Artificial Intelligence(CS5329)

Kalah Game

Implementation And Analysis Of Alpha-Beta And MinmaxAB Algorithm

PROJECT TEAM MEMBERS:

1. AKSHATHA MANOHAR JAIN
2. VIDHYASHREE,NAGABUSHANA
3. SUNDARA RAJ SREENATH,SAHANA

AUTHOR :

Akshatha Manohar Jain

netID : a_j289

StudentID : A04824992

Submission date : O4/27/2020

INDEX

1. Introduction

Artificial Intelligence is branch of computer science which deals with making intelligent machines. AI makes machines more smarter than humans. There are a lot of different algorithms and studies done under artificial intelligence to make life more easier. One of the major goal of artificial intelligence is to find the best solution from the available resources. Also, AI offers the machines to evolve. Here, evolving means machines getting more and more intelligent day by day.

Artificial intelligence is also used in game playing. The games that are played by the machines use artificial intelligence. Every move in the game is calculated and is a result of a very intelligent algorithm running in the background. There are many algorithms used for this purpose. Game developers have used techniques from AI research to create more challenging opponents. They can examine player behavior and change their responses to make the games more challenging using emergent behavior. The techniques used in AI game programming includes decision trees and path finding.

Artificial intelligence has long been used to simulate human players in board games. Computer Kalah Game is the best-known example. The aim of this project is to show how a Kalah game can be played using two different algorithms like Minimax-A-B and Alpha Beta search. We also concentrate on creating three evaluation functions and decide which algorithm works better with which evaluation function.
Game Playing is an important domain of artificial intelligence. Games don't require much knowledge; the only knowledge we need to provide is the rules, legal moves and the conditions of winning or losing the game.

Both players try to win the game. So, both of them try to make the best move possible at each turn. Searching techniques like BFS(Breadth First Search) are not accurate for this as the branching factor is very high, so searching will take a lot of time. So, we need another search procedures that improve,

- Generate procedure so that only good moves are generated.
- Test procedure so that the best move can be explored first.

The most common search technique in game playing is Minimax search procedure. It is depth-first depth-limited search procedure.

In this project we are going to write a program which will simulate Kalah Game to test two algorithms.

1. MINMAX-A-B
2. ALPHA-BETA-SEARCH

The algorithms will be mentioned in detail in the following chapters. In this section, I want to mention about Kalah Game.

AI is the area of computer science focusing on creating machines that can engage on behaviors that humans consider intelligent. The ability to create intelligent machines has intrigued humans since ancient times, and today with the advent of the computer and 50 years of research into AI programming techniques, the dream of smart machines is becoming a reality. Researchers are creating systems which can mimic human thought, understand speech, beat the best human chess player, and countless other feats never before possible. Find out how the military is applying AI logic to its hi-tech systems, and how in the near future Artificial Intelligence may impact our lives

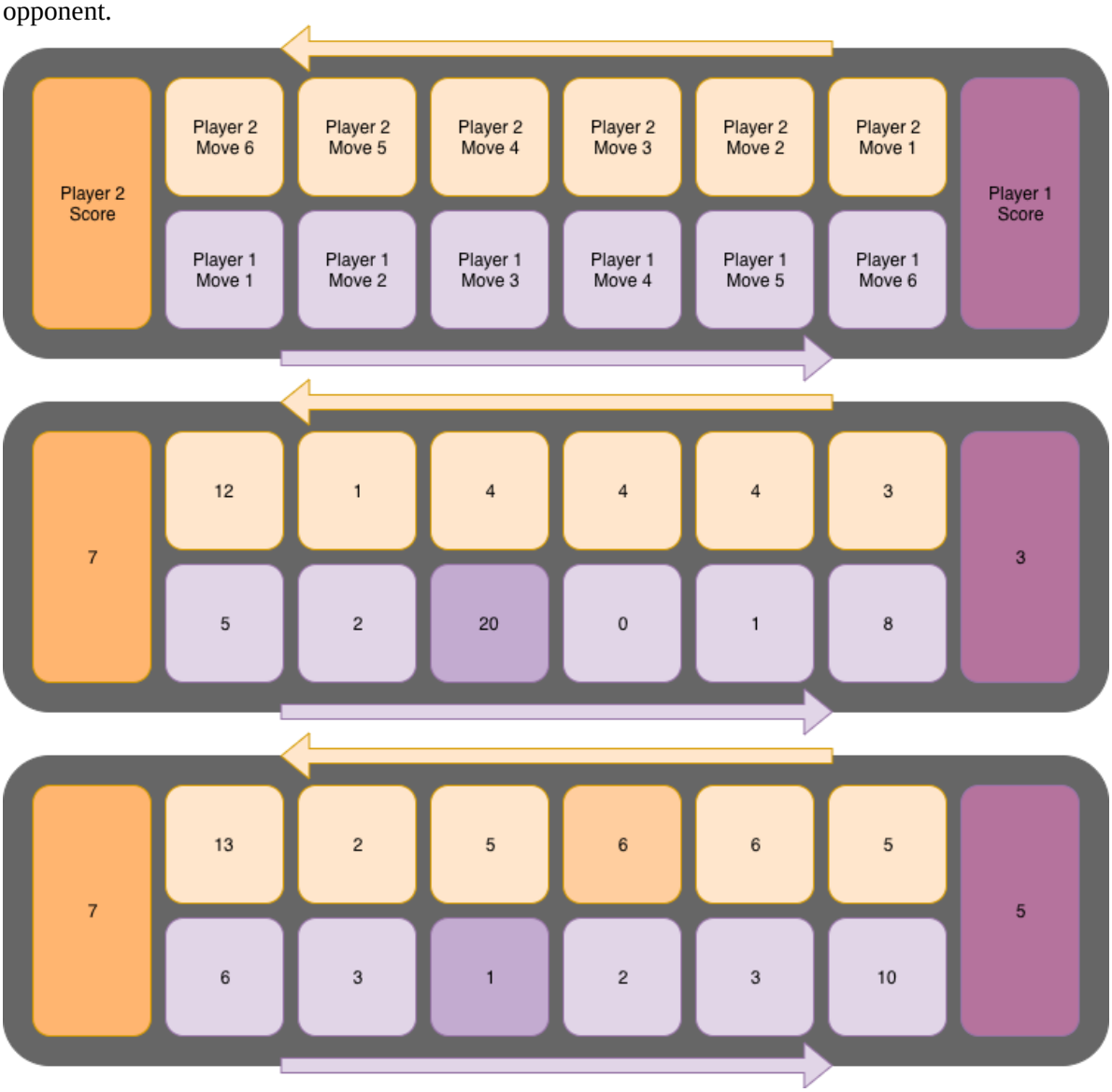
Goals

1. Deduction, reasoning, problem solving
2. Knowledge representation

3. Planning
4. Learning
5. Natural language processing (communication)
6. Perception
7. Motion and manipulation
8. Long-term goals
   8.1 Social intelligence
   8.2 Creativity
   8.3 General intelligence

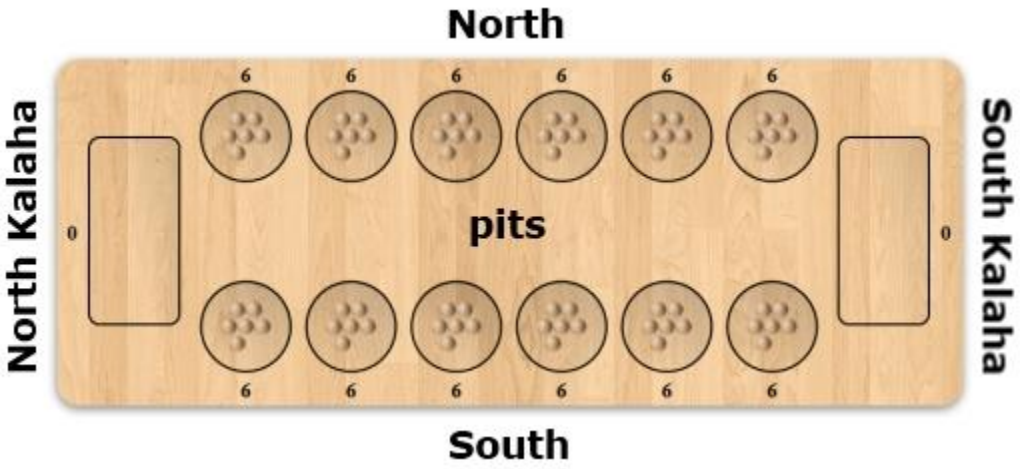
Kalah game:

Kalah is a count-and-capture game that is a fairly recent addition to the large group of *Mancalah* games. Kalah is a game that start with a number of seeds. There are 6 pots which are called houses and there are 2 big pots which are called end zone or Player stores. The purpose of the game is to have more seeds than opponent.



Solving (6,6)-Kalah

Kalah is an abstract strategy game invented in 1940 by William Julius Champion, Jr. The notation (*m*,*n*)-Kalah refers to Kalah with *m* pits per side and *n* stones in each pit. In 2000, Kalah was solved for all $m \leq 6$ and $n \leq 6$, except (6,6).



In each pit, there are initially 6 stones. A move is made by taking all stones from a pit on your own side and sowing them one-by-one in counterclockwise direction. Your own kalaha is included in the sowing, but the opponent's kalaha is skipped.

There are three possible outcomes of a turn:

- The sowing ends in your own kalaha: It is your turn to move again.
- The sowing ends in an empty pit on your own side: All stones in the opposite pit (on the opponent's side) along with the last stone of the sowing are placed into your kalaha and your turn is over.
- Otherwise (the sowing ends on the opponent's side or in a nonempty pit on your own side): Your turn is over.

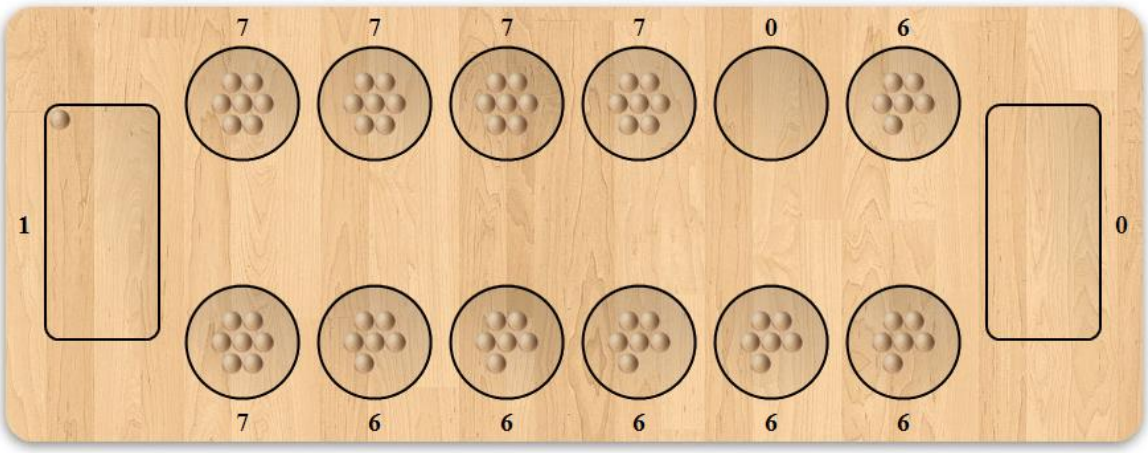
If all pits on your side become empty, the opponent captures all of the remaining stones in his pits. These are placed in the opponent's kalaha and the game is over.

You win the game when you have 37 or more stones in your kalaha. If both players end up with 36 stones, the game is tied.

Variations

- If all pits on your side become empty, you capture all of the remaining stones in your opponent's pits.
- If your sowing ends in an empty pit on your own side, but the opposite pit has no stones, then you are not allowed to capture the last stone of the sowing.

After the 1st move:



A move is made by taking all stones from a pit on your own side and sowing them one-by-one in counterclockwise direction. Your own kalaha (to the right) is included in the sowing, but the opponent's kalaha (to the left) is skipped.

There are three possible outcomes of a turn:

- The sowing ends in your own kalaha: It is your turn to move again.
- The sowing ends in an empty pit on your own side: All stones in the opposite pit (on the opponent's side) along with the last stone of the sowing are placed into your kalaha and your turn is over.
- Otherwise: Your turn is over.

If all pits on your side become empty, the opponent captures all of the remaining stones in his pits. These are placed in the opponent's kalah and the game is over.

You win the game when you have 37 (or more) stones in your kalah.

Results:

We have proven that the first player always wins Kalah with standard rules

The game values of Kalah can be summarized in the following table:

initial stones per pit

number of pits	1	2	3	4	5	6
1	D	L	W	L	W	D
2	W	L	L	L	W	W
3	D	W	W	W	W	L
4	W	W	W	W	W	D
5	D	D	W	W	W	W
6	W	W	W	W	W	**W**

2. The Problem Description

2.1 Problem statement

Game play is a domain of Artificial intelligence. Board games are generally developed using the search algorithm. A game tree of possible moves is developed and is used to find the best move. But the search tree will grow exponentially. It is not possible to search the entire search tree to find the best move. The next problem is how to compare the moves to find the best solution. So, the three main problems are creating the search tree, managing the tree depth, comparing the moves.

The main aim of this project is to create a "Kalah Game" using MINMAX A-B and ALPHA-BETA search algorithms. We create efficient evaluation function, to make intelligent move by the computer. Finally, we compare and analyze result of each algorithm.

Develop a Kalah game which will be played in between Computer Vs Computer. Use AlphaBeta and Minmax-Alpha-Beta algorithms to play this game. Write 3 evaluation functions to optimize the game states.

2.2 Solution

Basically the project is implemented to test the two algorithms which are MiniMaxAB and Alpha Beta search using Kalah as an example. With the help of heuristic functions we will evaluate the results of both the algorithms one of the major advantage of using heuristic function is that it reduces the number of states that are looked upon

Develop programs by implementing algorithms MINMAX-A-B and ALPHA-BETA-SEARCH in C or C++, language. Devise Deep-Enough (use some heuristics as given in Rich and Knight's book) and Move-Gen functions. Use "Kalah game" as an example to test your program. Execute your programs with the data and analyze the performance of each algorithm and each evaluation function by tabulating the total length of the game path, total number of nodes generated and expanded, execution time, the size of memory used by the program, and winning/losing statistics. Run the program for at least 4 different cutoff depths. Analyze the tabulated results and determine which algorithm and which evaluation functions are better.

Each node in the search tree should have a utility value, board states and the player. Next the depth of the search tree can be controlled by a Deep-Enough function. Deep Enough function evaluates the depth tree. So, that the tree cannot grow bigger than the pre-determined depth. The Utility function or the Evaluation function will give a utility value that can be used to compare the moves. The utility value is calculated for the terminal nodes and is then propagated up in the tree. The number of branches examined can be reduced by pruning away the branches that cannot influence the final decision. The Alpha-Beta and Minimax-A-B algorithm is used to search the trees.

This project shows how a game of Kalah proceed using two different algorithms namely Minimax-A-B and Alpha Beta search. These algorithms work their way and give the best path forward from all possible paths. Also, these algorithm work in tune with the evaluation functions that we have created.
Objective here is to check how well both these algorithms work in Kalah game using the different evaluation functions. By the end of this project we will be able to conclude which evaluation function works better with which algorithm.

Apart from that, this project will help us learn how to actually implement the Minimax-A-B algorithm and Alpha Beta search algorithm practically also how an evaluation function would change the output of these algorithms. This project also gives us a good chance to understand the importance of a good evaluation function.

Using some admissible and some non-admissible functions we will compare and tabulate the results. The entire code is written in C++ language and the results are documented.

3. Distribution of work

Team Members:
1. Akshatha Manohar Jain
2. Vidhyashree Nagabhushana
3. Sahana Srinath

AKSHATHA MANOHAR JAIN

- ✓ Each member of the team did extensive research on Kalah Game, the rules of the game, MINMAX-AB Algorithm, Alpha-Beta Search Algorithm and also, understood the Evaluation functions better.
- ✓ Helped in giving base idea about approaching the implementation during all the phases
- ✓ Implemented KalahFlow class.
- ✓ Discussed and implemented the Classes Kalah and KalahFlow.
- ✓ Created Evaluation function 2 and 3
- ✓ Did the unit testing of her modules and Integration testing after all the modules were integrated.

VIDHYASHREE NAGABHUSHANA

- ✓ Each member of the team did extensive research on Kalah Game, the rules of the game, MINMAX-AB Algorithm, Alpha-Beta Search Algorithm and also, understood the Evaluation functions better.
- ✓ Helped in giving base idea about approaching the implementation during all the phases
- ✓ Discussed and implemented the Classes Kalah and KalahFlow
- ✓ Implemented MinMaxAB search algorithm.
- ✓ Implemented the main function
- ✓ Created evaluation function 1
- ✓ Did the unit testing of my modules and Integration testing after all the modules were integrated.

SAHANA SREENATH

- ✓ Each member of the team did extensive research on Kalah Game, the rules of the game, MINMAX-AB Algorithm, Alpha-Beta Search Algorithm and also, understood the Evaluation functions better.
- ✓ Helped in giving base idea about approaching the implementation during all the phases
- ✓ Implemented AlphaBetaSearch algorithm.
- ✓ Discussed and implemented the Classes Kalah and KalahFlow
- ✓ Created evaluation function 4
- ✓ Did the unit testing of her modules and Integration testing after all the modules were integrated.

As integration was challenging, each member worked to make sure every function did its job the right way.

4. DOMAIN KNOWLEDGE

4.1. Two Players Game

Ever since the beginning of AI, there has been a great fascination in pitting the human expert against the computer. Game playing provided a high-visibility platform for this contest. It is important to note, however, that the performance of the human expert and the AI game-playing program reflect qualitatively different processes. More specifically, as mentioned earlier, the performance of the human expert utilizes a vast amount of domain specific knowledge and procedures. Such knowledge allows the human expert to generate a few promising moves for each game situation (irrelevant moves are never considered). In contrast, when selecting the best move, the game playing program exploits brute-force computational speed to explore as many alternative moves and consequences as possible. As the computational speed of modern computers increases, the contest of knowledge vs. speed is tilting increasingly in the computers favor, accounting for recent triumphs like Deep Blue's win over Gary Kasparov. [2]

There were two reasons that games appeared to be a good domain in which to explore machine intelligence:

- They provide a structured task in which it is very easy to measure success or failure.
- They did not obviously require large amounts of knowledge. They were thought to be solvable by straightforward search from the starting state to a winning position.

