1 The Algorithm for Playing the Board Game

To compute scores for the board configurations, your program will use a recursive algorithm that will repeatedly simulate a move from the computer followed by a move from the human, until an outcome for the game has been decided. This recursive algorithm will implicitly create a tree formed by all the moves that the players can make starting at the current board configuration. This tree is called a *game tree*. An example of a game tree is shown in Figure 1.

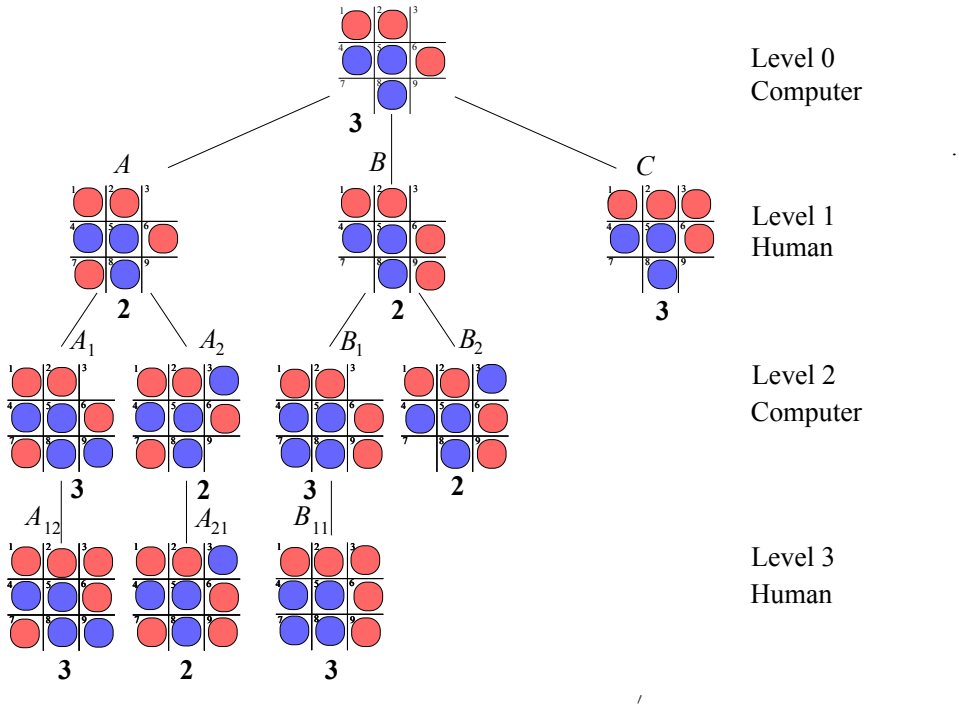

Figure 1: A game tree.

Assume that after 6 moves the game board is as the one shown at the top of Figure 1 and suppose that it is the computer's turn to move. The algorithm for computing scores will first try all possible moves that the computer can make: A, B, and C. For each one of them, the algorithm will then consider all possible moves by the human player: A_1, A_2, B_1, and B_2. Then, all possible responses by the computer are attempted, and so on. Note that a game tree might have many levels as players might need many moves to decide the outcome of the game. In order for your program to be responsive, we will limit the maximum number of levels of the game tree that the program will consider; this maximum number of levels is equal to the value stored in the fourth line of the input file. In Figure 1 we limited the number of levels to 3.

In Figure 1 each level of the tree is labelled by the player whose turn is next. So levels 0 and 2 are labelled "computer" and the other two levels are labelled "human". After reaching configurations A_{12}, A_{21}, and B_{11} in level 3, the algorithm computes a score for each one of them depending on whether the computer wins, the human wins, the game is a draw, or no decision has been reached. Scores are the numbers in bold below the configurations. The scores are propagated upwards as follows:

- For a configuration b on a level labelled "computer", the highest score of the configurations in the next level is selected as the score for b. This is because we have chosen the highest score (`COMPUTER_WINS` $= 3$) for a play that makes the computer win.
- For a configuration b on a level labelled "human", the score of b is equal to the minimum score

in the next levels, because the lowest score (`HUMAN_WINS`=0) has been given to any play that guarantees a win by the human player.

