GOBBLET GOBBLERS PROJECT

Artificial intelligence

MOHAMED AHMED SAMIR 1900002
MOHAMED SAEED IBRAHIM 1900084
MARK ASHRAF WILLIAM 1900156
HOSSAM EL-DIN ADEL TALAAT 1900969
MARCO SHERIF MAGDY 1900285
RANA HOSSNY HASSAN 1900266
SARA HOSSNY HASSAN 1901083
KEROLOS SAMEH EL-SHAIP BOLES 1900144

Table of Contents

Links

Game GitHub: [MohammadDallash/Gobblet-Ai-PYgame]

YouTube: (10) YouTube

Player vs AI hard mode video: Gobblet gobblers Player vs AI hard mode - YouTube

Contribution table

Name	ID	Contribution
Mohamed saeed ibrahim	1900084	- GPU palletization with Cuda C++ - UI design of the multiplayer menu - Multiplayer closing connection when a player disconnect - Generation of possible next states from the current sate - Game engine (State abstarct class, State transition, main game class, rendering layers) - GUI menu helpers (auto alignment of icons clicking)
Mark Ashraf William	1900156	- Several Game Logic functionalities like making a move and the logic of moving a piece, game flow and restrictions to make the game comply with rules. - UI elements like drawing inventory, connecting these several front end elements to the logic. - Evaluation Function. - Zobrist board hashing algorithm. - Added Multithreading to the python codebase. - Game sounds. - State Diagram. - Auto C++ file building using batch script and file organization.
Mohamed ahmed samir	1900002	- Checking for a winner and announcing him. - Getting the exact location of any mouse click at any time. - Integration between python and C++ code. - Heuristics choice. - Evaluation function. - Class diagram.

Marco sherif magdy	1900285	- Game Assets design, like game level art, Logo, tiles and sprite sheet, music. - Game States, State transition, screens for each state. - Game flow, like turns switching, handling win/draw screens. - Created menus for different states like winning screen, draw screen, options. - Game options, like sound/music options. - Added functionalities to the AI system, like checking for wins.
Hossam El-Din Adel Talaat	1900969	- Min-Max Algo - Handling AI vs AI animation - Handling AI vs Player animation - Game logic, the validity of moving piece. - Multiplayer on the same LAN - Contribution on Player vs Player online
Sara Hossny hassan	1901083	- working in music player backend - working in Spritesheet backend - Handling drag the pieces to achieve touch-movie rule - highlight the place in which pieces move in the board - Alpha-beta pruning Algorithm - Sequence diagram
Rana Hossny hassan	1900266	- creating the main menu and option menu - handle mouse and keyboard events - Alpha-beta pruning optimization. - Sequence diagram
Kerolos sameh el-shaip boles	1900144	- draw state - tilemap drawing

Languages and Frameworks used

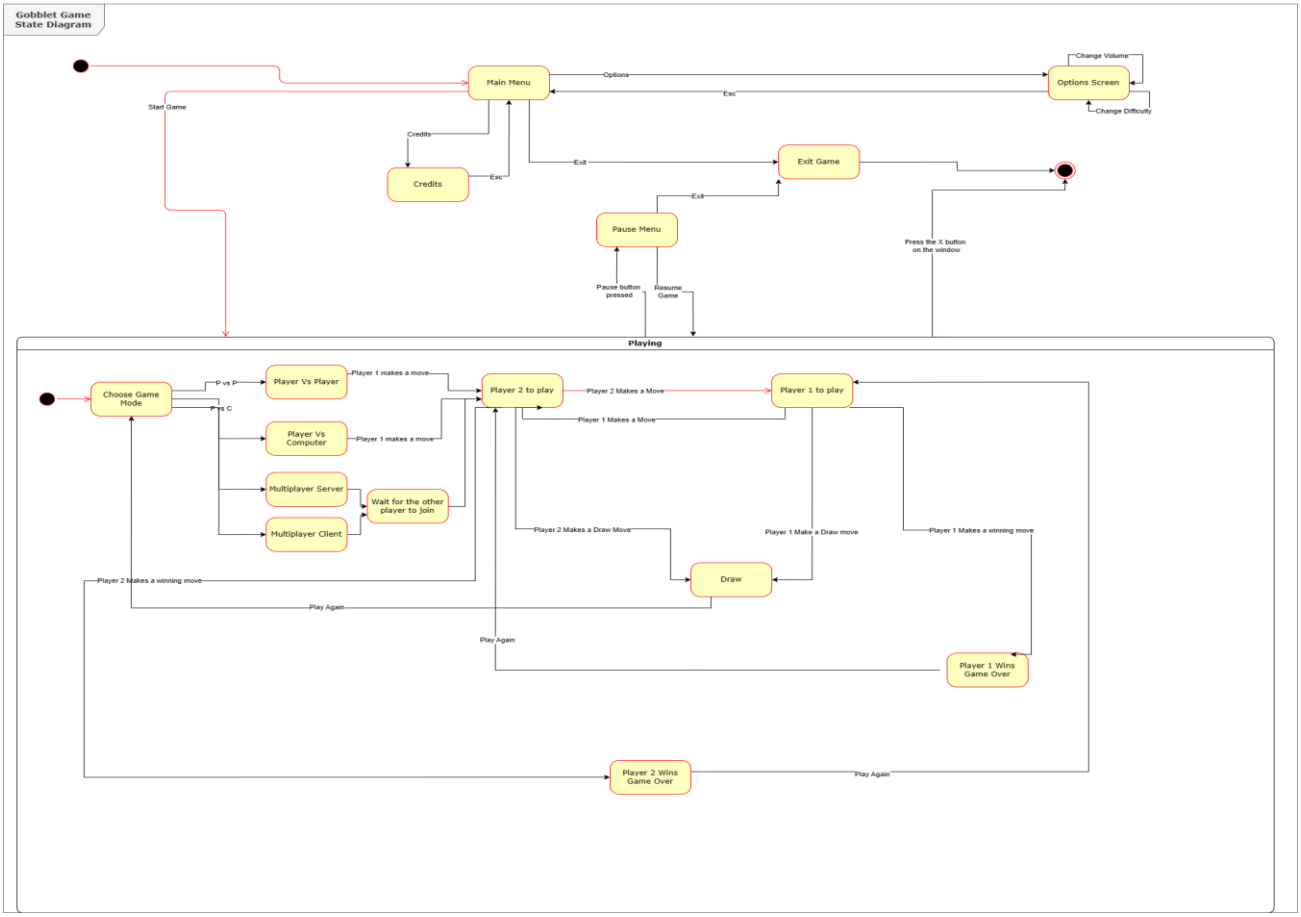
We utilized Python and the Pygame library to handle game logic and UI due to its resource availability, quick development with acceptable performance.