The first of these reasons remains valid and accounts for continued interest in the area of game playing by machine. Unfortunately, the second is not true for any but the simplest games. It is clear, that a program that simply does a straightforward search of the game tree will not be able to select even its first move during the lifetime of its opponent. Some kind of heuristic search procedure is necessary. One way of looking at all the search procedures we have discussed is that they are essentially generate-and- test procedures in which the testing is done after varying amounts of work by the generator. At one extreme, the generator generates entire proposed solutions, which the tester then evaluates. At the other extreme, the generator generates individual moves in the search space, each of which is then evaluated by the, tester and the most promising one is chosen.
Looked at this way, it is clear, that to improve the effectiveness of a search-based problem-solving program two things can be done:

- Improve the generate procedure so that only good moves (or paths) are generated.
- Improve the test procedure so that the best moves (or paths) will be recognized and explored first.

In game-playing programs, it is particularly important that both these things be done. If we use a simple legal-move generator, then the test procedure (which probably uses some combination of search and a heuristic evaluation function) will have to look at each of them. Because the test procedure must look at so many possibilities, it must be fast. So, it probably cannot do a very accurate job. Suppose on the other hand, that instead of a legal move generator, we use a plausible move generator in which only some small number of promising moves are generated. As the number of legal moves available increases, it becomes increasingly important to apply heuristics to select only those that have some kind of promise. With a more selective move generator, the test procedure can afford to spend more time evaluating each of the moves it is given so it can produce a more reliable result. Thus, by incorporating heuristic knowledge into both the generator and the tester, the performance of the overall system can be improved. Of course, in game playing, as in other problem domains, search is not the only available technique. In some games, there are at least sometimes when more direct techniques are appropriate. For example, in chess, both openings and endgames are often highly stylized, so they are best played by table lookup into a database of stored patterns. To play an entire game then, we need to combine search-oriented and non-search-oriented techniques.

The ideal way to use a search procedure to find a solution to a problem is to generate moves through the problem space until a goal state is reached. In the context of game-playing programs, a goal state is one in which we win. Unfortunately, for some of interesting games, it is not usually possible, even with a good plausible-move generator, to search until a goal state is found. The depth of the resulting tree (or graph) and its branching factor are too great. In the amount of time available, it is usually possible to search a tree only ten or twenty moves (called ply in the game-playing literature) deep. Then, in order to choose the best move, the resulting board positions must be compared to discover which is most advantageous.

This is done using a static evaluation function, which uses whatever information it has to evaluate individual board positions by estimating how likely they are to lead eventually to a win. Its function is similar to that of the heuristic function h' In the A* algorithm: in the absence of complete information, choose the most promising position. Of course, the static evaluation function could simply be applied directly to the positions generated by the proposed moves. But since it is hard to produce a function like this that is very accurate, it is better to apply it as many levels down in the game tree as time permits. A lot of work in game-playing programs has gone into the development of good static evaluation functions.

There were also some nonlinear terms reflecting combinations of these factors. But Samuel did not know the correct weights to assign to each of the components. So, he employed a simple learning mechanism in which components that had suggested moves that turned out to lead to wins were given an increased weight, while the weights of those that had led to losses were decreased.

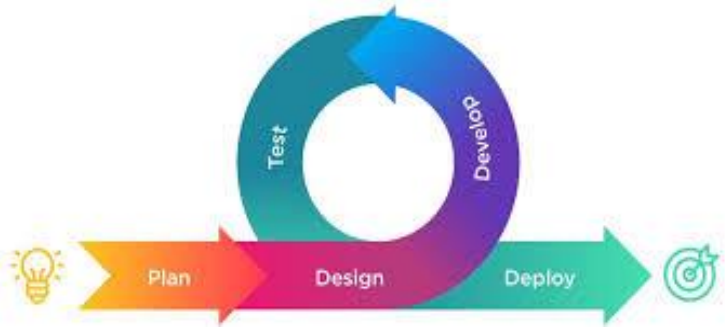
Unfortunately, deciding which moves have contributed to wins and which to losses is not always easy. Suppose we make a very bad move, but then, because the opponent makes a mistake, we ultimately win the game. We would not like to give credit for winning to our mistake. The problem of deciding which of a series of actions is responsible for a particular outcome is called the credit assignment problem [Minsky, 1963].

We have now discussed the two-important knowledge-based components of a good gameplaying program: a good plausible-move generator and a good static evaluation function. They must both incorporate a great deal of knowledge about the particular game being played. But unless these functions are perfect, we also need a search procedure that makes it possible to look ahead as many moves as possible to see what may occur. Of course, as in other problem-solving domains, the role of search can be altered considerably by altering the amount of knowledge that is available to it. But, so far at least, programs that play nontrivial games rely heavily on search.

4.2 Project Design

The following software development steps are followed to design the project.

1. Initial phase or Requirements Analysis.
2. Planning and Design
3. Implementation and Execution
4. Integration
5. Testing



- **Initiation**

  - As part of the project initiation, each member of the team had to do extensive research on the algorithms that had to be used. As part of the advanced artificial intelligence coursework, we had Minimax-A-B algorithm and Alpha Beta search. But to practically implement these algorithms was a challenge. We referred our textbook of the course and also the slides provided by the professor. Also, we found a lot of material online.

  - We did extensive study on these algorithms, studied the nuances in it. For better understanding of the Minimax-A-B, we first studied the Minimax algorithm. These algorithms are explained in the coming sections. Apart from this, we dedicated a lot of our time to learn the game of Kalah Game. To implement a game using these algorithms, it is utmost important to know the game properly.

  - Each member of the team watched multiple videos online and read a lot about the game from different sources. To better understand, we also played the game online. We also spent a lot of our time researching about evaluation function and how to write evaluation function as it is a very important part of this project. We had to create 3 evaluation functions as part of this project.

Each team member had to come up with one evaluation function atleast. All this helped us get good amount of knowledge about each module of the program. The next step was to come up with the plan on to actually implement it.

- **Planning and Design**

  The next step in the project development cycle was to plan and design the project. We knew we had to make the project as efficient as possible, hence we chose to go ahead with the object oriented approach. We had to make modules of the big project and each team member could take up each task module. As we decided to follow object oriented approach, we knew, we had to divide the project into different classes. We had to plan and design properly because the whole project depended on this.

  The main metrics for comparing strategies are the number of "stones" in each Kalah (home bin) when the game ends and the number of moves it took to complete the game. More heuristic strategies also used are discussed in the next sections. Given a game like Kalah, it is trivial to see that Mini-max, Alpha-Beta pruning fit very well to the game setup. Thus, these approaches were considered into our design and incorporated into our final build.

  Kalah is a well defined game involves searching and strategy, and has a reasonable search space for possible states. By running many combinations of heuristics, starting positions, and look-ahead depths, we effectively isolated the heuristics that perform the best in various categories.

➢ **Implementation and Execution**

This project uses two very important algorithms of artificial intelligence. They are Minimax-A-B (Minimax algorithm with Alpha Beta pruning) and Alpha Beta search algorithms. In the next sections, these two algorithms will be explained in detail.

Game trees

It is a structure in the form of a tree consisting of all the possible moves which allow you to move from a state of the game to the next state.

A game can be defined as a search problem with the following components:

• Initial state: It comprises the position of the board and showing whose move it is.
• Successor function: It defines what the legal moves a player can make are.
• Terminal state: It is the position of the board when the game gets over.
• Utility function: It is a function which assigns a numeric value for the outcome of a game.

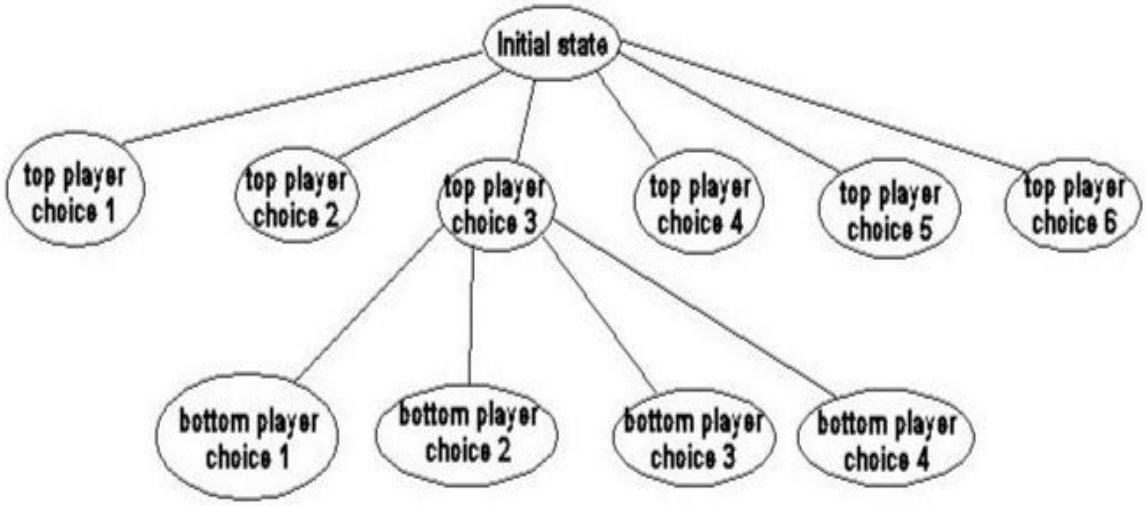
There are two players involved in a game, called MIN and MAX. The player MAX tries to get the highest possible score and MIN tries to get the lowest possible score, i.e., MIN and MAX try to act opposite of each other.

You can think of the possible game states as being arranged, conceptually, in a kind of search tree called a game tree. Each node of the tree contains a particular game state, g.

Its children are the game states that can result from making each valid move from the state g. The root of the tree is the initial game state; that is, the Kalah game before the first move is made.

The children of this initial state are all of the possible states that can arise from the top player making a valid opening move (since the top player goes first). There are six such states, corresponding to moving the beans out of each of the top player's six holes. (All other moves are illegal and, as such, are not considered.)

Here is a partial look at a Kalah game tree:



In the picture, from the initial state, there are six possibilities from which the top player may choose his move.

From one of those, "choice 3," the picture indicates that there are four choices that bottom player will then have. Not pictured are the choices that the top player can make as a result of each of the bottom player's choices. (Not surprisingly, the game tree can grow large rather quickly.) We'll call the leaves in such a game tree the final states. These leaves indicate the states in which one player or the other has won the game.

Game trees are, in general, very time consuming to build, and it's only for simple games that it can be generated in a short time. If there are b legal moves, i.e., b nodes at each point and the maximum depth of the tree is m, the time complexity of the minimax algorithm is of the order bm(O(bm)). To curb this

situation, there are a few optimizations that can be added to the algorithm. Fortunately, it is viable to find the actual minimax decision without even looking at every node of the game tree. Hence, we eliminate nodes from the tree without analyzing, and this process is called pruning.

The method that we are going to look in this article is called alpha-beta pruning. If we apply alpha-beta pruning to a standard minimax algorithm, it returns the same move as the standard one, but it removes (prunes) all the nodes that are possibly not affecting the final decision. AlphaBeta pruning is not actually a new algorithm, rather an optimization technique for 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. It cuts off branches in the game tree which need not be searched because there already exists a better move available.

In addition to the time complexity of a search depth and the Mini-max-to-Alpha-Beta performance improvement, most of the troubles were associated with getting Mini-max originally working correctly. Slowly stepping through the program and debugging allowed us to find out what the problems were and fix them accordingly.

- **Integration**

We ran into design decisions on how to represent board state, and therefore changed not only our board representation but also our entire state representation scheme a couple of times. With a software project increasing in size involving multiple team members, small changes in the code became less apparent as development progressed.

Once each class was created, we had to integrate all the classes together so that it would in sync. This was the most difficult part. We encountered many errors during this phase. Also, we had to include many functions in the main.cpp file to call the functions present in the class files. We had to make sure that all the modules of the project coordinated well with each other and worked to produce the desired output. This is the phase when we actually got everything together in the main.cpp.
To integrate all the different modules it was important to design the main. It was difficult to understand the impact of each class on the other. We had to do little coding in this phase to make it all work well together in the main.cpp. Most importantly, at this phase we also paid more attention to how our output looked. It was important to decide what to be displayed in the output. The major problem we faced in this respective was displaying the kalah board after each move. Displaying board after each move was a challenge. This project was a big project and hence the lines of code had increased tremendously after the integration of all the classes with the main.cpp file.

As mentioned in the above sections, we have concentrated to incorporated the object oriented programming concepts. At every stage we have tried to keep that in mind. The integration phase was challenging because we needed to keep the classes or modules in the project but at the same time it was difficult to get everything connected.

Early in Mini-max development, we discovered that we were thinking about the utilities in the wrong manner. We were initially not trying to maximize our utility and minimize the opponent's utility, but were instead looking at the board from the wrong perspective when deciding on moves. After some discussion this was resolved so that each agent analyzed the board using their own point-of-view/perspective.

- **Testing**

Once all the modules of the project were got together and connected, we had to do the integration testing. But before coming to the integration testing part of the project, we have to mention the unit testing that was done. As any software development cycle, this project also had to go through each and every phase in a

timely manner. And, as we always talk about testing being a part of every phase in the development cycle, even in this project, we did unit testing at every stage of the development of this project.
Unit testing is the testing done on each module. We did unit testing for each and every class or the module of the project. As mentioned above, we had divided the work among all the team members. Each team member took care of each class. The procedure that we followed was, as soon as we developed a particular function or the sub module, that was tested at that very instant itself. Even if it couldn't produce an output because it was dependent on other modules or functions, we compiled it and tested the module intensively for any kind of error present. This was very difficult because unit testing had to be done right or else later we would have bigger problems because of that.

Once all the modules or classes were developed and compiled and integrated together, now was the time for integration testing. We had to test if the project produced the desired output. One of the important purpose of this project to analyze the results of this project and investigate which evaluation function worked better for which search algorithm among the two search algorithms used in this project.
All the members of the team worked together to do the integration testing because we did have a lot of issues at this stage. Once the integration testing was done, we were done with the implementation of the project.

Early in Mini-max development, we discovered that we were thinking about the utilities in the wrong manner. We were initially not trying to maximize our utility and minimize the opponent's utility, but were instead looking at the board from the wrong perspective when deciding on moves. After some discussion this was resolved so that each agent analyzed the board using their own point-of-view/perspective.
In addition to the time complexity of a search depth and the Mini-max-to-Alpha-Beta performance improvement, most of the troubles were associated with getting Mini-max originally working correctly. Slowly stepping through the program and debugging allowed us to find out what the problems were and fix them accordingly.

4.3 Heuristics

The study of artificial intelligence has much to say about good ways to search toward a goal when it's impractical to check all possible paths toward it. We can first make use of the following observation:
Suppose the top player has made a move in the game, and the bottom player wants to figure out the best move to make, using the search tree approach we've been discussing.

Then the bottom player need only concern himself with the subtree that has the current game state as its root. Once a move is made, all the other moves that could have been made can be ignored, as it is now not possible to take those paths down the tree.

Thus, when analyzing the next move to make, we need only generate the part of the search tree that originates from the current game state. This approach reduces our storage needs significantly and we don't waste time or memory processing parts of the tree that we can no longer reach.
Even if we generate only the part of the tree that we need, that part may still be much too large to store. This is where a heuristic search comes into play. In a heuristic search, we generate as much of the relevant subtree as is practical, using the resulting game states to guide us in selecting a move that we hope will be the best.

There are several strategies that we could use. At the heart of the strategy that we'll use is the notion of an evaluation function that we discussed earlier. We'll need to rate each particular game state in some way, so that we can decide which of a large number of game states is the best outcome for us.
A simple approach is the following:

eval(state) = beans in my pot in this state - beans in opponent's pot
It's also important to note here that you do not need to actually build a game tree in memory. Our algorithm will perform a sort of depth-first search on the game tree, meaning that we can use parameters in a recursive method (stored on the run-time stack) to perform the search, negating the need to actually build and store a game tree. This will dramatically reduce the amount of memory needed to choose a move, since only one path in the tree will ever need to be stored on the run-time stack at a time. Putting these ideas together, we can develop a search algorithm that will look for the move that leads to the game state that evaluates to the highest value.

That algorithm looks something like this:

```
int search(GameState s, int depth)
{
        if (depth == 0)
                return evaluation of s
        else
        {
        if (it's my turn to move)
        {
                for each valid move that I can make from s
                {
                        make that move on s yielding a state s'
                        search(s', depth - 1)
                }
                return the maximum value returned from recursive search calls
        }
        else
        {
                for each valid move that my opponent can make from s
                {
                        make that move on s yielding a state s'
                        search(s', depth - 1)
                        }
                        return the minimum value returned from recursive search calls        }
        }
}
```

There are a couple of things we need to discuss about the algorithm above:
First, notice that there are two cases of recursion: either it is the computer player's turn (who is currently making the decision) or its opponent's turn. In each case, the algorithm is almost the same, except:

- when it is the computer player's turn, the maximum value is returned. In other words, the computer player wants to make the best possible move it can.
- when it is the opponent's turn, the minimum value is returned. This is because it is assumed that the opponent will also make the move that's in its best interest (which is in our worst interest).

You may not assume that the computer player will always be the top or the bottom player. The top or the bottom player (or both!) might be a computer player.

When deciding whether it's "my turn" or "my opponent's turn," we'll have to exercise some caution to ensure that you're making the right decision.

Second, notice the depth parameter. This will be used to limit the depth of our search, to make sure that our search is of a manageable length. Each time we recurse one level deeper, the depth is reduced by one, and we stop recursing when it reaches zero.

Thirdly, observe that when the top player makes a move, it isn't necessarily the case that the bottom player will be making the next move. So, care must be taken in deciding whose turn it is.

The easiest way to deal with this problem is to let the current game state keep track of this for you. Lastly, note that this algorithm returns the evaluation of the best state, not the best state itself. We'll need to exercise some care in actually implementing this algorithm so that it will be able to actually choose a move.

