount of moves we look ahead, otherwise we will end up in a computational nightmare. The branching factor of various games, and how many nodes need to be searched at the given depth is given in the following table.

Game	Tic-tac-toe	Connect Four	Checkers	Chess	Go
Branching factor	4	4	2.8	35	250
Depth 1	4	4	2.8	35	250
Depth 2	16	16	7.84	1225	62500
Depth 3	64	64	21.95	42875	15625000
Depth 4	256	256	61.4656	1500625	3906250000
Depth 5	1024	1024	172.104	52521875	976562500000
Depth 6	4096	4096	481.89	1838265625	244140625000000

So even looking just six turns ahead in games like chess and go is impractical without a lot of optimisations to the main minimax algorithm, but most of the top chess engines (as of May 2019) still use something derived from it, although there are challenges to the algorithm for the first time in the history of computers playing chess. The basic minimax algorithm is what we are going to be looking at though.

So after getting our evaluation function, and limiting the depth of our search, we can create our `minimax` algorithm. It takes in three parameters, `node` which is the node we are currently exploring in the tree, `depth` which is how much deeper we should search, and `maximisingPlayer` which is true if the player is trying to maximise, and false if it is the opponent. The evaluation function is

always done from the first players perspective, so a positive value means the first player is winning, and a negative value means the second player is winning. We will call `evaluationFunction(Node)` but not actually define it, we will assume it satisfies the properties of an evaluation function for the game though.

```
int minimax(Node node, int depth, boolean maximisingPlayer) {
   if(depth == 0) {
      return evaluationFunction(node);
   }

   if(maximisingPlayer) {
      // set the current evaluation to be as small as possible if we wish to
          maximise it
      int value = Integer.MIN_VALUE;
      for(int i = 0; i < node.children.length; i++) {
         // if we can do better, then set value to the better evaluation
         value = Math.max(value, minimax(node.children[i], depth - 1; false));
      }
      return value;
   }
   else { // we are trying to minimise
      // set the current evaluation to be as large as possible if we wish to
          minimise it
      int value = Integer.MAX_VALUE;
      for(int i = 0; i < node.children.length; i++) {
         value = Math.min(value, minimax(node.children[i], depth - 1; true));
      }
      return value;
   }
}
```

We also need a helper algorithm for our initial call to `minimax`, much like most of the other recursive algorithms we have looked at. We wish to track which move is best, so we do that in the initial algorithm as well.

```
int minimaxStart(Node node, int depth, boolean maximisingPlayer) {
   int bestIndex = 0;
   if(maximisingPlayer) {
      int bestValue = Integer.MIN_VALUE;
      for(int i = 0; i < node.children.length; i++) {
         int tmpValue = minimax(node.children[i], depth, false);
         if(tmpValue > bestValue) { // we're trying to maximise, so if we get a
             bigger value, update
            bestValue = tmpValue;
            bestIndex = i;
         }
      }
   }
   else { // minimising player
      int bestValue = Integer.MAX_VALUE;
      for(int i = 0; i < node.children.length; i++) {
         int tmpValue = minimax(node.children[i], depth, true);
         if(tmpValue < bestValue) { // we're trying to minimise, so if we get a
             smaller value, update
            bestValue = tmpValue;
            bestIndex = i;
         }
      }
   }
```

```
23
24      return bestIndex;
25  }
```

This algorithm returns the best move from the list of possible moves from the current game position.

For a visual demonstration of what is happening, rather than following through the code, this diagram shows the choices being made at each step in a tree of depth 4. The player whose turn it is is trying to maximise their evaluation function, so they make the best move for them, based on what the other player is going to do, and so on, down the tree. The values of $+\infty$ and $-\infty$ are used where typically in actual implementation the largest value storable would be used.

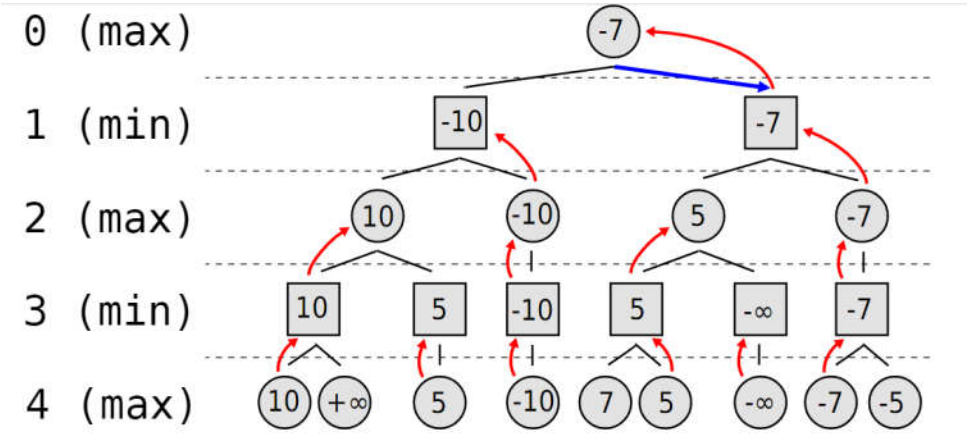

Figure 6: A visual representation of the `minimax` algorithm.

The first major optimisation the `minimax` algorithm is called `alpha-beta pruning`, and drastically limits the number of nodes that need to be searched in any given game tree. It might not be obvious, but we are search the tree in a depth first manner. Another optimisation technique used by top chess engines is called `iterative deepening`, which, together with a clever evaluation function, can further limit the number of nodes in the game tree that need to be searched. It is to the point that as of December 2016, the best chess engines all evaluate the chess game tree with an effective branching factor of 1.52 (Stockfish 8, latest release is Stockfish 10, so it is probably even better).