We also chose C++ to write our AI algorithms, because of its superior performance over python which allowed us to reach deeper levels in search trees.

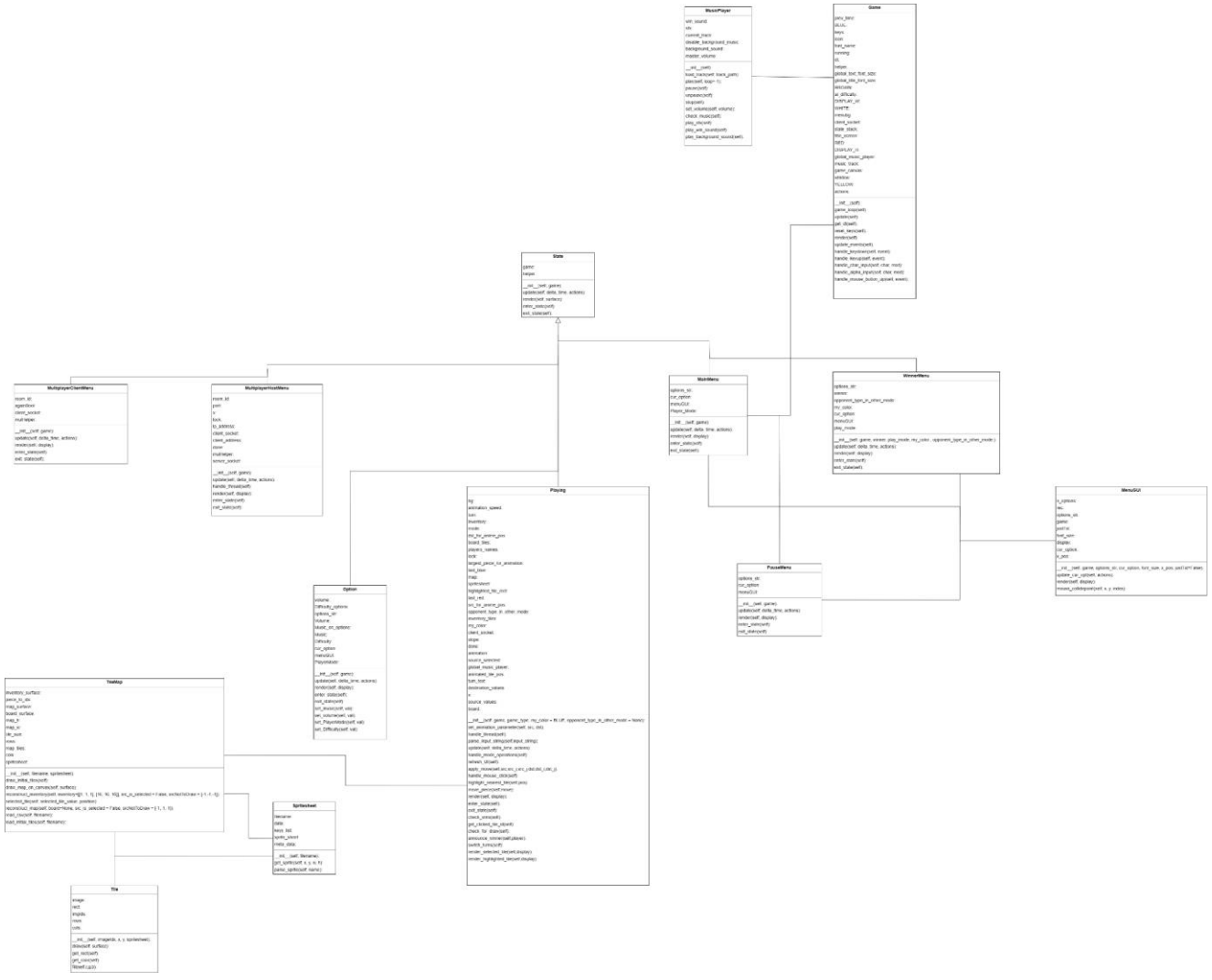
We utilized the Subprocess python module to run our C++ binary, write to its stdin, and intercept its stdout streams. We found from our tests the best alternative to wrapping our C++ code using python. This combination has optimized development efficiency and ensured optimal performance for game and AI components. also, in multiplayer, we utilized python sockets,