A Heuristic is a technique to solve a problem faster than classic methods, or to find an approximate solution when classic methods cannot. This is a kind of a shortcut as we often trade one of optimality, completeness, accuracy, or precision for speed. At each branching step, it evaluates the available information and decides on which branch to follow. It does so by ranking alternatives. Heuristics can incorporate whatever knowledge we can build into our program.

4.4 Evaluation Function

An Evaluation Function, also known as Heuristic Evaluation function or Static Evaluation Function by game-playing programs to estimate the value or goodness of a position in the minmax and related algorithms. The evaluation function is typically designed to be fast and accuracy is not a concern (therefore heuristic); the function looks only at the current position and does not explore possible moves (therefore static). The performance of a game playing program depends upon the quality of the evaluation function. It assigns a approximate material value for each position. Basic criteria for an Evaluation function:

1. Evaluation function must agree with the utility function on terminal states.
2. It must not take too long.
3. It should accurately reflect the actual chances of winning (i.e. should use probability).

One popular strategy for constructing evaluation functions is as a weighted sum of various factors that are thought to influence the value of a position. The Evaluation Function calculates the next move by assuming that the opposition will take the best possible move in its current situation. The utility or worth of a position is calculated by taking the difference between the players score and the opponents score

5. Methodologies

In this project, we will mainly focus on Minimax-AB algorithm from Rich & Knight and Alpha-Beta search from Russell & Norvig. But both algorithms are improved versions of Minimax algorithm which improve the efficiency of Minimax algorithm.

1. MINIMAX SEARCH
2. ALPHA BETA PRUNING

5.1 MINIMAX SEARCH

The minimax search procedure is a depth- first, depth-limited search procedure. The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions. Now we can apply the static evaluation function to those positions and simply choose the best one.
After doing so, we can back that value up to the starting position to represent our evaluation of it. The starting position is exactly as good for us as the position generated by the best move we can make next. Here we assume that the static evaluation function returns large values to indicate good situations for us, so our goal is to maximize the value of the static evaluation function of the next board position.

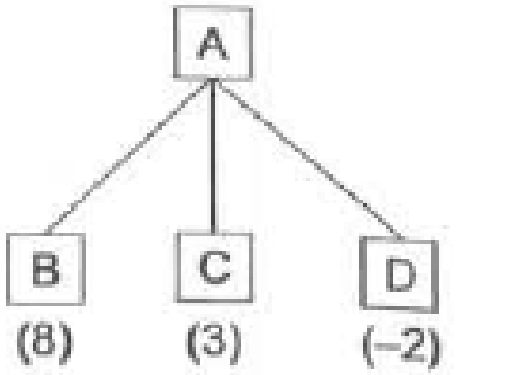


Fig. 12.1 *One-Ply Search*

An example of this operation is shown in Fig. It assumes a static evaluation function that returns values ranging from - 10 to 10, with 10 indicating a win for us, - 10 a win for the opponent, and 0 an even match. Since our goal is to maximize the value of the heuristic function, we choose to move to B. Backing B's value up to A, we can conclude that A's value is 8, since we know we can move to a position with a value of 8.

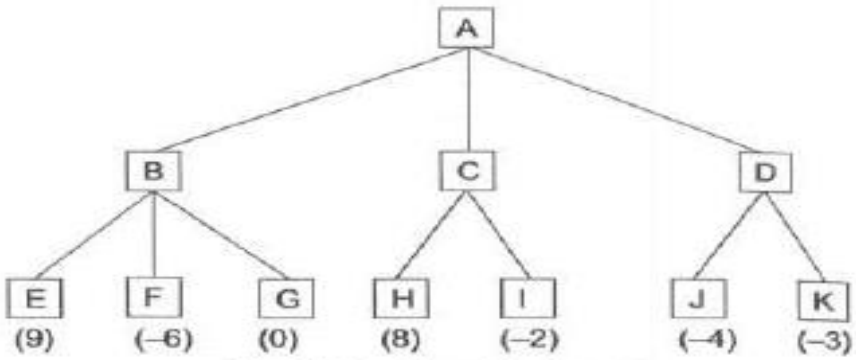


Fig. 12.2 *Two-Ply Search*

But since we know that the Static evaluation function is not completely accurate, we would like to carry the search farther ahead than one ply. This could be very important, for example, in a chess game in which we are in the middle of a piece exchange. After our move, the situation would appear to be very good, but, if we look one move ahead, we will see that one of our pieces also gets captured and so the situation is not as favorable as it seemed. So, we would like to look ahead to see what will happen to each of the new game positions at the next move which will be made by the opponent. Instead of applying the static evaluation function to each of the positions that we just generated, we apply the plausible-move generator, generating a set of successor positions for each position. If we wanted to stop here, at two-ply look ahead, we could apply the static evaluation function to each of these positions, as shown in Fig.

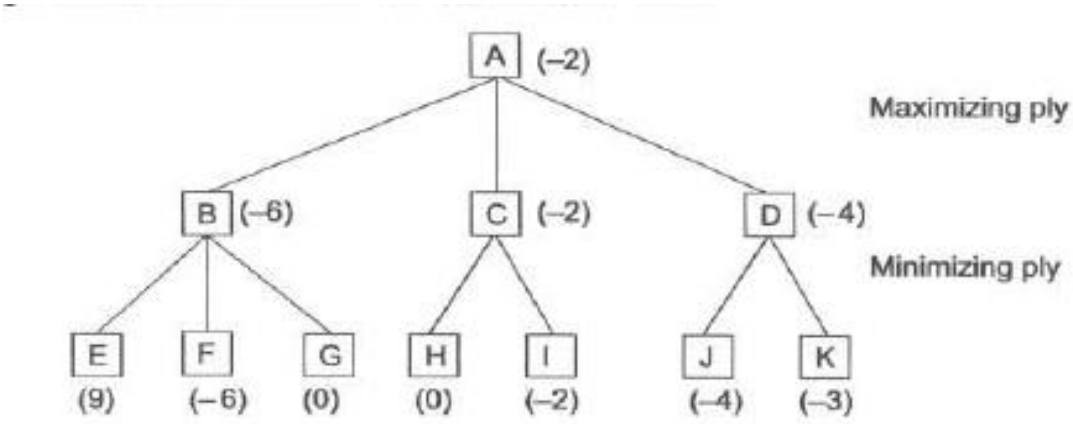


Fig. 12.3 *Backing Up the Values of a Two-Ply Search*

But now we must take into account that the opponent gets to choose which successor moves to make and thus which terminal value should be backed up to the next level. Suppose we made move B. Then the opponent must choose among moves E, F, and G. The opponent's goal is to minimize the value of the evaluation function, so he or she can be expected to choose move F. This means that if we make move B, the actual position in which we will end up one move later is very bad for us. This is true even though a possible configuration is that represented by node E, which is very good for us. But since at this level we are not the ones to move, we will not get to choose it. Figure 12.3 shows the result of propagating the new values up the tree. At the level representing the opponent's choice, the minimum value was chosen and backed up. At the level representing our choice, the maximum value was chosen. Once the values from the second ply are backed up, it becomes clear that the correct move for us to make at the first level, given the information we have available, is C, since there is nothing the opponent can do from there to produce a value worse than - 2. This process can be repeated for as many plies as time allows, and the more accurate evaluations that are produced can be used to choose the correct

move at the top level. The alternation of maximizing and minimizing an alternate ply when evaluations are being pushed back up corresponds to the opposing strategies of the two players and gives this method the name minimax. Having described informally the operation of the minimax procedure, we now describe it precisely. It is a straightforward recursive procedure that relies on two auxiliary procedures that are specific to the game being played.

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

5.2 Minimax algorithm

1. If DEEP-ENOUGH(Position, Depth), then return the structure VALUE = STATIC(Position, Player); PATH = nil This indicates that there is no path from this node and that its value is that determined by the static evaluation function.
2. Otherwise, generate one more ply of the tree by calling the function MOVE- GEN(Position Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned if DEEP-ENOUGH had returned true.
4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows. Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the best score that can be achieved by an element of SUCCESSORS. For each element SUCC of SUCCESSORS, do the following:

(a) Set RESULT-SUCC to MINIMAX(SUCC, Depth + 1, OPPOSITE(Player)) This recursive call to MINIMAX will actually carry out the exploration of SUCCC.

(b) Set NEW-VALUE to - VALUE(RESULT-SUCC). This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower Ievel.

(c) If NEW-VALUE> BEST-SCORE, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:

(i) Set BEST-SCORE to NEW-VALUE.

(ii) The best-known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined by the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).

5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So, return the structure VALUE = BEST-SCORE PATH = BEST-PATH

5.3 Minimax procedure

- Evaluate positions at cutoff search depth and propagate information upwards in the tree
- Score of MAX nodes are the maximum of child nodes
- Score of MIN nodes is the minimum of child nodes
- Bottom-up propagation of scores eventually gives score for all possible moves from root node
- This gives us the best move to make

5.4 Improvement required in Minimax

As with many recursive programs, a critical issue in the design of the minmax procedure is when to stop the recursion and simply call the static evaluation function. There are variety of factors that may influence this decision. They include:

- Has one side Won?
- How many plies have we already explored?
- How promising is this path?
- How much time is left?
- How stable is this configuration?

Minimax is horrendously inefficient If we go to depth d, branching rate b, we must explore bd nodes but many nodes are wasted. We needlessly calculate the exact score at every node but at many nodes we don't need to know exact score e.g. outlined nodes are irrelevant. One problem that arises in defining MINIMAX as a recursive procedure is that it needs to return not one but two results:

- The backed-up value of the path it chooses.
- The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, is actually needed.

5.5 Is there a good Minimax?

- Yes, we need to prune branches we do not to search from the tree
- start propagating scores as soon as leaf nodes are generated
- do not explore nodes which cannot affect the choice of move
- or in other words, do not explore nodes that we can know are no better than the best found so far
- The method for pruning the search tree generated by minimax is called Alpha-beta.

Properties of Mini-Max algorithm

- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **$O(b_m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.

- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

Limitation of the minimax Algorithm

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we will discuss in the next topic.

5.6 Pruning search algorithm

The search tree will grow exponentially, and it is not possible to store the entire tree and examining it. The most straightforward approach to controlling the amount of search is to set a fixed depth limit, so that the cutoff test succeeds for all nodes at or below depth d. The depth is chosen so that the amount of time used will not exceed what the rules of the game allow. A slightly more robust approach is to apply iterative deepening. When time runs out, the program returns the move selected by the deepest completed search. But these approaches can have some disastrous consequences because of the approximate nature of the evaluation function. Consider a situation where the player makes a move emptying hole in his region. Next opponent plays he make a move that makes him to capture all the stones in the players hole. This will be a win situation for opponent and not the player. So, to avoid these cases the player has to look into one more ply. But if we use pruning algorithms then the unprofitable branches are pruned away, this gives us enough time to look into one more ply

5.7 Alpha beta pruning

Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.

As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.

Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

The two-parameter can be defined as:

a. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.

b. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Intuitively, alpha beta is an improvement over minimax that avoids searching portions of the tree that cannot provide more information about the next move. Specifically, alpha-beta is a depth-first, branch and-bound algorithm that traverses the tree in a fixed order (such as left to right) and uses the information it gains in the traversals to "prune" branches that can no longer change the minimax value at the root. Such cut off branches consist of options at game positions that the player will always avoid, because better choices are known to be available.

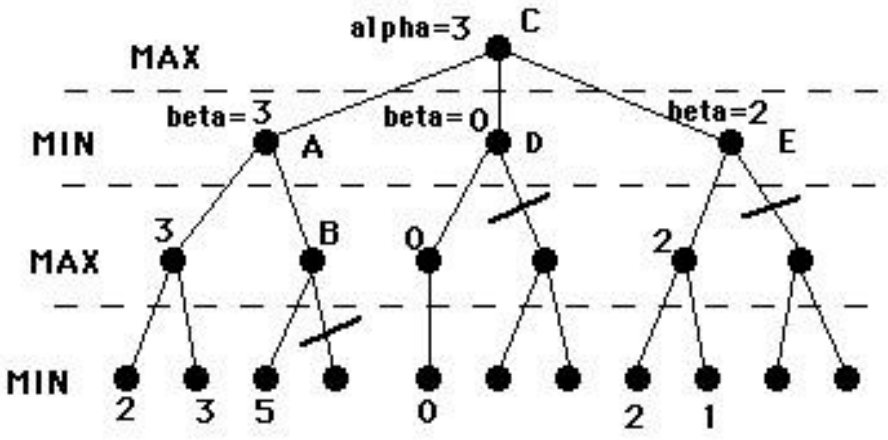
- An alpha value is an initial or temporary value associated with a MAX node. Because MAX nodes are given the maximum value among their children, an alpha value can never decrease; it can only go up.
- A beta value is an initial or temporary value associated with a MIN node. Because MIN nodes are given the minimum value among their children, a beta value can never increase; it can only go down.

Alpha (α): a lower bound on best that the player to move can achieve.
Beta (β): an upper bound on what the opponent can achieve.
Whenever $\alpha \geq \beta$higher, or $\beta \leq \alpha$higher further search at this node is irrelevant.

For example, suppose a MAX node's alpha = 6. Then the search needn't consider any branches emanating from a MIN descendant that has a beta value that is less-than-or-equal to 6. So, if you know that a MAX node has an alpha of 6, and you know that one of its MIN descendants has a beta that is less than or equal to 6, you needn't search any further below that MIN node. This is called alpha pruning.

The reason is that no matter what happens below that MIN node, it cannot take on a value that is greater than 6. So, its value cannot be propagated up to its MAX (alpha) parent.
Similarly, if a MIN node's beta value = 6, you needn't search any further below a descendant MAX that has acquired an alpha value of 6 or more. This is called beta pruning.

Let’s see an example :



1) Start at C and descend to full depth and assign the heuristic to a state and siblings (min 2,3) back up these values to their parent node max(3).
2) Offer this value to A as its beta value. So now A has a beta value 3 and its value can be no larger than 3.
3) Descend to other grandchildren of A and terminate the search of their parent if any of the grandchildren has a value >= A’s value i.e. 3. Here the B is Beta pruned as its value is > than 3, because it has to be at least 3.
4) Once A value is known assign it to its parent C. Now C has an alpha value 3.

5) Repeat the process to C's grandchildren and now D is alpha pruned because its value is less than the alpha=3. Because it can never be greater than 0.
6) Repeating the process as we descend to E, E is alpha pruned as its value is 2 and it is less than the alpha value 2.
7) Thus, the c value is Alpha=3.

Key points about alpha-beta pruning

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

5.8 Rules of Alpha-beta pruning

- Alpha Pruning: Search can be stopped below any MIN node having a beta value less than or equal to the alpha value of any of its MAX ancestors.
- Beta Pruning: Search can be stopped below any MAX node having an alpha value greater than or equal to the beta value of any of its MIN ancestors.

Move Ordering in Alpha-Beta pruning

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b_m)$.
- **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b_{m/2})$.

Rules to find good ordering

Following are some rules to find good ordering in alpha-beta pruning:

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move.
- We can bookkeep the states, as there is a possibility that states may repeat

5.9 Alpha beta algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

5.10 Importance of cut off

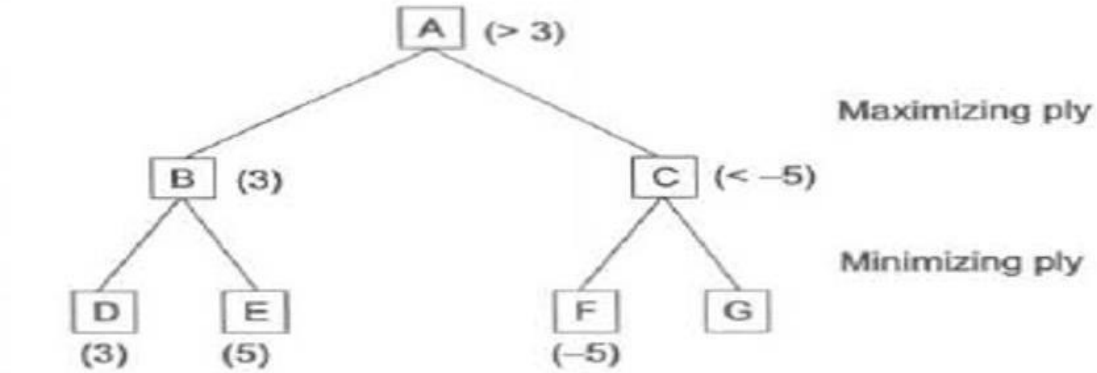
- If you can search to the end of the game, you know exactly the path to follow to win
- Thus, the further ahead you can search, the better
- If you can ignore large parts of the tree, you can search deeper on the other parts
- The number of nodes at each turn grows exponentially
- You want to prune branches as high in the tree as possible
  - Exponential time savings possible

5.11 Minmax A B

When applied to a standard minimax tree, it prunes away branches that cannot possibly influence the final decision.

MINIMAX-A-B uses two values, USE-THRESH and PASS-THRESH. USE-THRESH is used to compute cutoffs. PASS-THRESH is merely passed to the next level as its USE-THRESH. Of course, USE-THRESH must also be passed to the next level, but it will be passed as PASS-THRESH so that it can be passed to the third level down as USE-THRESH again, and so forth. Just as values had to be negated each time they were passed across levels, so too must these thresholds be negated. This is necessary so that, regardless of the level of the search, a test for greater than will determine whether a threshold has been crossed.

To see how to do this, let's return first to the simple example



At a maximizing level, such as that of node A, alpha is set to be the value of the best successor that has yet been found. (Notice that although at maximizing levels it is beta that is used to determine cutoffs, it is alpha whose new value can be computed. Thus, at any level, USE-THRESH will be checked for cutoffs and PASS-THRESH will be updated to be used later.) But if the maximizing node is not at the top of the tree, we must also consider the alpha value that was passed down from a higher node. To see how this works, look again at Fig. 12.5 and consider what happens at node F. We assign the value 0 to node I on the basis of examining node K. This is so far, the best successor of F. But from an earlier exploration of the subtree headed by B, alpha was set to 3 and passed down from A to F. Alpha should not be reset to 0 on the basis of node I. It should stay as 3 to reflect the best move found so far in the entire tree. Thus, we see that at a maximizing level, alpha should be set to either the value it had at the next-highest maximizing level or the best value found at this level, whichever is greater.

