# INTRODUCTION TO ARTIFICIAL INTELLIGENCE PROJECT

19127057 – Trần Vĩnh Phát

19127108 – Ngô Phú Chiến

19127120 – Ngô Nhật Du

19CLC9 HCMUS

## 1. THEORY:

### GAME THEORY

**Game theory** is the study of mathematical models of strategic interaction among rational decision-makers.

Originally, it addressed zero-sum games, in which each participant's gains or losses are exactly balanced by those of the other participants.

**Game theory** views any multiagent environment as a game.
The impact of each agent on the others is “significant,” regardless of whether the agents are cooperative or competitive.

- **Game vs. Search**

  **Complexity**: games are too hard to be solved

  **Time limits:** make some decision even when calculating the optimal decision is infeasible

  **Efficiency**: penalize inefficiency severely

- **Game as Search**

  **$S_0$** – Initial State: How the game set up from the beginning

  **Player(s)**: MIN / MAX

  **Action(s)** – Successor Function: A list of pairs (move, state) of legal moves

  **Result(s, a)** – Transition Model: Result move **a**, on state **s**

  **Test(s)** – Terminal: Has the game finished?

  **Utility(s, p)** – Utility Function: A numerical value of **a** terminal state **s** for a player **p**

### ADVERSARIAL SEARCH

- Known as games: covers competitive environments in which the agents’ goals are in conflict
- Is search when there is an "enemy" or "opponent" changing the state of the problem every step in a direction you do not want.
- You change state, but then you don't control the next state.
  Opponent will change the next state in a way: unpredictable or  to you
- You only get to change (say) every alternate state.

#### 1. Minimax

The **Minimax** algorithm tries to predict the opponent's behaviour. It predicts the opponent will take the *worst* action from our viewpoint. We are MAX - trying to maximise our score / move to best state. Opponent is MIN - tries to minimise our score / move to worst state for us.

The number of game states is exponential in the tree’s depth → Do not examine every node

**NegaMax:** is a common way of implementing minimax and derived algorithms. Instead of using two separate subroutines for the Min player and the Max player, it passes on the negated score due to following mathematical relation: max(a, b) = -min(-a, -b).

**NegaScout:** This algorithm is an alpha-beta enhancement and improvement of Judea Pearl’s Scout-algorithm, introduced by Alexander Reinefeld in 1983. The improvements rely on a **NegaMax** framework and some fail-soft issues concerning the two last plies, which did not require any re-searches. On Alpha-Beta pruning, **NegaScout** claims that it can accelerate the process by setting [Alpha,Beta] to [Alpha,Alpha-1].

### 2. Alpha-beta pruning

Is a way of pruning the space rather than search the entire space to fixed depth, we can get rid of branches if we know they won't be picked.

Prune away branches that cannot possibly influence the final decision.

Pruning is good:
Calculating the heuristic evaluation of a state takes *time* (consider chess).
It is good if we can avoid evaluating many states.

**Alpha-beta pruning** can decrease the number of nodes the **NegaMax algorithm** evaluates in a search tree in a manner similar with its use with the minimax algorithm.

### 3. Heuristic minimax

Both minimax and alpha-beta pruning search all the way to terminal states.

Heuristic minimax cut off the search earlier with some depth limit using an evaluation function

## 2. APPLICATION:

### A. HOW TO BUILD THE SOURCE CODE:

In this project we use C# 9.0 to have a better performance. It requires .Net 5.0 to be installed and the C# compatible IDE:

- JetBrains Raider 2020.3 or newer.
- Microsoft Visual Studio 2019 16.9 or newer with .Net Desktop Development installed.

Please visit this link to download the latest version of **.Net 5.0 SDK** to build the program: [Download .NET 5.0 (Linux, macOS, and Windows) (microsoft.com)](). Remember to install the execution then restart your computer.

- **JETBRAINS RAIDER** users: please update to the **latest version**

1. Open Raider and drag the "reversi.sln" to the main windows of Raider.
2. Open the project "client-console" and wait for the analyzing process to complete.
3. Choose **Run** menu, click **Run 'client-console'** and you will see the result.