UML Diagrams

State Diagram

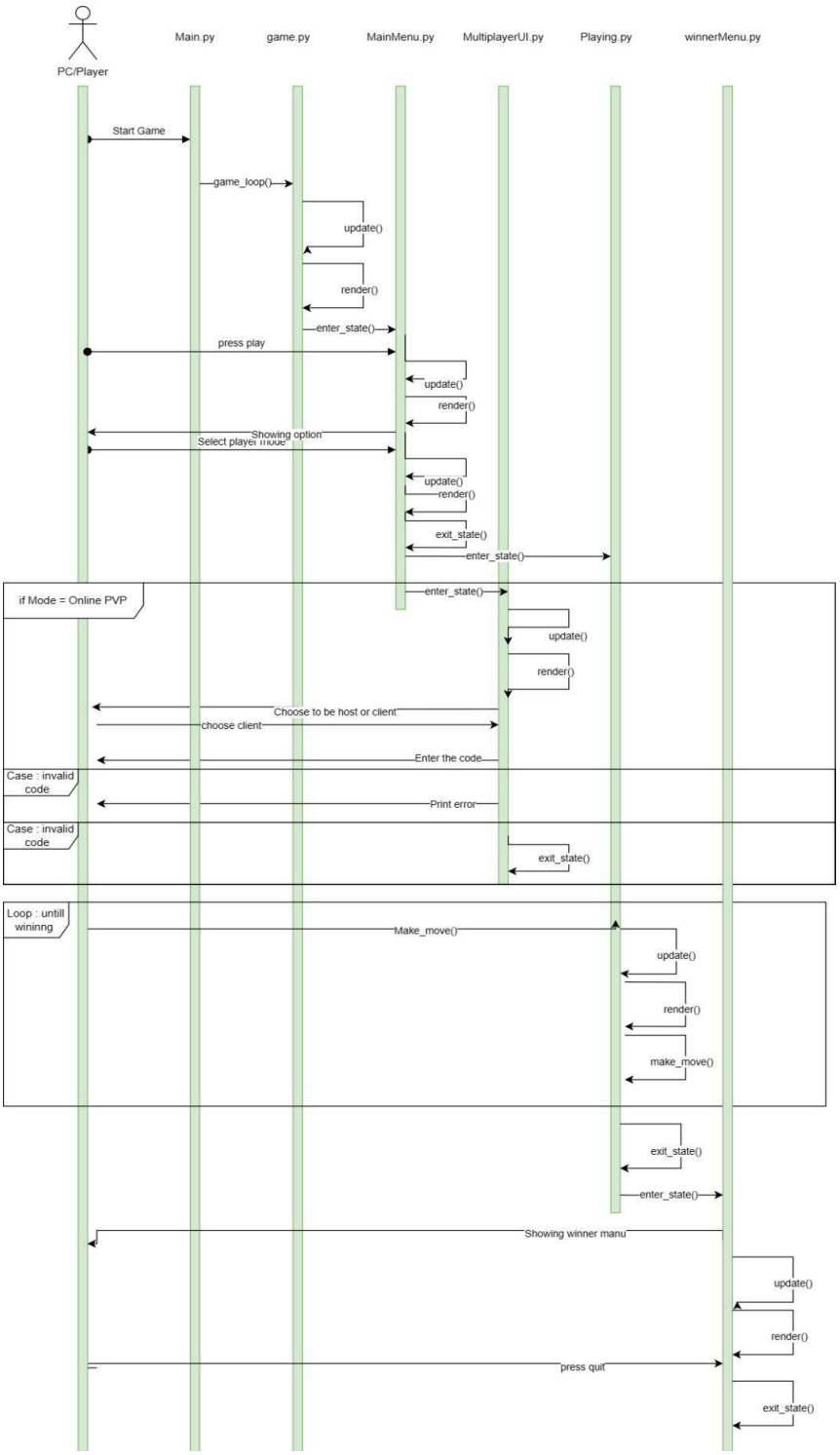


Class Diagram

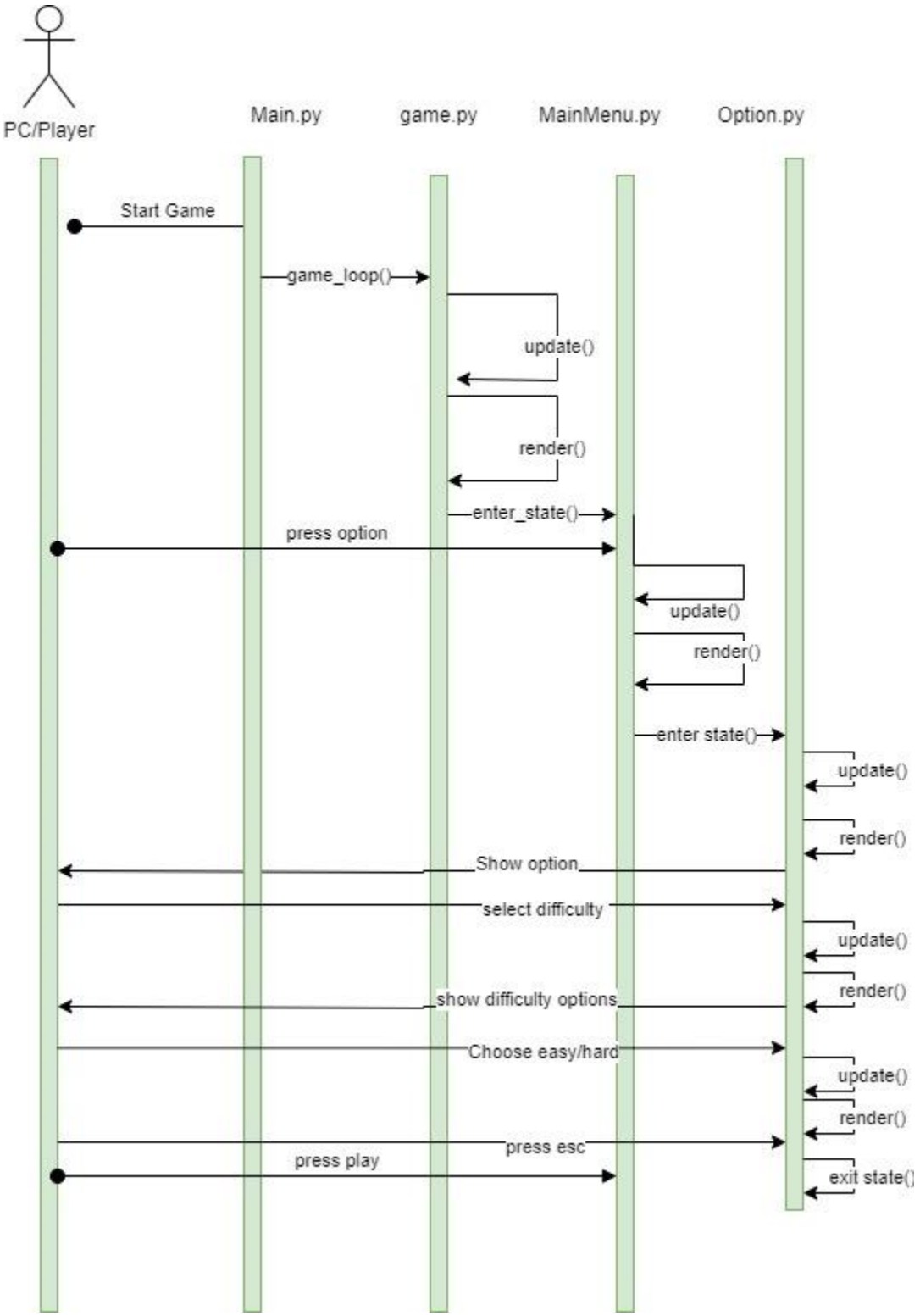


Sequence Diagrams

1- Sequence diagram for start gaming:



2- Sequence diagram for select option : easy difficulty and then start game :



Game-playing Supported Algorithms and heuristics

Min-Max

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.Every board state has a value associated with it. In a given state if the maximizer has the upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

Functionality:

- **Generate Child States:** The function starts by generating all possible child states from the current game state.
- **Base Case:** If the maximum depth is reached ,the function returns the current state as the base case of the recursion.
- **Maximizing Player Logic**: If it's the maximizer's turn, the function iterates through all child states, choosing the one that maximizes the evaluation. The function recursively calls itself with each child state, decreasing the depth, and alternating to the minimizing player.It updates the maximum evaluation found .
- **Minimizing Player Logic:** If it's the minimizer's turn, the function performs similarly but aims to minimize the evaluation score.It iteratively checks all child states for the one with the minimum evaluation and updates the minimum score.

Return Value:

- Returns a State representing the optimal move for the current player.

Time complexity (approximately) : O(b^d): where b is the average branching factor and d is the depth of tree.

Space Complexity (approximately): O(b*d): where b is the average branching factor, d is maximum depth of tree .

Alpha-beta pruning

Alpha-Beta pruning is an optimization technique for the minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree.