The corresponding statement can be made about beta at minimizing levels. In fact, what we want to say is that at any level, PASS-THRESH should always be the maximum of the value it inherits from above and the best move found at its level. If PASS-THRESH is updated, the new value should be propagated both down to lower levels and back up to higher ones so that it always reflects the best move found anywhere in the tree.

At this point, we notice that we are doing the same thing in computing PASS-THRESH that we did in MlNlMAX to compute BEST-SCORE. We might as well eliminate BEST-SCORE and let PASS-THRESH serve in its place.

With these observations, we are in a position to describe the operation of the function MINlMAX A-B, which requires four arguments, Position, Depth, Use-Thresh, and Pass-Thresh. The initial call, to choose a move for PLAYER-ONE from the position CURRENT, should be

MINIMAX-A-B(CURRENT, 0, PLAYER-ONE, maximum value STATIC can compute, minimum value STATIC can compute) These initial values for Use-Thresh and Pass-Thresh represent the worst values that each side could achieve.

5.12 Minmax AB algorithm

MINIMAX-A-B (Position, Depth, Player, Use-Thresh, Pass-Thresh)

1. If DEEP-ENOUGH (Position, Depth) then return the structure
VALUE=STATIC (Position, Player);
PATH=nil
2. Otherwise, generate one or more ply of the tree by calling the function MOVE-GEN(Position, Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, there are no moves to be made; return the same structure that would have been returned if DEEP-ENOUGH had returned TRUE.
4. If SUCCESSORS is not empty, then go through it, examining each element and keeping track of the best one. This is done as follows.

For each element SUCC of SUCCESSORS:

a) Set RESULT-SUCC to MINIMAX-A-B (SUCC, Depth+1, OPPOSITE (Player, -Pass-Thresh, -Use thresh).
b) Set NEW-VALUE to –VALUE (RESULT-SUCC).
c) If NEW-VALUE > Pass-Thresh, then we have found a successor that is better than any that have been examined so far. Record this by doing the Following.
    i. Set Pass-Thresh to NEW-VALUE.
    ii. The best-known path is now from CURRENT to SUCC and then on to the appropriate path from SUCC as determined by the recursive call to MINIMAX-A-B. So, set BEST-PATH to the result of attaching SUCC to the front of PATH (RESULT-SUCC).
d) If Pass-Thresh (reflecting the current best value) is not better than Use-Thresh, then we should stop examining this branch. But both Thresholds and values have been inverted. So, if Pass-Thresh >=Use-Thresh, then return immediately with the value
VALUE=Pass-Thresh
PATH=BEST-PATH

5. Return the structure

VALUE =Pass-Thresh
PATH=BEST-PATH

6. Source Code Implementation

Program Description

The project is divided into many modules or classes. Each class has a purpose. All the relevant functions are put together into one class. The functions prototypes or the declarations are given in the header files of each class. The definitions of all the functions are present in the implementation file of the class. Other than these class files, there is a main.cpp file which brings all the classes together. Other than that, the main.cpp file contains few extra functions which basically call the functions in the class files. We have a display function also which is called in the main.cpp file. This display function displays all the necessary results. In the project description, it was mentioned that the output should have few information like which player is the winner and how many nodes were generated in total. The program is explained below in detail going through each class file.

Below are the list of files of this project.

- Main.cpp
- AlphaBeta.h
- AlphaBeta.cpp
- MinMaxAB.h
- MinMaxAB.cpp
- Kalah.h
- Kalah.cpp
- KalahFlow.h
- KalahFlow.cpp

Main.cpp

Here the control of the program begins with a menu driven program as follows:

Initial position of the Kalah Boad.

The implemented program description is as follows:

The whole project is split between the above mentioned files. Each class has its own purpose. In the main, we have multiple options from which the user can choose depending on what has to be displayed. A switch case is used to execute this. There is a display function in the main.cpp file which displays all the necessary information like Initial Board position

There are many other functions defined in the main.cpp file which are triggered from the option the user chooses from the main menu (displayed in the output screen). The functions in the main.cpp file then call the functions from the other class file. Evaluation functions are passed to the MinMaxAB and AlphaBetaSearch functions. There is another moveDecider function which checks if the player can move and move each player.

In the main.cpp file, we have functions which call MinMaxAB and AlphaBetaSearch functions in different orders, just like mentioned in the menu. When the user chooses an option from the menu, the appropriate function is called. The functions here are not inside the main function but outside the main function and inside the main.cpp file.

Only one function is called inside the main function and that is the display() function. This menu function displays the different options from which the user can choose.

Main function displaying kalah Board initial position

```
int main()
{
    int choice, selectedDepth;
    Kalah *kalah = new Kalah();

        cout << endl << endl;
        cout << "*****************************************************************************************************************" << endl;
        cout << "                                          YOU ARE NOW IN THE KALAH GAME ZONE!                                    " << endl ;
        cout << "*****************************************************************************************************************" << endl;
        cout << endl;
        cout << "                                    The game rules are as below:                                 " << endl << endl;
        cout << " ** Kalah is played by two players on a board with two rows of 6 holes facing each other." << endl << endl;
        cout << " ** It has two KALAHS (Stores for each player.)" << endl << endl;
        cout << " ** The stones are called beans. At the beginning of the game, there are 6 beans in every hole. The KALAHS are empty." << endl << endl;
        cout << " ** The object of Kalah is to get as many beans into your own kalah by distributing them." << endl << endl;
        cout << " ** A player moves by taking all the beans in one of his 6 holes and distributing them counterclockwise," << endl;
        cout << "    by putting one bean in every hole including his, but excluding the opponent's kalah." << endl << endl;
        cout << " ** There are two special moves:" << endl << endl;
        cout << " ** Extra move: If the last bean is distributed into his own kalah, the player may move again. " << endl
             << "    He has to move again even if he does not want to." << endl << endl;
        cout << " ** Capture: If the last bean falls into an empty hole of the player and the opponent's hole above (or below) is not empty," << endl
             << "    the player puts his last bean and all the beans in his opponent's hole into his kalah. He has won all those beans." << endl << endl;
        cout << " ** The game ends if all 6 holes of one player become empty (no matter if it is this player's move or not)." << endl
             <<"     The beans in the other player's holes are given into this player's kalah." << endl << endl;
        cout << " ** The player who won more beans (at least 37) becomes the winner." << endl << endl;

        cout << " *****************************************************************************************************************" << endl;
        cout << "                                          ***KALAH BOARD DISPLAY***                                              " << endl ;
        cout << " *****************************************************************************************************************" << endl;

        kalah->display();

        cout << " You now have the below choices :" << endl << endl
             << "Choose your option :" << endl << endl;

        cout << " ** Choose 1 for MinMaxAB: " << endl;
        cout << " ** Choose 2 for AlphaBetaSearch: " << endl;
        cout << " ** Choose 3 to QUIT:" << endl;
        cin >> choice;

        cout << "Please enter the depth that you would want to check the results for: The allowed depths are 2 and 4:" << endl;
        cin >> selectedDepth;

        while(selectedDepth !=2 && selectedDepth !=4)
        {
            cout << "Invalid Depth! Please enter depth value 2 or 4!" << endl;
            cin >> selectedDepth;
```

User has to choose from this menu and an appropriate function will be called in the main.

When the user choice is 1(MinMax A B), function main() will execute case 1, in the switch. Below is the screenshot.

```
switch(choice)
{
    case 1:
        {
            cout << "Initial board " << endl;
            kalah->display();

            char result = kalah->result();

            char player ='A';
            int start_timer = clock();

            while(result == 'N')
            {

                KalahFlow *min_max_ab = new KalahFlow(player);

                min_max_ab->current_player_status( kalah );

                cout << "It is player " << player << "'s turn!" << endl;

                MinMaxAB(min_max_ab,0,player,2000,-2000,selectedDepth);

                int pit = min_max_ab->heuristic;

                if(player == 'A')
                    player = kalah->move_generator_A(pit);

                else
                    player = kalah->move_generator_B(pit);

                kalah->display();

                result = kalah->result();

                game_path_length++;
            }

            int stop_timer = clock();
            kalah->display();
            int total_time = stop_timer - start_timer;
            winner(result, total_time, kalah );
        }

        break;

    case 2:
        {
            cout << "Initial board " << endl;
```

When the user choice is 2(AlphaBetaSearch), function main() will execute case 2, in the switch. Below is the screenshot.

```
884
885             case 2:
886                 {
887                     cout << "Initial board " << endl;
888                     kalah->display();
889
890                     char result = kalah->result();
891
892                     char player = 'A';
893                     int start_timer = clock();
894
895                     while(result == 'N')
896                     {
897
898                         KalahFlow *alpha_beta_flow = new KalahFlow(player);
899
900                         alpha_beta_flow->current_player_status(kalah);
901
902                         cout << "It is player " << player << "'s turn!" << endl;
903
904                         AlphaBetaSearch(alpha_beta_flow,0,player,1000,-1000, selectedDepth);
905
906                         int pit;
907
908                         for(int i = 0; i < SIZE; i++)
909                         {
910                             if(alpha_beta_flow->successors[i] == NULL)
911                                 continue;
912
913                             if(alpha_beta_flow->successors[i]->heuristic == alpha_beta_flow->heuristic)
914                                 pit = i;
915                         }
916                         if(player == 'A')
917                             player = kalah->move_generator_A(pit);
918
919                         else
920                             player = kalah->move_generator_B(pit);
921
922                         kalah->display();
923
924                         result = kalah->result();
925
926                         game_path_length++;
927                     }
928
929                     int stop_timer = clock();
930                     kalah->display();
931                     int total_time = stop_timer - start_timer;
932                     winner(result,total time, kalah);
```

When the user choice is 3(Exit), function main() will execute case 3, in the switch. Below is the screenshot.

```
929                     int stop_timer = clock();
930                     kalah->display();
931                     int total_time = stop_timer - start_timer;
932                     winner(result,total_time, kalah);
933                 }
934
935                 break;
936
937             case 3:
938                 cout <<"Thanks. See you again later....." << endl;
939                 return 0;
940         }
941
942         display_PerfomanceAnalysis();
943
944     return 0;
945 }
946
```

Each option is well described and could be easily understood by the users. Below screenshot shows how menu looks like.
For all the above mentioned cases display _PerformanceAnalysis() function will be executed which displays the who the winner is, the number of nodes generated, the number of nodes expanded, the total time taken and the memory it used(as shown in the below screenshot).

```
**********************************************************************************************************
                                                ***GAME OVER***
**********************************************************************************************************


The program took '0.439' seconds of Execution time for the completed game!

1 node takes 81 bytes of Memory

The algorithm takes : '40338' bytes = '39' kb of memory!

The program generated '498' nodes for the completed game!

The program expanded '142' nodes for the completed game!

The total length of the Game path is '28' for the completed game!


Process returned 0 (0x0)   execution time : 21.686 s
Press any key to continue.
```

AlphaBeta.h and AlphaBeta.cpp

AlphaBetaSearch(): Alpha Beta search algorithm implementation for Kalah Game. The AlphaBeta.h file just contains the declaration of the alphabeta function. Other class files have been included in this header file. The AlphaBeta.cpp file, there is only one function defined. This function follows the algorithm (from the textbook) mentioned in the above. Evaluation functions are called inside this function. An integer is passed as a parameter to this function along with other parameters. This integer parameter basically decides which evaluation function should be called.

```
int AlphaBetaSearch(KalahFlow *alphabeta, int depth, char player, int alpha, int beta, int selected, int evaluation_choice)
{

    int bestValue = -100;
    int value;

    if(alphabeta->is_deep_enough(depth, selected))
    {
        if(evaluation_choice == 1)
            return alphabeta->myEvaluationFunction1();

        else if (evaluation_choice == 2)
            return alphabeta->myEvaluationFunction2();

        else if (evaluation_choice == 3)
            return alphabeta->myEvaluationFunction3();

        else
            return alphabeta->myEvaluationFunction4();
    }
```

parameters used: an object of type KalahFlow, the current depth which is 0 during the initial call, the current player, which was 'A' initially, alpha, beta, selectedDepth, which can be 2 or 4, the maximum depth, evaluation_choice 1, 2 , 3 or 4 select from three possible evaluation function options.

This program contains 4 evaluation functions. This function returns the 'maxvalue' which basically helps decide which move to make. Function returns the returns the best value.

```
    }

    alphabeta->heuristic = bestValue;

    return bestValue;
```

MinMaxAB.h and MinMaxAB.cpp

The MinMaxAB algorithm is different from the alpha beta search algorithm. Instead of referring to alpha and beta, MinMaxAB uses two values, USE-THRESH and PASS-THRESH. USETHRESH is used to compute cutoffs. PASS-THRESH is merely passed to the next level as its USETHRESH. Of course, USE-THRESH must also be passed to the next level, but it will be passed as PASS-THRESH so that it can be passed to the third level down as USE-THRESH again, and so forth.

In our program we have 'use_Threshold' and 'pass_Threshold' for this purpose. At a maximizing level , such as that of node A, alpha is set to be the value of the best successor that has yet been found. At any level, 'use_Threshold' will be checked for cutoffs and 'pass_Threshold' will be updated to be used later. But if the maximizing node is not at the top of the tree, we must also consider the alpha value that was passed down from a higher node.

The algorithm is explained in the Methodology section. The MinMaxAB.h file just contains the declaration of the MinMaxAB function. Other class files have been included in this header file. The MinMaxAB.cpp file, there is only one function defined. This function follows the algorithm mentioned in the above.

MinMaxAB(): MinMAxAB search algorithm implementation for Kalah Game.
parameters used are an object of type KalahFlow, the current depth which is 0 during the initial call, the current player, which was 'A' initially, Use-Threshold, Pass-Threshold value, selectedDepth, which can be 2 or 4, the maximum depth, evaluation_choice-1, 2 or 3 – to select from three possible evaluation function options.

Evaluation functions are called inside this function. An integer is passed as a parameter to this function along with other parameters. This integer parameter basically decides which evaluation function should be called. This program contains 4 evaluation functions.

```
int MinMaxAB(KalahFlow *min_max,int depth, char player, int use_Threshold, int pass_Threshold, int selectedDepth, int evaluation_choice)
{
    int structure;
    int new_value;
    char new_Player;
    int result_successor = 0;

    if(min_max->is_deep_enough(depth, selectedDepth))
    {

        if(evaluation_choice == 1)
            structure = min_max->myEvaluationFunction1();

        else if (evaluation_choice == 2)
            structure = min_max->myEvaluationFunction2();

        else if (evaluation_choice == 3)
            structure = min_max->myEvaluationFunction3();

        else
            structure = min_max->myEvaluationFunction4();
```

Funtion returns the best value.

KalahFlow.h and kalahFlow.cpp

```
class KalahFlow
{

public:

    char player;
    Kalah current_status;
    int heuristic;
    KalahFlow * successors[SIZE];
    int successor_count;

    KalahFlow(char);
    void current_player_status(Kalah);
    bool is_deep_enough(int, int);
    void generate_successors();
    int myEvaluationFunction1();
    int myEvaluationFunction2();
    int myEvaluationFunction3();
    int myEvaluationFunction4();

};
```

The header file of the Class KalahFlow contains the definition of the Kalah board.

Board has the variables player, heuristic, succesoor count. It contains other variable called current_status of type Kalah.
All these variables give the pit number in the Kalah board just infront of the player and behind the player. There are many functions defined. All the functions are declared in the header file and defined in the implementation file. There are many functions in this class.