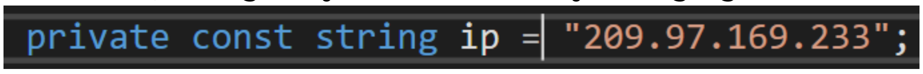
- **MICROSOFT VISUAL STUDIO 2019** users: please update to the **latest version**

1. Make sure that .Net Desktop Development is installed in your VS.
2. Open Visual Studio and open **"reversi.sln".**
3. Choose **Debug** menu → click **Start Debugging** or **F5** and you will see the result.

### CONNECT TO AN ONLINE BOT:

***CHANGE IP:***

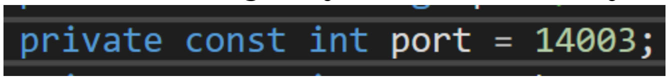
1. In **reversi.sln**, look for **Program.cs** in **client-console**
2. At line 17th, change to your desired **IP** by changing the **red** number inside the **"quotes"**



```
private const string ip = "209.97.169.233";
```

***CHANGE PORT:***

1. In **reversi.sln**, look for **Program.cs** in **client-console**
2. At line 18th, change to your desired **Port** by changing the **yellow** number after the equal sign "**=**"



```
private const int port = 14003;
```

### B. PSEUDOCODE:

In this project, we use both **NegaScout** and **NegaMax** algorithm.

```
int NegaScout(cell, color, depth, alpha, beta){
    if(depth == maxDepth || cannot play anymore)
        return heuristic(cell,color)
    score = -infinity
    oppColor = color == ‘B’ ? ‘W’ : ‘B’
    adaptiveBeta = beta
    moves = cell.GetPossibleMoves(color)
    for each move in moves:
    {
        oldStates = DoMoves(cell,move)
        curr = -NegaScout(cell,oppColor,depth+1,-adaptiveBeta,-max(alpha,score))
        if (cur > score) {
            if(adaptiveBeta == beta || depth >= maxDepth - 2)
                score = curr
            else score = -NegaScout(cell,oppColor,-beta,-curr)
            if (score >= beta){
                UndoMoves(cell,oldStates)
                return heuristic(cell,color)
            }
        }
        UndoMoves(cell,oldStates)
        adaptiveBeta = max(alpha,score) + 1
    }
    return score
}
```

```
int negamax(cell,color,depth,alpha,beta){
    if(depth == 0)
        return Heuristic(cell,color)
    moves = cell.GetPossibleMoves(color)
    for each move in moves:
    {
        oldStates = DoMoves(cell,move)
        curr = -negamax(cell,color == 'B' ? 'W' : 'B',depth-1,-beta,-alpha)
        UndoMoves(oldStates)
        if(curr >= beta)
            return curr
        if(val > alpha)
            alpha=val
    }
    return alpha
}
```

## C. ADVANTAGES AND DISADVANTAGES:

### ADVANTAGES:

- Find the best move in a shorter time in a large space than when using minimax combined with alpha-beta pruning.
- Using both NegaMax and NegaScout for a greater search.
- The majority of moves are made quickly and capture tactical squares resulting in higher odds of winning.
- Our bot has a depth of 10 so it has the intelligence to calculate the opponent's moves. After that, the bot will make the above decisions and win the opponent.

### DISADVANTAGES:

- The heuristic function does not seem to be good.
- There are still existing some moves that take much time to find out because the available move order is not reasonable. This is the downside of NegaScout in our bot.
- Due to the fact that we do not feel like NegaScout is stable enough, we decide to have the support from NegaMax for a better experience (with approximately 70% chances of use for NegaMax and 30% for NegaScout).
- We do not have multithread yet so it is still a bit slow compared to our expectations.
- There are many ways to make the bot smarter, one of them is KB, but we do not have the conditions to put it in yet, so the bot's power is limited under some conditions.

Nowadays in some open-source game engines, people often use NegaScout algorithm or MTD (f) algorithm to search for a short time in a larger space. Along with some more sophisticated heuristic functions to calculate the result of a step more accurately. Using dictionaries of common opening list for the first step to be able to defeat the opponent in the fewest moves, while creating an advantage for itself.