The Pseudocode:

```
// position : The current state of the game

//depth : The maximum depth of the game tree.

//alpha : The best already explored option along the path to the root for the maximizer.

//beta : The best already explored option along the path to the root for the minimizer.

//burning : A flag to enable Alpha-Beta pruning or not .

//mutation: A flag to enable  the mutation heuristic or not , which allows the AI to choose randomly among equally evaluated states for diversity in gameplay.

//—-----------------------------------------------------------------

Function minMax_alpha_beta(position: State, depth: Integer, alpha: Integer, beta: Integer, pruning: Boolean, mutation: Boolean) -> State

    Define evl , largest_Eval , minest_Eval as Integer

    Define temp, minest_state, largest_state as State

    Define childs_States as Vector of State

    childs_States = generate_possible_states(position, pruning AND (difficulty != 1))

    If difficulty = 1 AND size of childs_States > MXchild Then:

            Resize childs_States to MXchild

    If depth = 0 Then

            Return position
```

```
    If position.turn = 0 Then  // Maximizing player
            largest_Eval  =INT32_MIN
            Reverse the order of childs_States
         For i from 0 to size of childs_States - 1 Do
             largest_state = minMax_alpha_beta(childs_States[i], depth- 1,
alpha, beta, burning, mutation)
              evl = largest_state.static_evl
                alpha = max(alpha, evl)
            If evl > largest_Eval OR (evl = largest_Eval AND mutation AND
random integer modulo 3 = 1) Then:
                      temp = childs_States[i]
                      largest_Eval = evl
            If alpha >= beta AND burning Then:
                Break
    Else : // Minimizing player
        minest_Eval =INT32_MAX
        For i from 0 to size of childs_States - 1 Do:
            minest_state = minMax_alpha_beta(childs_States[i], depth - 1,
alpha, beta, burning, mutation)
            evl = minest_state.static_evl
            beta = min(beta, evl)
            If evl < minest_Eval OR (evl = minest_Eval AND mutation AND
random integer modulo 3 = 1) Then:
                temp = childs_States[i]
                     minest_Eval = evl
            If alpha >= beta AND burning Then
                   Break
Return temp
```

```
End Function
```

Functionality:

- **Generate Possible States**: Generates potential childeren states from the current position. The generation is influenced by the burning flag and game difficulty settings.
- **Dynamic State Reduction:** Limits the number of child states in easier difficulty settings .
- **Minimax function :** Adopts the classic Minimax approach, where the function recursively searches through the game tree to find the optimal move.
- **Mutation Heuristic:** When multiple moves have the same evaluation, the mutation heuristic (if enabled) allows the AI to randomly select among these moves, adding unpredictability to the AI's play style.

Optimization Strategies in Minimax Algorithm with Alpha-Beta Pruning:

1. **Sorting Child States for Enhanced Pruning**
    - Ascending Sort for Minimizer: Arrange the child states in ascending order based on their static evaluation values. This approach helps the minimizer (the opponent) to explore less promising moves first, which can lead to earlier alpha-beta pruning, reducing the number of explored nodes.
    - Descending Sort for Maximizer: Sort the Maximizer's child states in descending order. This strategy allows the maximizer (the AI player) to first consider the most advantageous moves, potentially reaching higher evaluation values sooner and triggering beta cutoffs effectively.
2. **Early Termination for Winning States**
    - Immediate Return on Winning Conditions: if the algorithm reaches winning states , it will return the state without exploring the other states .
3. **State Structure Enhancement**
    - To avoid redundant calculations of static evaluation:, the State data structure is updated to include the static_evl (static evaluation) attribute. This change means that each state carries its evaluation score, which is computed only once and can be easily accessed during the sorting and evaluation processes in the Minimax algorithm.

Time complexity (approximately): The worst-case performance of the algorithm is O(b^d):, where b is the average branching factor and d is the depth of the search tree . However, the best-case performance of the algorithm is O($b^{d/2}$)

Space Complexity (approximately): O(bd) :where b is the average branching factor into d is maximum depth of tree .

Iterative deepening

In iterative deepening, we used the Zobrist Hashing algorithm to hash each state to get a unique number that represents each state, the states and their corresponding evaluation in a was stored in a Transposition Table which is essentially a hashtable that is used to save the hash of each state with its corresponding evaluation function. We then added a time constraint to help reduce the time it takes to get the answer. We found that it has worse performance than Minimax with alpha-beta pruning. After doing some research, we settled on only using Minimax with alpha-beta pruning due to the following reasons:

1) The number of states generated was too large (thousands), and the read/write times on a Transposition Table was too long, this is because most of the hash tables are implemented to append elements that are colliding, which means that it has an access time closer to O(n) (where n is the number of elements stored in a hashtable) than O(1), we also tried implementing our own transposition table without a linked list but the results were not much different due to the large number of states being accessed, this meant that adding a transposition table would add more problems without much benefit.
2) Mini-max with alpha beta pruning is already responsive enough with an average of 1-2 seconds of response time as opposed to 3-4 seconds with iterative deepening (with no timing constraints), which also means that adding timing constraint is not really beneficial.
3) Iterative deepening without a transposition table is pointless. why search depth 1,2,3 when I can search in depth 3 directly?

Nonetheless, the code that we used to implement iterative deepening still exists in our github repo in the branch hashing-itr-deepening for testing purposes and maybe future improvements.

Tree generation (all possible children):

Early Return for Winning State:

- If the current state indicates a win, the function immediately returns a vector containing the current state.