KalahFlow() is A parameterized constructor that is used to initialize an object of class KalahFlow. parameters used is player(A or B).

```
KalahFlow::KalahFlow(char player)
{
    this->player = player;
    heuristic = -2000;
    successor_count = 0;

    for(int i = 0; i < SIZE; i++)
        successors[i] = NULL;
}
```

current_player_status():
Function copies the current status of the player into the current_status object of type Kalah, Parameter used is Kalah k .

```
void KalahFlow::current_player_status(Kalah k)
{
    for(int i = 0; i < SIZE; i++)
    {
        this->current_status.Player_A[i] = k.Player_A[i];
        this->current_status.Player_B[i] = k.Player_B[i];
    }

    this->current_status.store_A = k.store_A;
    this->current_status.store_B = k.store_B;
}
```

generate_successors():
Function generates successors(child node) when the deep enough function is satisfied to evaluate the future moves of the opponent based on the present state of the board. This root node has child branches which ends in the heuristic value of the end leaf node. Here the leaf node is the value of number of stones in the pit at a given board state.

```
bool KalahFlow::is_deep_enough(int depth, int selected)
{
    if(heuristic != -2000)
        return heuristic;

    if(depth >= selected || current_status.result() != 'N' )
        return true;

    else
    {
        expanded_node_count++;
        generate_successors();
        return false;
    }
}
```

Kalah.h and Kalah.cpp

The header file of the Kalah class contains the definition of the kalah, move_decider and move_generator. Kalah class has the variables store_A, store_B, slots, maxWinNumber, pointer, array of Player_A , Player_B are the pits of respective player and the function definitions which are explained next.

```
class Kalah
{
    public:
    int Player_A[SIZE],
        Player_B[SIZE],
        store_A,
        store_B,
        slots,
        maxWinNumber;

    int *pointer;

    Kalah();
    Kalah(Kalah *);
    void display();
    void operator=(Kalah);
    char move_decider(int, char);
    char move_generator_A(int);
    char move_generator_B(int);
    char result();

};
```

Kalah(): A default constructor that initializes all the data members of the Kalah Class. Each pit of Player A and Player B are initialized to having 6 stones stores of both players initialized to 0 max number of stones that a person needs to be considered as having a winning chance is assigned to 3.

```
Kalah::Kalah()
{
    for(int i = 0 ; i < SIZE; i++)
    {
        Player_A[i] = 6;
        Player_B[i] = 6;
    }

    store_A = store_B = 0;
    slots = 5;

    pointer = NULL;
    maxWinNumber = 36;
}
```

Kalah(): is a copy constructor that is used to initialize an object of class Kalah with another object. Copies the kalah value from one object to another

```
Kalah::Kalah(Kalah *b)
{
    for(int i = 0; i < SIZE; i++)
    {
        this->Player_A[i] = b->Player_A[i];
        this->Player_B[i] = b->Player_B[i];
    }

    this->store_A = b->store_A;
    this->store_B = b->store_B;
}
```

dislay(): This function displays the Kalah board

```
void Kalah::display()
{
    cout << endl;
    cout << "\t\tPLAYER A";
    cout << endl << endl;
    cout << " \t | ";

    for(int i = 0 ; i < SIZE; i++)
        cout  << i << setw(2) << "|" << " ";

    cout << "---------------> Pit Numbers of A" << endl;
    cout << "\t============================";
    cout << endl;
    cout << "\t" << "|";
    cout << setw(2);


    for(int i = (SIZE - 1) ; i >= 0 ; i--)
        cout << Player_A[i] << setw(2) << "|" << " ";

    cout << endl;
    cout << "\t============================";

    cout << endl;
    cout << "store_A" << "\t\t\t\t   store_B" <<endl;
    cout << "------"  << "\t\t\t\t   ------";
    cout << endl;
    cout <<"|" << setw(3) << store_A << " |"; // Fourth Line
    cout <<"\t\t\t\t   |" << setw(3) << store_B << " |" << endl;
    cout << "------"  << "\t\t\t\t   ------";
    cout << endl;
    cout << "\t============================";
    cout << endl;
    cout << "\t" << "|";
    cout << setw(2);

    for(int i = 0 ; i < SIZE; i++)
        cout << Player_B[i] << setw(2) << "|" << " ";

    cout << endl;
    cout << "\t============================";
    cout << endl;
    cout << " \t | ";

     for(int i = 0 ; i < SIZE; i++)
        cout  << i << setw(2) << "|" << " ";
```

ads\fullchangedfile (3).cpp

Display function will display the kalah board according to changes made at each node.

```
**************************************************************************
                                        ***KALAH BOARD DISPLAY***
**************************************************************************

              PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | --------------> Pit Numbers of A
       ========================
       | 6 | 6 | 6 | 6 | 6 | 6 |
       ========================
store_A                         store_B
------                          ------
|  0 |                          |  0 |
------                          ------
       ========================
       | 6 | 6 | 6 | 6 | 6 | 6 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | --------------> Pit Numbers of B

              PLAYER B
```

Move_decider(): This function decides which player has to play. Ands calls the move_denerator function accordingly. We have move_generator_A and move_generator_B function based on the current position of the pit and the number of stones in the pit. We also have to consider the opponents position as well as opponents stone accumulated. We also consider the stones as temporary position for comparison. We also evaluate who should make the next move based on the function.

```
char Kalah::move_decider(int hole_num,char player)
{
    if(player == 'A')
        return (move_generator_A(hole_num));
    else
        return (move_generator_B(hole_num));
}
```

operater=(Kalah k): This is an operator overloading function that overloads the '=' Object of type Kalah is used as a parameter.

```
void Kalah::operator=(Kalah k)
{
    for(int i = 0; i < SIZE; i++)
    {
        Player_A[i] = k.Player_A[i];
        Player_B[i] = k.Player_B[i];

    }

    store_A = k.store_A;
    store_B = k.store_B;
    slots = 5;
    pointer = NULL;
    maxWinNumber = 36;

}
```

move_generator_A(): This function helps generate moves for Player A. If last stone lands in Player A's store, Player A gets another chance. If player A's stone lands in an empty hole in Player A's holes, player A captures all player B's stone from the opposite hole. If the above two special scenarios are not encountered, the moves are normal. Which uses parameter as Hole number from which the player should start playing and returns the next player that gets a chance after A's move. In most cases, B gets a turn. When A's stone lands in its store, A is returned, meaning, A gets an extra move

```
char Kalah::move_generator_A(int hole_num)
{
    int current_Position = hole_num;
    pointer = Player_A;

    int stoneCount = pointer[current_Position];

    pointer[current_Position] = 0;
    int opPosition, opStones;

    current_Position++;

    while(stoneCount > 0)
    {
        //last stone lands in Player A's store
        if(stoneCount >= 1 && current_Position == 6 )
        {
            stoneCount--;
            store_A ++;

            if(stoneCount == 0)
                return 'A'; // giving A another chance as last stone falls in A'
        }

        else if(current_Position >= 0 && current_Position <= 5)
        {
            if(stoneCount == 1)
            {
                stoneCount--;

                opPosition = slots - current_Position;

                if(pointer[current_Position] == 0 && Player_B[opPosition] > 0)
                {
                    pointer = Player_B;
                    opStones = pointer[opPosition];
                    pointer[opPosition] = 0;

                    store_A += opStones + 1;

                    if(stoneCount == 0)
                        return 'B';
                }

                else
                {
                    pointer[current_Position]++;
```

move_generator_B(): This function helps generate moves for Player B. If last stone lands in Player B's store, Player B gets another chance. If player B's stone lands in an empty hole in Player B's holes, player B captures all player A's stone from the opposite hole. If the above two special scenarios are not encountered, the moves are normal. Which uses parameter Hole number from which the player should start playing and returns the next player that gets a chance after B's move. In most cases, A gets a turn. When B's stone lands in its store, B is returned, meaning, B gets an extra move.

```
char Kalah::move_generator_B(int hole_num)
{
    int current_Position = hole_num;
    pointer = Player_B;

    int stoneCount = pointer[current_Position];

    pointer[current_Position] = 0;
    int opPosition, opStones;

    current_Position++;

    while(stoneCount > 0)
    {
        //last stone lands in Player B's store
        if(stoneCount >= 1 && current_Position == 6)
        {
            stoneCount--;
            store_B++;

            if(stoneCount == 0)
                return 'B'; // giving B another chance as last stone falls in B's store
        }

        else if(current_Position >= 0 && current_Position <= 5)
        {
            if(stoneCount == 1)
            {
                stoneCount--;

                opPosition = slots - current_Position;

                if(pointer[current_Position] == 0)
                {
                    pointer = Player_A;
                    opStones = pointer[opPosition];
                    pointer[opPosition] = 0;

                    store_B += opStones + 1;

                    if(stoneCount == 0)
```

result(): This function checks the result of the game. If A has more than 36 stones, A wins. If B has more than 36 stones, B wins and returns A if Player A wins, B if player B wins, N if no win yet.

```
char Kalah::result()
{
    int empty_pits_A = 0,
        empty_pits_B = 0;

    for(int i = 0; i < SIZE; i++)
    {
        if(Player_A[i]==0)
            empty_pits_A++;

        if(Player_B[i]==0)
            empty_pits_B++;
    }

    if(empty_pits_A == 6)
    {
        for(int i = 0 ; i < SIZE; i++)
        {
            store_B += Player_B[i];
            Player_B[i] = 0;
        }

        cout << "Player A ran out of all the stones! All the pits in A are empty! " << endl;
    }

    if( empty_pits_B == 6)
    {
        for(int i = 0 ; i < SIZE ; i++)
        {
            store_A += Player_A[i];
            Player_A[i] = 0;
        }

        cout << "Player B ran out of all the stones! All the pits in B are empty! " << endl;
    }

    //if Player A has more than 36 stones, Player A wins
    if(store_A > maxWinNumber)
        return 'A';
```

7.Evaluation Functions

A computer game system mainly includes four parts, that are the board representation, the move generation, search algorithm, and the situation assess. The board representation describes the game situation, including the encoding and storage of the game board, players and obstacles. The move generation refers to the generation of the optimal path from the current path. As the core part of the computer game, the search algorithm is to find the optimal moving, the common algorithms include the Minimax algorithm, the Alpha Beta pruning algorithm and etc. The situation evaluation access the current game situation, provide the premise for the search algorithm, thus it is important to the performance of the whole system.

Evaluation function is the heuristic to give a number to a particular state of the program. These points for each state then helps the algorithms to decide the next move of the player. In the game of Kalah, heuristic evaluation function is applied to the kalah board and it returns a number which would give the strength of that move compared to the opponent. The primary reason for the use of a heuristic evaluation function is to reduce the amount of computation required to solve a problem.

A good evaluation function should order the terminal states in the same way as the true utility function and, for non terminal states, should be strongly correlated with the actual chances of winning.
Most evaluation functions work by calculating various features of a given state. Taken together, these features define categories of states that have the same values for all features. Any given category will have states that lead to each of the various outcomes. The evaluation function returns a value that reflects the proportion of states with each outcome. The weighted value of these proportions is called the expected value and is used for evaluation purposes.

Characteristics of a good evaluation function:

1. It should agree with the utility function in the on the terminal states.
2. The calculation for an evaluation function should not be too long.
3. An evaluation function should accurately reflect the actual chances of winning.

Each team member had to come up with evaluation function. All 4 evaluation functions have been given below.

Evaluation function1: Vidhyashree

```
int KalahFlow::myEvaluationFunction1()
{
    int terminal_value;

    if(player == 'A')
    {
        int stones_of_A = 0;

        for(int i = 0 ; i < 6 ; i++)
        {
            if(current_status.Player_A[i] == 0)
                stones_of_A++;
        }

        if(stones_of_A == 6)
            terminal_value = 2000;
    }
    else if(player == 'B')
    {
        int stones_of_B = 0;

        for(int i = 0 ; i < 6 ; i++)
        {
            if(current_status.Player_B[i] == 0)
                stones_of_B++;
        }
        if(stones_of_B == 6)
            terminal_value = -2000;
    }

    heuristic = terminal_value;

    return terminal_value;
}
```

In the below evaluation function, we have considered the case where the pits of A or B are empty. If all pits in A is 0, we assign a value of 2000 to the terminal node. If all pits in B are empty, we assign -2000 to the terminal node. We are maximizing the chances of A winning and minimizing the chances of B winning.

Evaluation function2: Akshatha Jain
I came up with 2 evalation functions.

```
int KalahFlow::myEvaluationFunction2()
{

int terminal_value;
    if(player == 'A')
        terminal_value = current_status.store_A - current_status.store_B;

    else if(player == 'B')
        terminal_value = current_status.store_A - current_status.store_B;

    heuristic = terminal_value;

    return terminal_value;

}
```

myEvaluationFunction2() (1st found this evaluation function)
This evaluation function checks whether Player A or player B is playing. Then it will calculate the terminal value for the respective players. Terminal value is the amount of difference between stones at store_A(PlayerA's total stones in the storeA till this move) and store_B(PlayerB's total stones in the storeB till this move). As our program always 1st gives chace to Player A, It is always guareented that PlayerA or the Player who plays 1st will win the game according to the algorithms.

So, I assumed Player A's store will consists of more number of stones So I calculated a difference assuming that heuristic will get the best value.
In this way heuristic will get the maximum possible value. Function returns the terminal value which is an interger.

I got an idea to create another evaluation while developing this. I have attached evaluation3 in the next page. Which actually works a way lot better in terms of time and memory.

Evaluation functions utilizes linear time complexity of O(1).

Evaluation function3: Akshatha Jain(Best one)

```
int KalahFlow::myEvaluationFunction3()
{
    int p0 = current_status.store_A;
    int p1 = current_status.store_B;
    int terminal_value;

    if(player == 'A')
    {
        if(p0 != 0)
            terminal_value = (current_status.store_A/current_status.store_B);

        else
            return -1;
    }

    else if(player == 'B')
    {
        if(p1 !=0)
            terminal_value = (current_status.store_B/ current_status.store_A);

        else
            return -1;

    heuristic = terminal_value;
    return terminal_value;
}
}
```

This evaluation function checks whether Player A or player B is playing. It has 2 Interger variables

1. P0 : which gets the number of stones at store_A
2. P1: which gets the number of stones at store_B

Function will check whether the po or p1 is not equal to zero(To explain this mathematically if the value of po or p1 is equal to 0. Then 0 divide by anything will be zero). So, zero cannot be the best value or maximum value. If this condition is satisfied it calculates the terminal value.

Also got a chance to learn how important an evaluation function is and how different evaluation function makes the algorithm take a different route. We also got to learn on how to make an evaluation function. Each of the evaluation function displayed a particular behavior in terms of the number of nodes generated to determine a winner. From the gathered results, we were able to identify which evaluation function worked better and gave us a better performance. We compared both the algorithms with each of the evaluation functions

I can say This evaluation function better compared to all the evaluation both in terms of time and memory. So I will consider this evaluation function for explanation in future. As my evaluation function will

Evaluate the current board Calculate efficiently and always returns the same value for the same board Terminal node values are given precisely – maximal positive value for max player and maximal negative value to the opponent playerIt also correlates the score for the current player to win with best moves – assuming the max player will not make any wrong moves or mistakes to let the opponent win

Otherwise the evaluation function returns -1 and exits**. Evaluation function utilizes linear time complexity of O(1).**

Evaluation function4: Sahana

```
int KalahFlow::myEvaluationFunction4()
{
    int terminal_value;
    if (player == 'A'){
        int stones_of_A = 0;
        for(int i=0; i<6; i++) {
            if (current_status.Player_A[i]==0) stones_of_A++;
            else if(current_status.Player_A[i]==i) terminal_value = 1500;
            else if(current_status.Player_A[i]<i) terminal_value = 800;
            else
                terminal_value = 500;
        }
        if(stones_of_A == 6)
            terminal_value = 1000; }
    if (player == 'B'){
        int stones_of_B = 0;
        for(int i=0; i<6; i++) {
            if (current_status.Player_A[i]==0) stones_of_B++;
            else if(current_status.Player_A[i]==i) terminal_value = 1500;
            else if(current_status.Player_A[i]<i) terminal_value = 800;
            else
                terminal_value = 500;
        }
        if(stones_of_B == 6)
            terminal_value = 1000; }
    heuristic = terminal_value;
    return terminal_value;

}
```

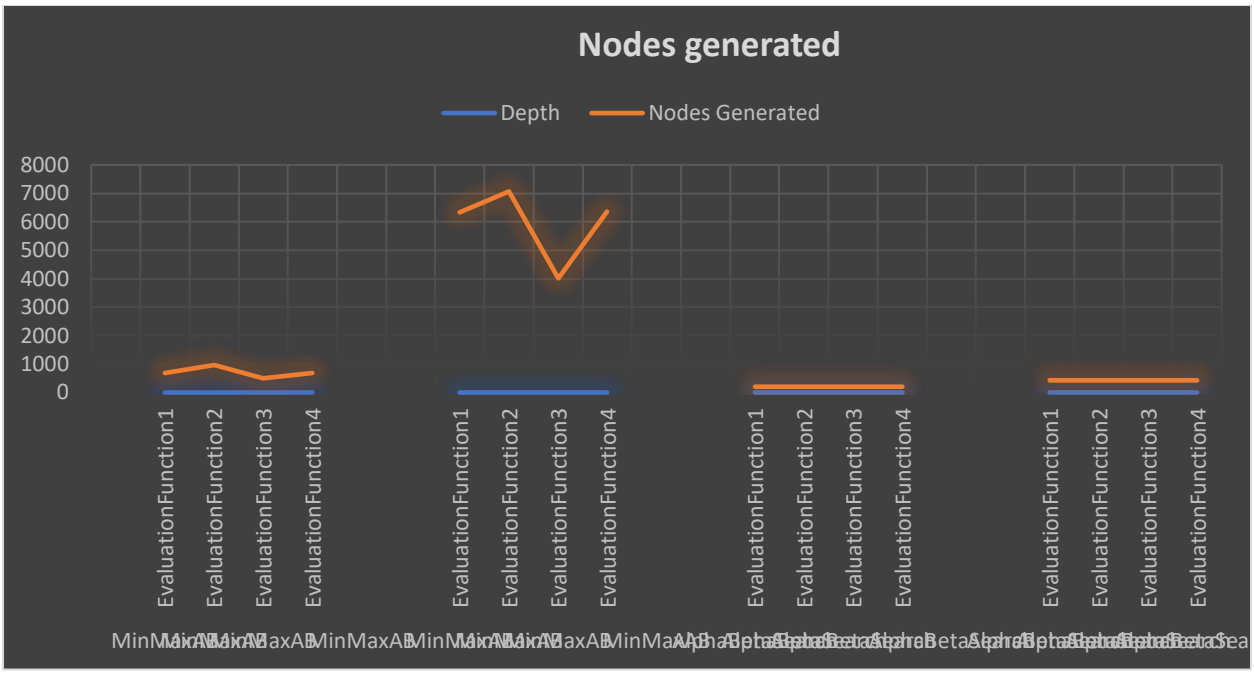
The evaluation function Sahana decided to implement was to initially set the value of the board which is passed by reference to this function, we then check if the player is max player or a min player, the number of stone initially in the current board state which is stored in an array. We define a heuristic value to the terminal node to find the best value for alpha and beta, the value is compared and best value is allocated, this heuristic value is returned uphill the root node.