Since for the board at the top of Figure 1, putting an orange tile in position 3 yields the configuration with the highest score, then the computer will choose to play in position 3. We give below the algorithm for computing scores and for selecting the best available move. The algorithm is given in Java, but we have omitted variable declarations and some initialization steps. A full version of the algorithm can be found inside class `PlayGame.java`, which can be downloaded from the course's website.

```
private PosPlay computerPlay(char symbol, int highestScore, int lowestScore, int level) {
    if (level == 0) configurations = t.createDictionary();
    if (symbol == COMPUTER) opponent = HUMAN; else opponent = COMPUTER;

    for(int row = 0; row < board_rows; row++)
      for(int column = 0; column < board_cols; column++) {
        if(t.squareIsEmpty(row,column)) { // Empty position found
          t.storePlay(row,column,symbol);
          if (t.wins(symbol)||t.isDraw(symbol)||(level == max_level))
            reply = new PosPlay(t.evalBoard(symbol),row,column,level);
          else {
              lookupVal = t.repeatedConfig(configurations);
(*)           if (lookupVal != null)
              reply = new PosPlay(lookupVal.getScore(),row,column,lookupVal.getLevel());
            }
            else {
              reply = computerPlay(opponent, highestScore, lowestScore, level + 1);
              if (t.repeatedConfig(configurations) == null)
                t.insertConfig(configurations,reply.getScore(),reply.getLevel());
            }
          }

          t.storePlay(row,column,' ');
          if((symbol == COMPUTER && reply.getScore() > value) || // A better play was found
            (symbol == HUMAN && reply.getScore() < value)) {
            bestRow = row;
            bestColumn = column;
            value = reply.getScore();
            bestLevel = reply.getLevel();
            if (symbol == COMPUTER && value > highestScore) highestScore = value;
            else if (symbol == HUMAN && value < lowestScore) lowestScore = value;
            if (highestScore >= lowestScore) /* alpha/beta cut */
              return new PosPlay(value, bestRow, bestColumn,bestLevel);
          }
        }
    return new PosPlay(value, bestRow, bestColumn,bestLevel);
}
```

The fist parameter of the algorithm is the symbol (either 'b' or 'o') of the player whose turn is next. The second and third parameters are the highest and lowest scores for the board positions that have been examined so far. The last parameter is used to bound the maximum number of levels of the game tree that the algorithm will consider.

1.1 Speeding-up the Algorithm with a Dictionary

The above algorithm includes several tests that allow it to reduce the number of configurations that need to be examined in the game tree. For this assignment, the most important test used to speed-up the program is the one marked (*). Every time that the score of a board configuration is computed, the configuration and its score are stored in a dictionary, that you will implement using a hash table. Then, when algorithm `computerPlay` is exploring the game tree trying to determine the computer's best move, before it expands a configuration B it will look it up in the dictionary. If B is in the dictionary then its score is simply extracted from the dictionary instead of exploring the part of the game tree below B.

For example, consider the game tree in Figure 3. The algorithm examines first the left branch of the game tree, including configuration D and all the configurations that appear below it. After exploring the configurations below D, the algorithm computes the score for D and then in stores D and its score in the dictionary. When later the algorithm explores the right branch of the game tree, configuration D will be found again, but this time its score is simply obtained from the dictionary instead of exploring all configurations below D, thus reducing the running time of the algorithm.

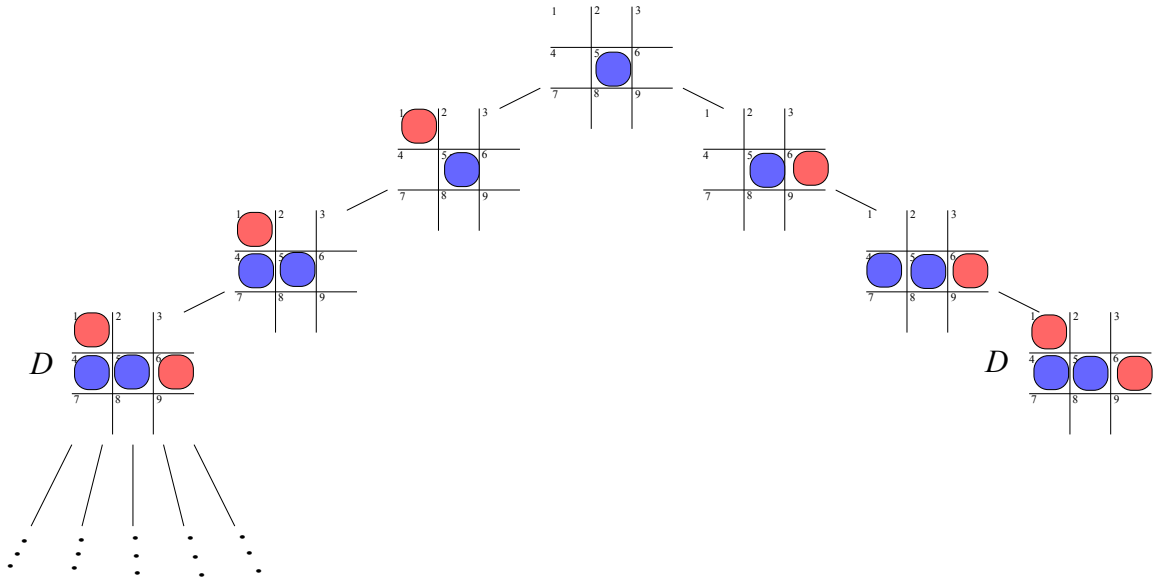

Figure 2: Detecting repeated configurations.