Initialization:

- A vector evaluated_states of pairs <int, State> is declared.
- A vector possible_outcome_states is initialized with 5 copies of the current state.

Preparing Possible Destinations:

- A 3D vector possible_destination is created to store possible locations for each piece size.
- Two nested loops iterate through the board to populate possible_destination based on the largest piece size at each position.

Generating States from Board Moves:

- Another set of nested loops iterate through the board to generate new states for possible moves.
- States are created by moving pieces to each possible destination, and the resulting states are added to possible_outcome_states.

Inventory Moves:

- Another set of loops iterates through the inventory to generate states for possible moves.
- States are created by moving pieces from the inventory to each possible destination, and the resulting states are added to possible_outcome_states.

  Sorting:

If sorting is true, the generated states are sorted using a custom sorting function (customSort).

  Return:The vector possible_outcome_states is returned.

This function seems to be part of a game-playing AI, where it explores possible moves and generates successor states for a given state. The static_evaluation function is used to assign a static evaluation score to each generated state. Sorting the states may be beneficial depending on the game and the search algorithm used.

Used heuristics in evaluation function

1) for each row, column and diagonal, we calculate its score by summing up the scores for each piece and its corresponding colors, for example, the largest piece in red has a score of -5 and the largest piece in blue has a score of 5, the smallest one in red has a score of -2 and the smallest one in blue has a score of 2, and so on. We then take a greedy approach by summing up the maximum and minimum score which gives us an indicator of which player has the upper hand in this state.

2) if a player has 3 pieces lined up on the line to win, we give a large score in favor of the other player blocking this winning move.

3) if a state is a winning state, we give it a large evaluation score.

Other heuristics

1- **Mutation Heuristic at alpha-beta pruning :**

description : enabling adding random_condition (rand()%3==1) at the condition to select the best evaluation .

Benefits: When multiple moves have the same evaluation, the mutation heuristic (if enabled) allows the AI to randomly select among these moves, adding unpredictability to the AI's play style. so there will be a different sequence everytime we run this algorithm.

Difficulty level

The maximum difficulty level supported is Hard which corresponds to depth = 3, at this level it becomes very tough to play against the AI (it will always win).

The easy level corresponds to depth = 1, at this level it is very easy to win against the AI

Supported features

Intuitive User Interface: Experience a user-friendly interface designed for navigation and interaction within the game.

Adjustable Difficulty Levels: our game offers varied difficulty :easy and difficult.

Audio Customization: Immerse yourself in the game with our adjustable sound settings. Turn the music on or off and control the volume to match your gaming mood.

Adjustable game modes: the user can choose any mode from those modes :

- **Player vs. Player:** Challenge your friends or family in a face-to-face .
- **Player vs. Computer:** Test your skills against our sophisticated AI.
- **Computer vs. Computer:** Observe and learn strategies as the AI competes against itself.
- **Online player vs player:** Connect and compete with other player across the globe in real-time.

Comprehensive Menu Options:

- **Options Menu:** Tailor your gaming experience with customizable settings.
- **Pause Menu:** Take a break when needed without losing your game progress.
- **winner Menu:** dedication the game is ended and if the user want to restart the game.

Extra features

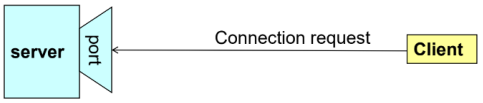
Player vs Player online

The game supports multiplayer, which is implemented through Sockets, where it's used to send information like move source, destination, player status (connected/disconnected).

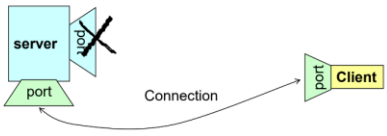
One player can host a room, the other player can join his room.

Note that a personalized virtual private network application like Hamachi, Game Ranger, Tunngle, etc MUST be used to play online. We can easily modify the code to support a personal hosted server without much change, and the concept will be the same.

- the connection is done through server-client peer-to-peer connection using Python sockets where The server opens a server socket and waits and listens to that socket using server_socket.listen() where for a client to make a connection request use client_socket.connect() , the server accepts the client's socket using server_socket.accept().



- When the server accepts the connection. Upon acceptance, the server gets a new socket and jumps the client to a different port and closes the original server socket as this game is 1vs1 so keeping the original socket open won't be needed.