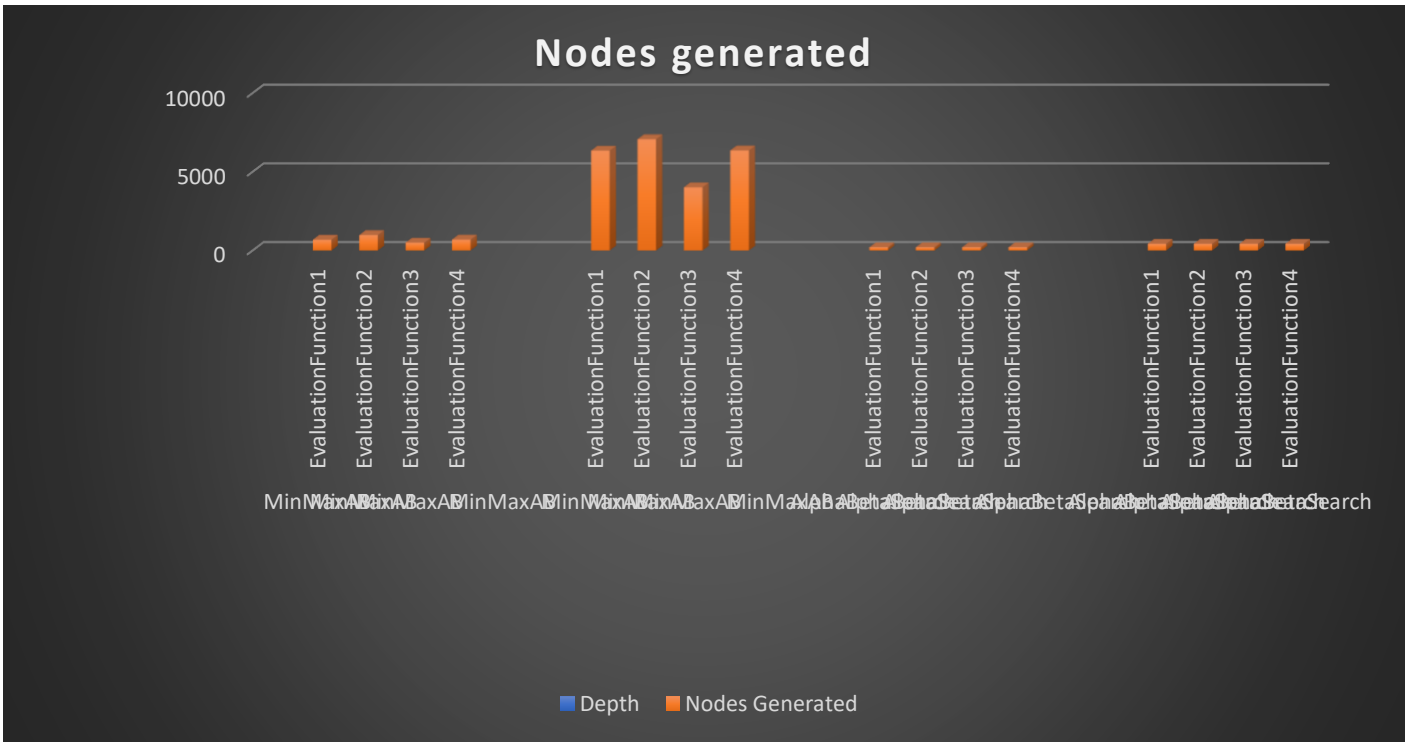
evaluation function will :

1. Evaluate the current board
2. Calculate efficiently and always returns the same value for the same board
3. Terminal node values are given precisely – maximal positive value for max player and maximal negative value to the opponent player
4. It also correlates the score for the current player to win with best moves – assuming the max player will not make any wrong moves or mistakes to let the opponent win(Ideal Situation)

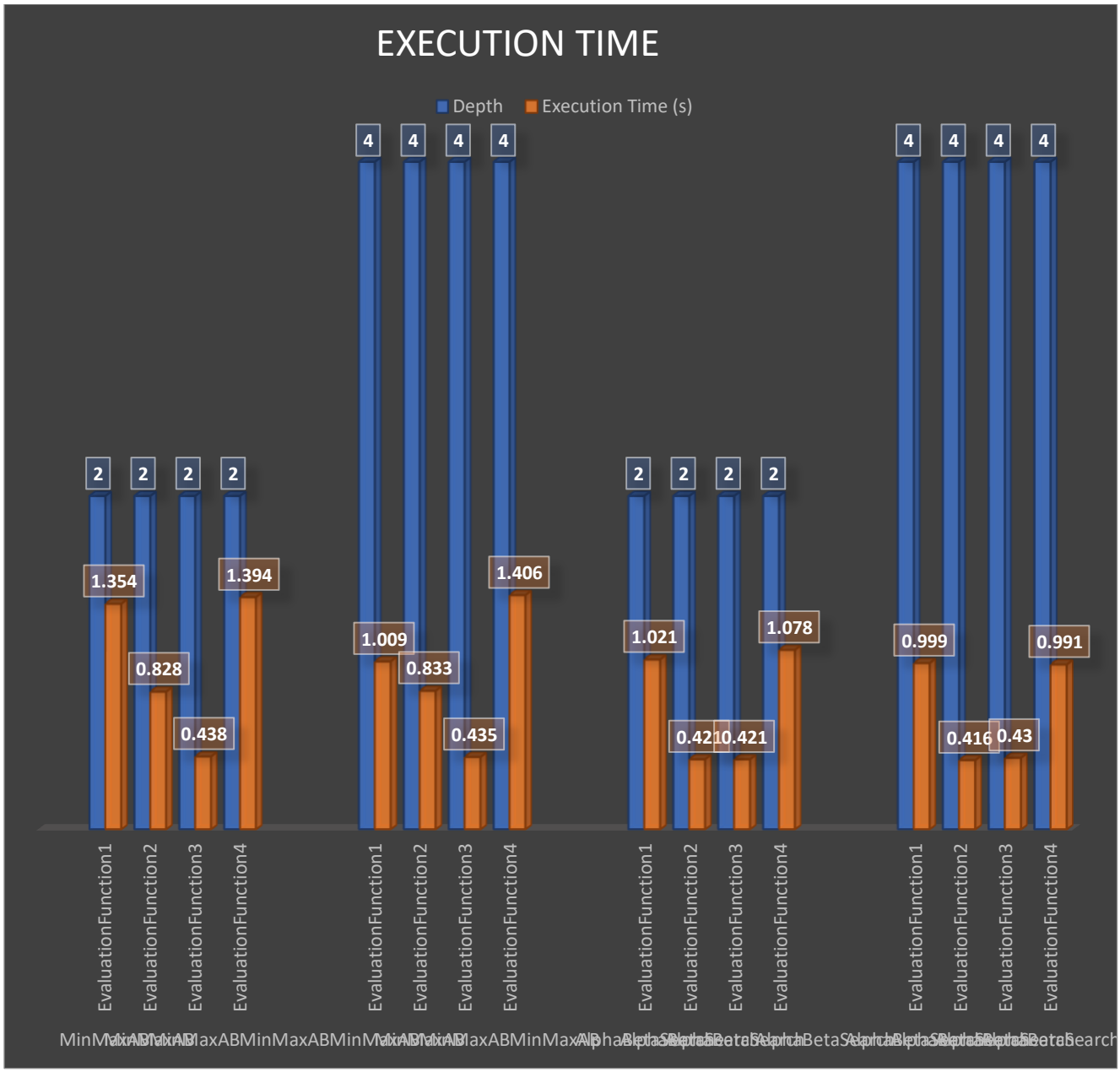
8.Analysis of the Results

Algorithm	Evaluation Function	Depth	Nodes Generated	Nodes Expanded	Total Path Length	Execution Time (s)	Memory Consumed (KB)
MinMaxAB	EvaluationFunction1	2	687	178	35	1.354	54
MinMaxAB	EvaluationFunction2	2	979	278	54	0.828	77
MinMaxAB	EvaluationFunction3	2	498	142	28	0.438	39
MinMaxAB	EvaluationFunction4	2	694	194	37	1.394	55
MinMaxAB	EvaluationFunction1	4	6342	1572	30	1.009	501
MinMaxAB	EvaluationFunction2	4	7067	1808	54	0.833	559
MinMaxAB	EvaluationFunction3	4	4013	983	28	0.435	318
MinMaxAB	EvaluationFunction4	4	6352	1588	37	1.406	502
AlphaBetaSearch	EvaluationFunction1	2	206	54	27	1.021	16
AlphaBetaSearch	EvaluationFunction2	2	206	54	27	0.421	16
AlphaBetaSearch	EvaluationFunction3	2	206	54	27	0.421	16
AlphaBetaSearch	EvaluationFunction4	2	206	54	27	1.078	16
AlphaBetaSearch	EvaluationFunction1	4	425	108	27	0.999	33
AlphaBetaSearch	EvaluationFunction2	4	425	108	27	0.416	33
AlphaBetaSearch	EvaluationFunction3	4	425	108	27	0.43	33
AlphaBetaSearch	EvaluationFunction4	4	425	108	27	0.991	33

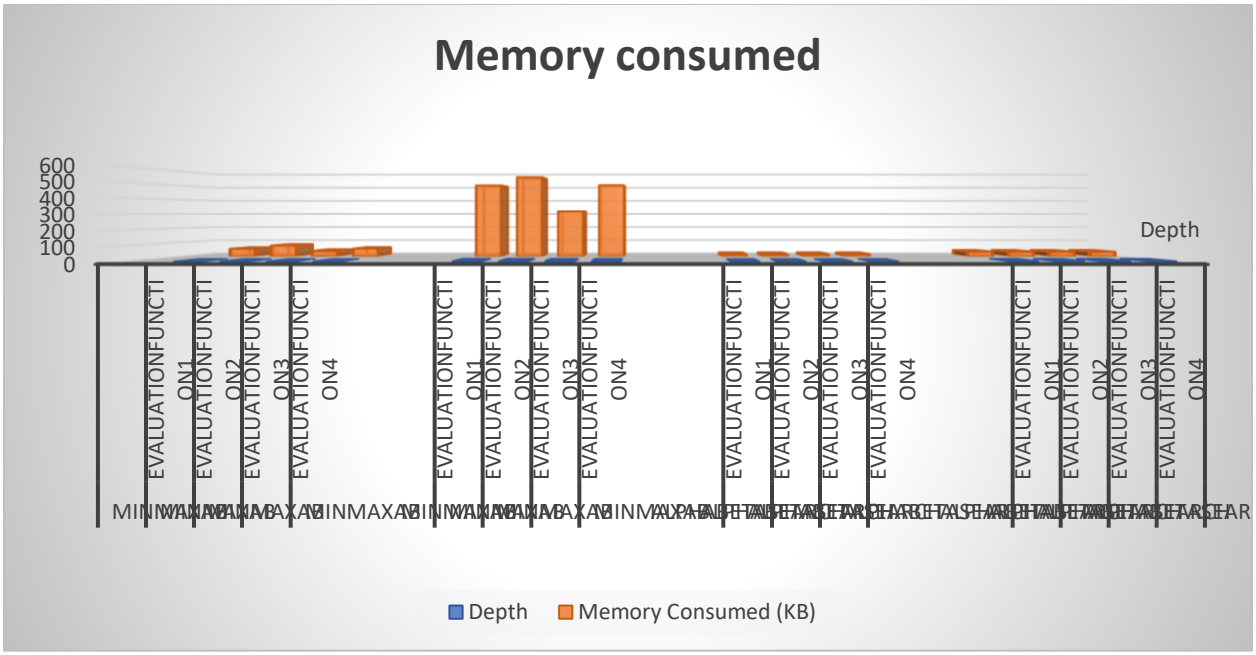




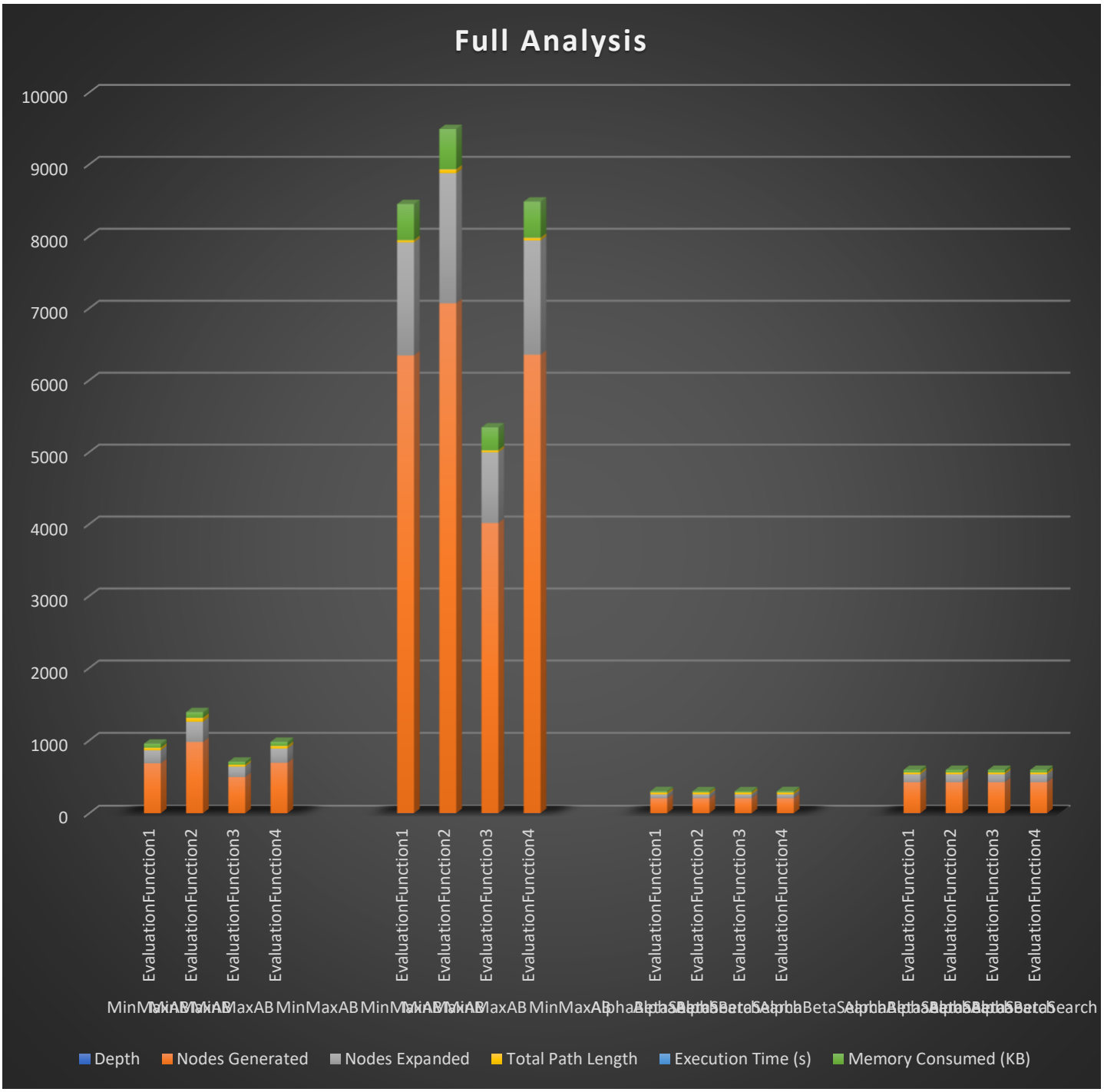
From the above 2 graphs we can say that nodes generated by evaluation function for both the algorithms at depth 2 and 4 are always less. So evaluation function3 is better compared to all others.



From the above graph we can say that for all the senarios evaluation function 3 is taking less time. So evaluation3 is better in terms of time compared to all other.



Above graph shows the memory consumption which is always less for evaluation3 comapared to all other.



Above chart shows the overall Analysis for the full execution of the program. From which we can prove that evaluation function3 is better in terms of time as it is taking less time compared to other functions.
Also it is clear that evaluation function3 generated less number of nodes compared to all other evaluation functions. Evaluation function 3 uses less memory compared to all other functions.
So overall performance of evaluation function3 is good compared to others.

The above table shows all the results we gathered by running our program. Looking at the data, we can deduce the following information.

- As mentioned earlier, we have 4 evaluation functions in this program and using each of those evaluation functions with different algorithms has given us a varied results.

- Games is played between Computer and Computer. Both of the Computers following the minimaxAB algorithm. Each time the number of nodes generated, number of nodes expanded and total length of the path were different. This was because of the different evaluation functions applied in each game.

- Choice 1 of our menu initiates the Computer vs Computer game, using the minimaxAB algorithm and the results are shown in above table.

- Choice 2 of our menu initiates the Computer vs Computer game, using the Alphabeta algorithm and the results are shown in above table.

- By seeing at the table we can say it directly that evauation function3 is the best as it uses less time to execute or linear time and the nodes generated will be less compared to all the other evaluation functions.

- It is difficult to measure the performance of the two algorithms or to decide which algorithm is better. But looking at the collected data we can deduce few very important information.
  For analysis of the results by considering evaluation function results we can say that number of nodes generated and time required for the execution is less in Alpha Beta.

We see that AlphaBetaSearch has a better performance than the MinMaxAB search algorithm. We can conclude this with an instance of result here: Consider a run for depth 2 and Evaluation Function 3 for both MinMaxAB and AlphaBetaSearch algorithms. We notice that, in this scenario, MinMaxAB generated 498 nodes, expanded 142 nodes and the total game path length was 28. For the same scenario, AlphaBetaSearch generated 206 nodes, expanded 54 nodes and the total game path length was 27. This makes us infer that AlphaBetaSearch was better.

We have used optimization techniques which reduces the computation time to 0.36ms for depth 2 and 0.86ms for depth 4. We have removed concurrent loops to avoid continuous loop calls. We have created separated function call for each logic so that it is very clear to readers and developers. We have also implemented class structure for constructors and function definition declaration and public and private variables to maintain data integrity. Also an implementation of game loop for smooth cycle for each player in each depth-enough function.

9. Conclusion

The goal of this project was to understand the two major search algorithms used in Implementing the games using artificial intelligence. The two search algorithms used in this project are MinMaxAB algorithm and AlphaBetaSearch algorithm using Kalah Game as an example to test the implementation was a learning experience.

These two algorithms have there own similarities as well. We had to learn how to implement these two algorithms and use that to play a game of Kakah. This provided a chance to learn the rules of the game of Kalah. Over the course of completion of the project, I now have better understanding of MinMaxAB and AlphaBetaSearch algorithms. Having understood the concepts of these search algorithms prior to doing this project, this project gave me an opportunity to learn the implementation of the algorithms that we read to real life applications, Kalah Game in this case. Conversion of the understanding of concepts to a working project has been a path to learn the practical applications of these search algorithms.

Apart from this, we also got a chance to learn how important an evaluation function is and how different evaluation function makes the algorithm take a different route. We also got to learn on how to make an evaluation function. Each of the evaluation function displayed a particular behavior in terms of the number of nodes generated to determine a winner. From the gathered results, we were able to identify which evaluation function worked better and gave us a better performance. We compared both the algorithms with each of the evaluation functions

It is important to understand each of the above mentioned section properly to understand how the integration of all these components work together. The section result analysis gives us a detailed exposure to how each evaluation function that was created as part of this project worked in sync with the algorithms and also which evaluation function worked better.

I was able to keep a track of how many nodes were generated for each algorithm and each depth, which helped me to practically analyze as to which algorithm performed better and why. Was able to write code that was more readable, flexible and got a hold of using objects better. Besides, this project was a platform to learn in a better way the rules of Kalah Game and the history of how the game evolved over years and generations. I also got a better understanding on the software development process and the design principles as we followed the approach in this project. Such projects help us study different artificial intelligence algorithms. It also gave us a chance to understand the two algorithms in detail and also to compare the two algorithms. It is difficult to conclude which algorithm is better. Both have pros and cons. Among all the algorithms used in the industry to develop games, the minimax algorithm and alpha beta pruning algorithm are the most popular ones. Apart from all this, this project has given us an opportunity to learn or better our programming skills to develop programs which might not be straight forward at the first look.

I would like to sincerely thank you for this opportunity that helped me in many ways to get a hold of practical implementation of a given problem.

10.Source Code

```
#include<ctime>
#include <stdio.h>
#include <iostream>
#include <iomanip>
#include<ctime>
#include<cstdlib>

using namespace std;

int total_nodes,expanded_node_count,game_path_length;
const int SIZE = 6;

class Kalah
{
   public:
   int Player_A[SIZE],
      Player_B[SIZE],
      store_A,
      store_B,
      slots,
      maxWinNumber;

   int *pointer;

   Kalah();
   Kalah(Kalah *);
   void display();
   void operator=(Kalah);
   char move_decider(int, char);
   char move_generator_A(int);
   char move_generator_B(int);
   char result();

};

/*****************************************************************************
Kalah(): A default constructor that initializes all the data members of the
        Kalah Class.
        Each pit of Player A and Player B are initialized to having 6 stones
        stores of both players initialized to 0
        max number of stones that a person needs to be considered as having
        a winning chance is assigned to 36
parameters used: no
returns : constructor does not have a return type.
*****************************************************************************/

Kalah::Kalah()
{
```

```
    for(int i = 0 ; i < SIZE; i++)
    {
       Player_A[i] = 6;
       Player_B[i] = 6;
    }

    store_A = store_B = 0;
    slots = 5;

    pointer = NULL;
    maxWinNumber = 36;
}


/*****************************************************************************
Kalah(): A copy constructor that is used to initialize an object of class Kalah
         with another object. Copies the kalah value from one object to another
parameters used: no
returns : constructor does not have a return type.
*****************************************************************************/

Kalah::Kalah(Kalah *b)
{
    for(int i = 0; i < SIZE; i++)
    {
       this->Player_A[i] = b->Player_A[i];
       this->Player_B[i] = b->Player_B[i];
    }

    this->store_A = b->store_A;
    this->store_B = b->store_B;
}

/*****************************************************************************
dislay(): This function displays the Kalah board
parameters used: No parameters used
returns : Returns no values
*****************************************************************************/

void Kalah::display()
{
    cout << endl;
    cout << "\t\tPLAYER A";
    cout << endl << endl;
    cout << " \t | ";

    for(int i = 0 ; i < SIZE; i++)
       cout  << i << setw(2) << "|" << " ";

    cout << "----------> Pit Numbers of A" << endl;
    cout << "\t=========================";
```

```
    cout << endl;
    cout << "\t" << "|";
    cout << setw(2);


    for(int i = (SIZE - 1) ; i >= 0 ; i--)
        cout << Player_A[i] << setw(2) << "|" << " ";

    cout << endl;
    cout << "\t=========================";

    cout << endl;
    cout << "store_A" << "\t\t\t\t   store_B" <<endl;
    cout << "------"  << "\t\t\t\t   ------";
    cout << endl;
    cout <<"|" << setw(3) << store_A << " |"; // Fourth Line
    cout <<"\t\t\t\t   |" << setw(3) << store_B << " |" << endl;
    cout << "------"  << "\t\t\t\t   ------";
    cout << endl;
    cout << "\t=========================";
    cout << endl;
    cout << "\t" << "|";
    cout << setw(2);

    for(int i = 0 ; i < SIZE; i++)
        cout << Player_B[i] << setw(2) << "|" << " ";

    cout << endl;
    cout << "\t=========================";
    cout << endl;
    cout << " \t | ";

     for(int i = 0 ; i < SIZE; i++)
        cout  << i << setw(2) << "|" << " ";

    cout << "--------------> Pit Numbers of B";
    cout << endl << endl;
    cout << "\t\tPLAYER B";
    cout << endl << endl << endl << endl;
}

/*****************************************************************************
operater=(Kalah k): This is an operator overloading function that overloads the '='
parameters used: Object of type Kalah
returns : Returns no values
*****************************************************************************/

void Kalah::operator=(Kalah k)
{
    for(int i = 0; i < SIZE; i++)
    {
```

```
        Player_A[i] = k.Player_A[i];
        Player_B[i] = k.Player_B[i];

    }

    store_A = k.store_A;
    store_B = k.store_B;
    slots = 5;
    pointer = NULL;
    maxWinNumber = 36;

}

char Kalah::move_decider(int hole_num,char player)
{
    if(player == 'A')
        return (move_generator_A(hole_num));
    else
        return (move_generator_B(hole_num));
}


/*****************************************************************************
move_generator_A(): This function helps generate moves for Player A
                    If last stone lands in Player A's store, Player A gets another
                    chance. If player A's stone lands in an empty hole in Player A's
                    holes, player A captures all player B's stone from the opposite
                    hole. If the above two special scenarios are not encountered,
                    the moves are normal.
parameters used: Hole number from which the player should start playing
returns : Returns the next player that gets a chance after A's move.
          In most cases, B gets a turn. When A's stone lands in its store, A
          is returned, meaning, A gets an extra move
*****************************************************************************/

char Kalah::move_generator_A(int hole_num)
{
    int current_Position = hole_num;
    pointer = Player_A;

    int stoneCount = pointer[current_Position];

    pointer[current_Position] = 0;
    int opPosition, opStones;

    current_Position++;

    while(stoneCount > 0)
    {
        //last stone lands in Player A's store
        if(stoneCount >= 1 && current_Position == 6 )
```

```
{
   stoneCount--;
   store_A ++;

   if(stoneCount == 0)
      return 'A'; // giving A another chance as last stone falls in A's store
}

else if(current_Position >= 0 && current_Position <= 5)
{
   if(stoneCount == 1)
   {
      stoneCount--;

      opPosition = slots - current_Position;

      if(pointer[current_Position] == 0 && Player_B[opPosition] > 0)
      {
         pointer = Player_B;
         opStones = pointer[opPosition];
         pointer[opPosition] = 0;

         store_A += opStones + 1;

         if(stoneCount == 0)
            return 'B';
      }

      else
      {
         pointer[current_Position]++;

         if(stoneCount == 0)
            return 'B';
      }
   }

   else if(stoneCount > 1)
   {
      stoneCount--;

      pointer[current_Position]++;
   }
}

else if(current_Position > 6 && current_Position <= 12)
{
   pointer = Player_B;
   stoneCount--;

   pointer[current_Position-7]++;
```

```
        if(stoneCount == 0)
            return 'B';
      }

      //skip Player B's store
      else if(current_Position >= 12)
      {
         current_Position = -1;
         pointer = Player_A;
      }

      current_Position++;
   }
   return 'B';
}

/**************************************************************************************
move_generator_B(): This function helps generate moves for Player B
                    If last stone lands in Player B's store, Player B gets another
                    chance. If player B's stone lands in an empty hole in Player B's
                    holes, player B captures all player A's stone from the opposite
                    hole. If the above two special scenarios are not encountered,
                    the moves are normal.
parameters used: Hole number from which the player should start playing
returns : Returns the next player that gets a chance after B's move.
          In most cases, A gets a turn. When B's stone lands in its store, B
          is returned, meaning, B gets an extra move
**************************************************************************************/
char Kalah::move_generator_B(int hole_num)
{
   int current_Position = hole_num;
   pointer = Player_B;

   int stoneCount = pointer[current_Position];

   pointer[current_Position] = 0;
   int opPosition, opStones;

   current_Position++;

   while(stoneCount > 0)
   {
      //last stone lands in Player B's store
      if(stoneCount >= 1 && current_Position == 6)
      {
         stoneCount--;
         store_B++;

         if(stoneCount == 0)
            return 'B'; // giving B another chance as last stone falls in B's store
```

```
}

else if(current_Position >= 0 && current_Position <= 5)
{
   if(stoneCount == 1)
   {
      stoneCount--;

      opPosition = slots - current_Position;

      if(pointer[current_Position] == 0)
      {
         pointer = Player_A;
         opStones = pointer[opPosition];
         pointer[opPosition] = 0;

         store_B += opStones + 1;

         if(stoneCount == 0)
            return 'A';
      }

      else
      {
         pointer[current_Position]++;

         if(stoneCount == 0)
            return 'A';
      }
   }

   else if(stoneCount > 1)
   {
      stoneCount--;

      pointer[current_Position]++;
   }
}

else if(current_Position > 6 && current_Position <= 12)
{
   pointer = Player_A;
   stoneCount--;

   pointer[current_Position-7]++;

   if(stoneCount == 0)
      return 'A';
}

else if(current_Position >= 12)
```

```
        {
            current_Position = -1;
            pointer = Player_B;
        }

        current_Position++;
    }
    return 'A';
}


/*****************************************************************************
result(): This function checks the result of the game. If A has more than 36
          stones, A wins. If B has more than 36 stones, B wins.
parameters used: No parameters
returns : Returns A if Player A wins, B if player B wins, N if no win yet.
*****************************************************************************/

char Kalah::result()
{
    int empty_pits_A = 0,
        empty_pits_B = 0;

    for(int i = 0; i < SIZE; i++)
    {
        if(Player_A[i]==0)
            empty_pits_A++;

        if(Player_B[i]==0)
            empty_pits_B++;
    }

    if(empty_pits_A == 6)
    {
        for(int i = 0 ; i < SIZE; i++)
        {
            store_B += Player_B[i];
            Player_B[i] = 0;
        }

        cout << "Player A ran out of all the stones! All the pits in A are empty! " << endl;
    }

    if( empty_pits_B == 6)
    {
        for(int i = 0 ; i < SIZE ; i++)
        {
            store_A += Player_A[i];
            Player_A[i] = 0;
        }
```

```
        cout << "Player B ran out of all the stones! All the pits in B are empty! " << endl;
    }

    //if Player A has more than 36 stones, Player A wins
    if(store_A > maxWinNumber)
        return 'A';

    //if Player B has more than 36 stones, Player B wins
    else if(store_B > maxWinNumber)
        return 'B';

    else
        return 'N';
}



class KalahFlow
{

public:

    char player;
    Kalah current_status;
    int heuristic;
    KalahFlow * successors[SIZE];
    int successor_count;

    KalahFlow(char);
    void current_player_status(Kalah);
    bool is_deep_enough(int, int);
    void generate_successors();
    int myEvaluationFunction1();
    int myEvaluationFunction2();
    int myEvaluationFunction3();
    int myEvaluationFunction4();

};

/*****************************************************************************
KalahFlow(): A parameterized constructor that is used to initialize an object of class
            KalahFlow.
parameters used: player
returns : constructor does not have a return type.
*****************************************************************************/

KalahFlow::KalahFlow(char player)
{
    this->player = player;
    heuristic = -2000;
```

```
   successor_count = 0;

   for(int i = 0; i < SIZE; i++)
      successors[i] = NULL;
}

/*****************************************************************************
current_player_status(): copies the current status of the player into the current_status
object of type Kalah
parameters used: Kalah k
returns : returns nothing
*****************************************************************************/

void KalahFlow::current_player_status(Kalah k)
{
   for(int i = 0; i < SIZE; i++)
   {
      this->current_status.Player_A[i] = k.Player_A[i];
      this->current_status.Player_B[i] = k.Player_B[i];
   }

   this->current_status.store_A = k.store_A;
   this->current_status.store_B = k.store_B;
}

/*****************************************************************************
is_deep_enough(): This function checks if the depth is 2 or 4, which gives true for
              is_deep_enough for the algorithms. If it is not deep enough,
              we generate one more ply of the tree by calling the generate_successors()
              function
parameters used: the current depth passed by the functions, the selected maximum depth that the
            ply can be generated up to
returns : If the depth is greater than or equal to the selected depth, the function returns
       true. Otherwise, false is returned
*****************************************************************************/

bool KalahFlow::is_deep_enough(int depth, int selected)
{
   if(heuristic != -2000)
      return heuristic;

   if(depth >= selected || current_status.result() != 'N' )
      return true;

   else
   {
      expanded_node_count++;
      generate_successors();
      return false;
   }
}
```

```
/*****************************************************************************
generate_successors(): generates successors(child node) when the deep enough function
                       is satisfied to evaluate the future moves of the opponent based on
                       the present state of the board. This root node has child branches
                       which ends in the heuristic value of the end leaf node. Here the leaf
                       node is the value of number of stones in the pit at a given board state.
parameters used: no parameters used
returns : returns nothing
*****************************************************************************/

void KalahFlow::generate_successors()
{
   char player_temp;

   if(player == 'A')
      player_temp = 'B';

   else
      player_temp = 'A';

   for(int i = 0; i < SIZE; i++)
   {
      successor_count++;

      successors[i] = new KalahFlow(player_temp);

      if(current_status.Player_A[i]!=0 && player == 'A')
         successors[i]->current_status = current_status;

      else if(current_status.Player_B[i]!=0 && player == 'B')
         successors[i]->current_status = current_status;

      else
         successors[i] = NULL;

      if(successors[i]!=NULL)
      {
         total_nodes++;

         successors[i]->current_status.move_decider(i,successors[i]->player);
      }
   }
}

/*****************************************************************************
my_evaluation_function1():used to estimate the value or goodness of a position at leaf node
parameters used: no parameters
returns : returns the terminal value
*****************************************************************************/
```

```
int KalahFlow::myEvaluationFunction1()
{
   int terminal_value;

   if(player == 'A')
   {
      int stones_of_A = 0;

      for(int i = 0 ; i < 6 ; i++)
      {
         if(current_status.Player_A[i] == 0)
            stones_of_A++;
      }

      if(stones_of_A == 6)
         terminal_value = 2000;
   }
   else if(player == 'B')
   {
      int stones_of_B = 0;

      for(int i = 0 ; i < 6 ; i++)
      {
         if(current_status.Player_B[i] == 0)
            stones_of_B++;
      }
      if(stones_of_B == 6)
         terminal_value = -2000;
   }

   heuristic = terminal_value;

   return terminal_value;
}

/*****************************************************************************
my_evaluation_function2():used to estimate the value or goodness of a position at leaf node
parameters used: no parameters
returns : returns the terminal value
*****************************************************************************/

int KalahFlow::myEvaluationFunction2()
{

int terminal_value;
   if(player == 'A')
      terminal_value = current_status.store_A - current_status.store_B;

   else if(player == 'B')
      terminal_value = current_status.store_A - current_status.store_B;
```

```
    heuristic = terminal_value;

    return terminal_value;

}

/*****************************************************************************
my_evaluation_function3():used to estimate the value or goodness of a position at leaf node
parameters used: no parameters
returns : returns the terminal value
*****************************************************************************/

int KalahFlow::myEvaluationFunction3()
{
    int p0 = current_status.store_A;
    int p1 = current_status.store_B;
    int terminal_value;

    if(player == 'A')
    {
        if(p0 != 0)
            terminal_value = (current_status.store_A/current_status.store_B);

        else
            return -1;
    }

    else if(player == 'B')
    {
        if(p1 !=0)
            terminal_value = (current_status.store_B/ current_status.store_A);

        else
            return -1;

    heuristic = terminal_value;
    return terminal_value;
}
}

/*****************************************************************************
my_evaluation_function4():used to estimate the value or goodness of a position at leaf node
parameters used: no parameters
returns : returns the terminal value
*****************************************************************************/

int KalahFlow::myEvaluationFunction4()
{
    int terminal_value;
    if (player == 'A'){
        int stones_of_A = 0;
```

```
      for(int i=0; i<6; i++) {
         if (current_status.Player_A[i]==0) stones_of_A++;
         else if(current_status.Player_A[i]==i) terminal_value = 1500;
         else if(current_status.Player_A[i]<i) terminal_value = 800;
         else
            terminal_value = 500;
      }
      if(stones_of_A == 6)
         terminal_value = 1000; }
   if (player == 'B'){
      int stones_of_B = 0;
      for(int i=0; i<6; i++) {
         if (current_status.Player_A[i]==0) stones_of_B++;
         else if(current_status.Player_A[i]==i) terminal_value = 1500;
         else if(current_status.Player_A[i]<i) terminal_value = 800;
         else
            terminal_value = 500;
      }
      if(stones_of_B == 6)
         terminal_value = 1000; }
   heuristic = terminal_value;
   return terminal_value;

}

/*****************************************************************************
AlphaBetaSearch(): Alpha Beta search algorithm implementation for Kalah Game
parameters used: an object of type KalahFlow, the current depth which is 0 during the initial call,
            the current player, which was 'A' initially, alpha, beta, selectedDepth,
            which can be 2 or 4, the maximum depth, evaluation_choice 1, 2 or 3
            select from three possible evaluation function options.
returns : returns the best value
*****************************************************************************/

int AlphaBetaSearch(KalahFlow *alphabeta, int depth, char player, int alpha, int beta, int selected,
int evaluation_choice)
{

   int bestValue = -100;
   int value;

   if(alphabeta->is_deep_enough(depth, selected))
   {
      if(evaluation_choice == 1)
         return alphabeta->myEvaluationFunction1();

      else if (evaluation_choice == 2)
         return alphabeta->myEvaluationFunction2();

      else if (evaluation_choice == 3)
         return alphabeta->myEvaluationFunction3();
```

```
    else
        return alphabeta->myEvaluationFunction4();
  }

  if (player == 'A' )

  {

    for (int i=0; i < 6; i++ )

    {

      if(alphabeta->successors[i]== NULL)

      continue;

      value = AlphaBetaSearch(alphabeta->successors[i], depth+1, alphabeta->player, alpha, beta,
selected, evaluation_choice);

      if (bestValue > value)

      bestValue = max(bestValue,value);

      alpha=max(alpha,bestValue);

      if (beta <= alpha)

      break;  }

    alphabeta->heuristic = bestValue;

    return bestValue;

  }


  else

  {

    int bestValue = +100;

    int value;


    for(int i=0; i < 6; i++)

    {
```

```
        if(alphabeta->successors[i]== NULL)

        continue;

        value = AlphaBetaSearch(alphabeta->successors[i], depth+1, alphabeta->player, alpha, beta,
selected, evaluation_choice);

        if (bestValue < value)

        bestValue = min( bestValue, value);

        beta = min( beta, bestValue);

        if (beta <= alpha)

        break;

      }

      alphabeta->heuristic = bestValue;

      return bestValue;

   }

}

/**************************************************************************
MinMaxAB(): MinMAxAB search algorithm implementation for Kalah Game
parameters used: an object of type KalahFlow, the current depth which is 0 during the initial call,
            the current player, which was 'A' initially, Use-Threshold, Pass-Threshold value,
            selectedDepth, which can be 2 or 4, the maximum depth, evaluation_choice-1, 2 or 3 – to
            select from three possible evaluation function options.
returns : returns the best value
**************************************************************************/

int MinMaxAB(KalahFlow *min_max,int depth, char player, int use_Threshold, int pass_Threshold,
int selectedDepth, int evaluation_choice)
{
   int structure;
   int new_value;
   char new_Player;
   int result_successor = 0;

   if(min_max->is_deep_enough(depth, selectedDepth))
   {

      if(evaluation_choice == 1)
         structure = min_max->myEvaluationFunction1();

      else if (evaluation_choice == 2)
```

```
            structure = min_max->myEvaluationFunction2();

        else if (evaluation_choice == 3)
            structure = min_max->myEvaluationFunction3();

        else
            structure = min_max->myEvaluationFunction4();

        if(player == 'B')
            structure = -structure;

        min_max->heuristic = structure;

        return structure;
    }

    for(int i = 0 ; i < SIZE; i++)
    {
        if(min_max->successors[i] == NULL)
            continue;

        if(player == 'A')
            new_Player = 'B';

        else
            new_Player = 'A';

        result_successor = MinMaxAB(min_max->successors[i],depth+1,new_Player,-pass_Threshold,-
use_Threshold, selectedDepth, evaluation_choice);

        new_value = -result_successor;

        if(new_value > pass_Threshold)
        {
            min_max->heuristic = i;
            pass_Threshold = new_value;
        }

        if(pass_Threshold >= use_Threshold)
        {
            result_successor = pass_Threshold;
            return result_successor;
        }
    }

    result_successor = pass_Threshold;

    return result_successor;
}

/*************************************************************************************
```

```
winner(): Prints the result of the winner, the computation time, memory consumed,
        number of nodes generated, number of nodes expanded and the total game path length
parameters used: result if A won the game or B, total time , object of type Kalah
returns : returns nothing
*************************************************************************************/

void winner(char result,int total_time, Kalah k)
{
   if(result == 'A')
      cout << "Player A won the GAME with " << k.store_A << " stones! " << endl << endl;

   else if(result == 'B')
      cout << "Player B won the GAME with " << k.store_B << " stones! " << endl << endl;

   cout                                                                                    <<
"*************************************************************************************
*********************************" << endl;
   cout << "                                ***GAME OVER***                                " << endl
;
   cout                                                                                    <<
"*************************************************************************************
*********************************" << endl << endl << endl;

   cout << "The program took " << "'" << (total_time)/double(CLOCKS_PER_SEC)<< "'" << "
seconds of Execution time for the completed game! " << endl << endl;
   cout << "1 node takes 81 bytes of Memory "<<endl << endl;
   cout << "The algorithm takes : "<< "'" << (81* total_nodes) << "'" << " bytes = " << "'" << (81*
total_nodes) / (1024) << "'" << " kb" << " of memory!" << endl << endl;
   cout << "The program generated " << "'" << total_nodes << "'" << " nodes for the completed
game!" << endl << endl;
   cout << "The program expanded " << "'" << expanded_node_count << "'" << " nodes for the
completed game!" << endl << endl;
   cout << "The total length of the Game path is " << "'" << game_path_length << "'" << " for the
completed game! " << endl << endl;

}

/*************************************************************************************
main():The program starts execution from this function main. It is the driver to
      integrate all the classes and functions defined above
parameters used: no parameters
*************************************************************************************/
int main()
{
   int choice, selectedDepth;
   int evaluation_choice;
   Kalah *kalah = new Kalah();


      cout << endl << endl;
```

```
	cout <<
"*******************************************************************************
*****************************" << endl;
	cout << "                                        YOU ARE NOW IN THE KALAH GAME ZONE!
" << endl ;
	cout <<
"*******************************************************************************
*****************************" << endl;
	cout << endl;
	cout << "                            The game rules are as below:                        " << endl << endl;
	cout << " ** Kalah is played by two players on a board with two rows of 6 holes facing each
other." << endl << endl;
	cout << " ** It has two KALAHS (Stores for each player.)" << endl << endl;
	cout << " ** The stones are called beans. At the beginning of the game, there are 6 beans in every
hole. The KALAHS are empty." << endl << endl;
	cout << " ** The object of Kalah is to get as many beans into your own kalah by distributing
them." << endl << endl;
	cout << " ** A player moves by taking all the beans in one of his 6 holes and distributing them
counterclockwise," << endl;
	cout << "    by putting one bean in every hole including his, but excluding the opponent's kalah."
<< endl << endl;
	cout << " ** There are two special moves:" << endl << endl;
	cout << " ** Extra move: If the last bean is distributed into his own kalah, the player may move
again. " << endl
	   << "    He has to move again even if he does not want to." << endl << endl;
	cout << " ** Capture: If the last bean falls into an empty hole of the player and the opponent's
hole above (or below) is not empty," << endl
	   << "    the player puts his last bean and all the beans in his opponent's hole into his kalah. He
has won all those beans." << endl << endl;
	cout << " ** The game ends if all 6 holes of one player become empty (no matter if it is this
player's move or not)." << endl
	   <<"     The beans in the other player's holes are given into this player's kalah." << endl <<
endl;
	cout << " ** The player who won more beans (at least 37) becomes the winner." << endl << endl;


	cout                                       <<                                        "
*******************************************************************************
****************************" << endl;
	cout << "                                ***KALAH BOARD DISPLAY***                        "
<< endl ;
	cout                                       <<                                        "
*******************************************************************************
****************************" << endl;

	kalah->display();

	cout << " You now have the below choices :" << endl << endl
	   << "Choose your option :" << endl << endl;

	cout << " ** Choose 1 for MinMaxAB: " << endl;
```

```
    cout << " ** Choose 2 for AlphaBetaSearch: " << endl;
    cout << " ** Choose 3 to QUIT:" << endl;
    cin >> choice;

    cout << "Please enter the depth that you would want to check the results for: " << endl;
    cout << "2: Depth of 2" << endl;
    cout << "4: Depth of 4" << endl;
    cin >> selectedDepth;

    while(selectedDepth !=2 && selectedDepth !=4)
    {
       cout << "Invalid Depth! Please enter depth value 2 or 4!" << endl;
       cin >> selectedDepth;
    }

    cout << "Please enter the Evaluation function that you would want to check the results for: "<<
endl;
    cout << "1: Evaluation Function 1 ( Vidhyashree Nagabhushana ) " << endl;
    cout << "2: Evaluation Function 2 ( Akshatha Jain ) " << endl;
    cout << "3: Evaluation Function 3 ( Akshatha Jain ) " << endl;
    cout << "4: Evaluation Function 4 ( Sahana Sreenath) " << endl;
    cin >> evaluation_choice;


    while(evaluation_choice < 1 || evaluation_choice > 4 )
    {
       cout << "Invalid selection! Please enter values 1, 2, 3 or 4!" << endl;
       cin >> evaluation_choice;
    }

    switch(choice)
    {
       case 1:
          {
             cout << "Initial board " << endl;
             kalah->display();

             char result = kalah->result();

             char player ='A';
             int start_timer = clock();

             while(result == 'N')
             {
                KalahFlow *min_max_ab = new KalahFlow(player);

                min_max_ab->current_player_status( kalah );

                cout << "It is player " << player << "'s turn!" << endl;

                MinMaxAB(min_max_ab,0,player,2000,-2000,selectedDepth,evaluation_choice);
```

```
                int pit = min_max_ab->heuristic;

                if(player == 'A')
                   player = kalah->move_generator_A(pit);

                else
                   player = kalah->move_generator_B(pit);

                kalah->display();

                result = kalah->result();

                game_path_length++;
             }

             int stop_timer = clock();
             kalah->display();
             int total_time = stop_timer - start_timer;
             winner(result, total_time, kalah );
          }

          break;

       case 2:
          {
             cout << "Initial board " << endl;
             kalah->display();

             char result = kalah->result();

             char player = 'A';
             int start_timer = clock();

             while(result == 'N')
             {
                KalahFlow *alpha_beta_flow = new KalahFlow(player);

                alpha_beta_flow->current_player_status(kalah);

                cout << "It is player " << player << "'s turn!" << endl;

                AlphaBetaSearch(alpha_beta_flow,0,player,2000,-2000,
selectedDepth,evaluation_choice);

                int pit;

                for(int i = 0; i < SIZE; i++)
                {
                   if(alpha_beta_flow->successors[i] == NULL)
                      continue;
```

```
                    if(alpha_beta_flow->successors[i]->heuristic == alpha_beta_flow->heuristic)
                        pit = i;
                }

                if(player == 'A')
                    player = kalah->move_generator_A(pit);

                else
                    player = kalah->move_generator_B(pit);

                kalah->display();

                result = kalah->result();

                game_path_length++;
            }

            int stop_timer = clock();
            kalah->display();
            int total_time = stop_timer - start_timer;
            winner(result,total_time, kalah);
        }

        break;

    case 3:
        cout <<"Thanks. See you again later....." << endl;
        return 0;
    }
  return 0;
}
```