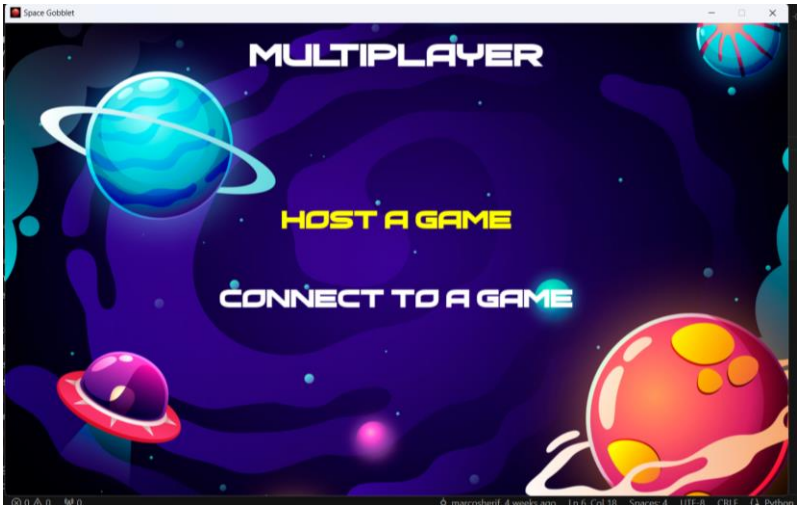


- The message between client and server is the move of each player where each player receives the move of the other side as a string and makes his move then sends it as a string also, and they keep exchanging till one player exits/ disconnects through the game in case this happened the client socket will be close then the game will send the player back to the main menu.
- To make secure connection between server and client , Hashing algorithm is used to concatenate the IP and PORT and cipher them to a unique ROOM-ID then client uses that ROOM-ID and decipher them to the original ip:port and then connect to the server socket also allowing players to join specific games in an easy way .

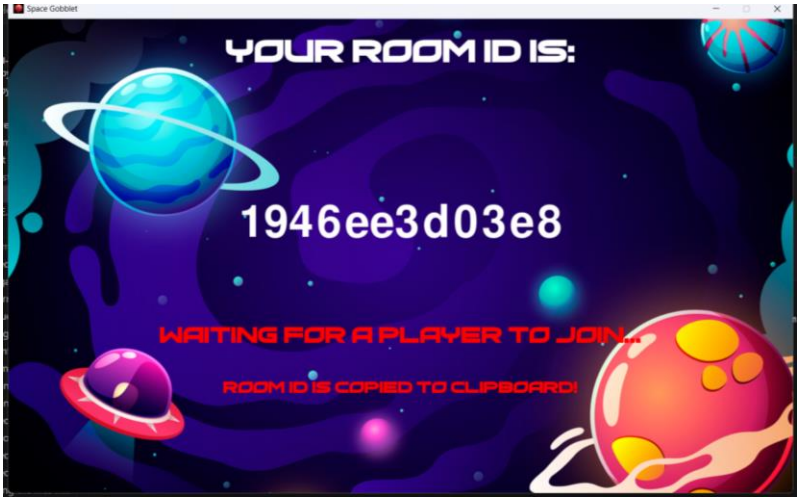
To host a room

a) from the main menu, press play and choose Online PVP mode.

b) choose to host a game.



c) Your Room ID will be copied to the clipboard automatically.

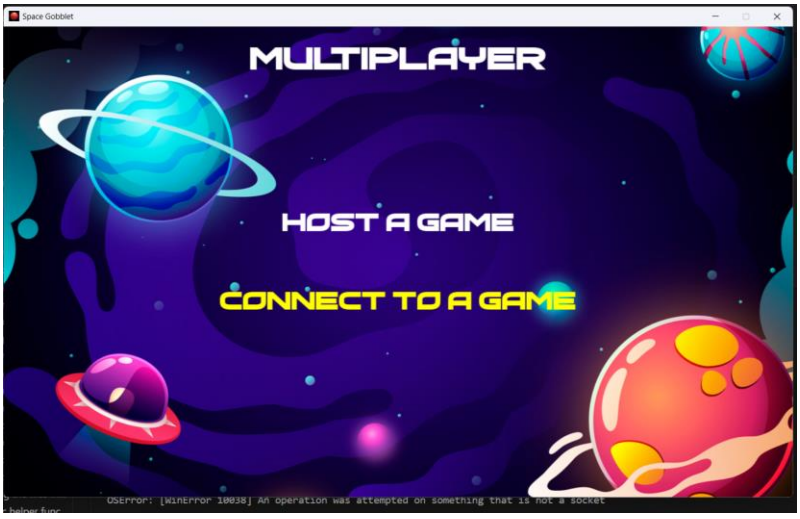


To connect to a host

a) from the main menu, press play and choose Online PVP mode.

b) choose to connect to a game.

c)



d) Enter the Host's Room ID (use ctrl+v to paste)

*

GPU parallelization with CUDA (not finished)

Before getting into the details of this section, let's talk about the reason why we chose to work on it in the first place.

Let's calculate the approximated number of operations done in the AI solver in a typical round with difficulty hard (in which the search tree is depth 3) :

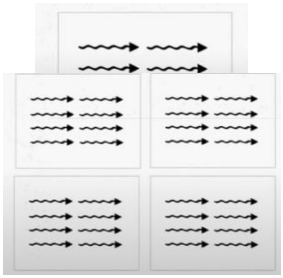
- Let b the average branching factor of the node and t the number of operations done in static evaluating the leaf nodes
- The upper bound of the total number of operations (assume no pruning for simplifying the calculation) =1 + b + b*b + b*b*b*t
- when t = 16 and b = 60 -> total number of operations is around 3.5 million operations which is an easy 1s task on a C++ program running in a modern CPU when running sequentially
- Adding more depth is impossible to run sequentially as if we did the same calculation with depth 4 we will get hours of execution time and years with depth 5

And that is why we should parallelize the calculations

GPU programming toolkit C++ for CUDA by Nvidia has been used because it is the most mature and well-documented and the work was done as follows:

Short overview of the technology:

The parallel architecture of Nvidia GPUs consists of a set of pipelined multiprocessors. The parallel computation on the GPU is performed as a set of concurrently executing thread blocks (kernels) details as follows:



- Threads: Single execution units that run kernels on the GPU. (Similar to CPU threads but there are usually many more of them).
- Thread Blocks: Thread blocks are a collection of threads. All the threads in any single thread block can communicate.
- Grid: A kernel is launched as a collection of thread blocks called the grid.

The host calls a kernel using a triple chevron <<< >>>. In the chevrons, we place the number of blocks and the number of threads per block. The following would launch 100 blocks of 256 threads each (total of 25600 threads).

For example, SomeKernel<<<50, 1024>>>(...);

This one would launch 50 blocks of 1024 threads each (51200 threads in total)

1. Testing the capabilities of the C++ Cuda framework

```
(base) mohammaddallash@dallash-pc-ubuntu:~/Documents/Gith
Block Size: 1024, Grid Size: 97656248
There are 49999998976 even numbers bellow 100000000000
Kernel Execution Time: 1701.37 ms
(base) mohammaddallash@dallash-pc-ubuntu:~/Documents/Gith
```

For experimenting purposes, We wrote a simple GPU parallelized script that counts the number of even numbers that are less than 10 power 11 (it is not even 100% parallelized as the counter is incremented using an atomic instruction) and it ran in 1.7 seconds only!!! which shows the capabilities of Cuda C++ framework.

2. literature review on previous implantations and techniques

we read a paper Parallel Alpha-Beta Algorithm on the GPU in which there is explanation an old approach of parallelizing the alpha-beta algorithm.

They implemented the algorithm iteratively as follows: they used a manually controlled stack of nodes in the shared memory. The parts of the alpha-beta algorithm that are executed in parallel by all of the threads in a block are node evaluation, move generation, and move execution.

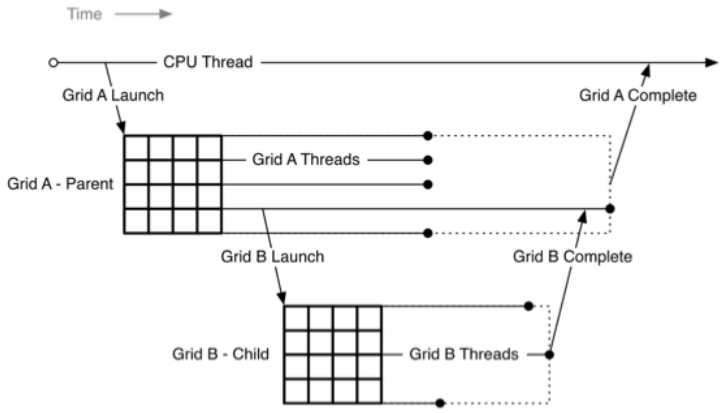
At the time of releasing the paper (2011) CUDA Dynamic Parallelism (a fancy term for making kernels execute other kernels that are necessary for recursive function calls) was not supported in their old hardware thus they had to make it iteratively

Luckily we do not have to proceed with this approach.

3. CUDA Dynamic Parallelism

Proceeding in CUDA Dynamic Parallelism implementation has a main challenge; As our code is not executed sequentially anymore, It is our job (the developers) to explicitly synchronize the flow of the program explained as follows:

Kernel grids launched with dynamic parallelism are fully nested. This means that child grids always complete before the parent grids that launch them, even if there is no explicit synchronization, as shown below.



If the parent kernel needs results computed by the child kernel to do its own work, it must ensure that the child grid has finished execution before continuing by explicitly synchronizing using cudaDeviceSynchronize(void). This function waits for the completion of all grids previously launched by the thread block from which it has been called. Because of nesting, it also ensures that any descendants of grids launched by the thread block have completed. cudaDeviceSynchronize() returns any error code of that has occurred in any of those kernels.

In conclusion, probably synchronization was the main challenge of this phase of the work which was done successfully.

4. The overhead of launching kernels

After some experimenting with, we discover that launching a new kernel introduces a noticeable time overhead of 2ms which does not sound that much but keeping in consideration that we launch a new kernel for each minMax call (which we do tenth of thousand of those) that will result in making an minMax with depth > 2 impossible …….. a complete failure of the whole idea : (

5. A sneaky workaround!

As we already migrated the AI solver to work on GPU in-depth 2 and it works just fine. we thought of mixing the two control flow as follows:

if depth <=1 :

launch the new branch on new kernel and run parallelised

else:

continue sequentially on the same kernel you are right now

That will result in running the depth 1 and 2 nodes parallelised and the deeper depth will run sequentially which will result in having around 300 kernel creations on for each round

that mixture of sequential and parallel unfortunately we didn't finish :(as we discover the overhead problem late.

All the experimenting was done a separate repo https://github.com/MohammadDallash/cuda-cpp-programming

and cuda mode is in separate branch

https://github.com/MohammadDallash/Gobblet-Ai-PYgame/tree/Cuda-C%2B%2B