11.A Copy of the program Run

I will consider myEvaluation function3 for the output.

1. MINMAXAB with depth2

```
**********************************************************************************
**********************************
                    YOU ARE NOW IN THE KALAH GAME ZONE!
**********************************************************************************
**********************************

                    The game rules are as below:

 ** Kalah is played by two players on a board with two rows of 6 holes facing each other.

 ** It has two KALAHS (Stores for each player.)

 ** The stones are called beans. At the beginning of the game, there are 6 beans in every hole.
The KALAHS are empty.

 ** The object of Kalah is to get as many beans into your own kalah by distributing them.

 ** A player moves by taking all the beans in one of his 6 holes and distributing them
counterclockwise,
   by putting one bean in every hole including his, but excluding the opponent's kalah.

 ** There are two special moves:

 ** Extra move: If the last bean is distributed into his own kalah, the player may move again.
   He has to move again even if he does not want to.

 ** Capture: If the last bean falls into an empty hole of the player and the opponent's hole
above (or below) is not empty,
   the player puts his last bean and all the beans in his opponent's hole into his kalah. He has
won all those beans.

 ** The game ends if all 6 holes of one player become empty (no matter if it is this player's
move or not).
    The beans in the other player's holes are given into this player's kalah.

 ** The player who won more beans (at least 37) becomes the winner.
```

```
**********************************************************************************
**********************************
                        ***KALAH BOARD DISPLAY***

**********************************************************************************
**********************************

          PLAYER A

      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
     =========================
     | 6 | 6 | 6 | 6 | 6 | 6 |
     =========================
store_A                  store_B
------                  ------
|  0 |                   |  0 |
------                  ------
     =========================
     | 6 | 6 | 6 | 6 | 6 | 6 |
     =========================
      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

          PLAYER B


 You now have the below choices :

Choose your option :

 ** Choose 1 for MinMaxAB:
 ** Choose 2 for AlphaBetaSearch:
 ** Choose 3 to QUIT:
1
Please enter the depth that you would want to check the results for:
2: Depth of 2
4: Depth of 4
2
Please enter the Evaluation function that you would want to check the results for:
1: Evaluation Function 1 ( Vidhyashree Nagabhushana )
2: Evaluation Function 2 ( Akshatha Jain )
3: Evaluation Function 3 ( Akshatha Jain )
4: Evaluation Function 4 ( Sahana Sreenath)
3
Initial board
```

```
            PLAYER A

      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      | 6 | 6 | 6 | 6 | 6 | 6 |
      =========================
store_A                  store_B
------                  ------
|  0 |                   |  0 |
------                  ------
      =========================
      | 6 | 6 | 6 | 6 | 6 | 6 |
      =========================
      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player A's turn!

            PLAYER A

      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      | 7 | 7 | 7 | 7 | 7 | 0 |
      =========================
store_A                  store_B
------                  ------
|  1 |                   |  0 |
------                  ------
      =========================
      | 6 | 6 | 6 | 6 | 6 | 6 |
      =========================
      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player A's turn!

            PLAYER A

      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      | 8 | 8 | 8 | 8 | 0 | 0 |
      =========================
```

```
store_A                 store_B
------                  ------
|  2 |                  |  0 |
------                  ------
      ========================
      | 7 | 7 | 6 | 6 | 6 | 6 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      | 8 | 8 | 8 | 8 | 0 | 1 |
      ========================
store_A                 store_B
------                  ------
|  2 |                  |  1 |
------                  ------
      ========================
      | 0 | 8 | 7 | 7 | 7 | 7 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player A's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      | 8 | 8 | 8 | 8 | 0 | 0 |
      ========================
store_A                 store_B
------                  ------
| 10 |                  |  1 |
------                  ------
      ========================
```

```
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B



It is player B's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      | 8 | 8 | 8 | 9 | 1 | 1 |
      ========================
store_A                  store_B
------                  ------
| 10 |                   |  2 |
------                  ------
      ========================
      | 0 | 0 | 8 | 8 | 1 | 8 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B



It is player A's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      | 8 | 8 | 8 | 9 | 2 | 0 |
      ========================
store_A                  store_B
------                  ------
| 10 |                   |  2 |
------                  ------
      ========================
      | 0 | 0 | 8 | 8 | 1 | 8 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B
```

```
It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      | 8 | 8 | 9 | 10 | 3 | 1 |
      =========================
store_A                   store_B
------                    ------
| 10 |                    |  3 |
------                    ------
      =========================
      | 0 | 0 | 0 | 9 | 2 | 9 |
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player A's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      | 8 | 8 | 9 | 10 | 4 | 0 |
      =========================
store_A                   store_B
------                    ------
| 10 |                    |  3 |
------                    ------
      =========================
      | 0 | 0 | 0 | 9 | 2 | 9 |
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
```

```
      ========================
      | 9 | 9 | 10 | 11 | 5 | 1 |
      ========================
store_A                  store_B
------                   ------
| 10 |                    |  4 |
------                   ------
      ========================
      | 0 | 0 | 0 | 0 | 3 | 10 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B



It is player A's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      | 9 | 9 | 10 | 11 | 6 | 0 |
      ========================
store_A                  store_B
------                   ------
| 10 |                    |  4 |
------                   ------
      ========================
      | 0 | 0 | 0 | 0 | 3 | 10 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B



It is player B's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      | 9 | 9 | 10 | 11 | 6 | 1 |
      ========================
store_A                  store_B
------                   ------
| 10 |                    |  5 |
```

```
------                    ------
      ==========================
      | 0 | 0 | 0 | 0 | 0 | 11 |
      ==========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B



It is player A's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ==========================
      | 9 | 9 | 10 | 11 | 7 | 0 |
      ==========================
store_A                   store_B
------                    ------
| 10 |                    |  5 |
------                    ------
      ==========================
      | 0 | 0 | 0 | 0 | 0 | 11 |
      ==========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B



It is player B's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ==========================
      |10 | 10 | 11 | 0 | 8 | 1 |
      ==========================
store_A                   store_B
------                    ------
| 10 |                    | 19 |
------                    ------
      ==========================
      | 1 | 1 | 1 | 0 | 0 | 0 |
      ==========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B
```

```
It is player A's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      |10 | 10 | 11 | 0 | 9 | 0 |
      ========================
store_A                    store_B
------                    ------
| 10 |                     | 19 |
------                    ------
      ========================
      | 1 | 1 | 1 | 0 | 0 | 0 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      |10 | 10 | 11 | 0 | 9 | 0 |
      ========================
store_A                    store_B
------                    ------
| 10 |                     | 19 |
------                    ------
      ========================
      | 0 | 2 | 1 | 0 | 0 | 0 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


It is player A's turn!
```

```
      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
     =========================
     |11 | 11 | 12 | 1 | 0 | 0 |
     =========================
store_A                 store_B
------                  ------
| 11 |                  | 19 |
------                  ------
     =========================
     | 1 | 3 | 2 | 1 | 0 | 0 |
     =========================
      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

          PLAYER B



It is player B's turn!

          PLAYER A

      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
     =========================
     |11 | 11 | 12 | 1 | 0 | 0 |
     =========================
store_A                 store_B
------                  ------
| 11 |                  | 19 |
------                  ------
     =========================
     | 0 | 4 | 2 | 1 | 0 | 0 |
     =========================
      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

          PLAYER B



It is player A's turn!

          PLAYER A

      | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
     =========================
     |11 | 11 | 13 | 0 | 0 | 0 |
     =========================
```

```
store_A                 store_B
------                  ------
| 11 |                  | 19 |
------                  ------
      =========================
      | 0 | 4 | 2 | 1 | 0 | 0 |
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B



It is player B's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      |11 | 11 | 13 | 0 | 0 | 0 |
      =========================
store_A                 store_B
------                  ------
| 11 |                  | 20 |
------                  ------
      =========================
      | 0 | 0 | 3 | 2 | 1 | 0 |
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B



It is player A's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      |12 | 12 | 0 | 1 | 1 | 1 |
      =========================
store_A                 store_B
------                  ------
| 17 |                  | 20 |
------                  ------
      =========================
      | 1 | 1 | 0 | 3 | 2 | 1 |
```

```
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B


It is player B's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      |12 | 12 | 0 | 1 | 1 | 1 |
      =========================
store_A                  store_B
------                  ------
| 17 |                   | 20 |
------                  ------
      =========================
      | 0 | 2 | 0 | 3 | 2 | 1 |
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B


It is player A's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      |12 | 12 | 0 | 1 | 2 | 0 |
      =========================
store_A                  store_B
------                  ------
| 17 |                   | 20 |
------                  ------
      =========================
      | 0 | 2 | 0 | 3 | 2 | 1 |
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B
```

```
It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      |12 | 12 | 0 | 1 | 2 | 0 |
      =========================
store_A                   store_B
------                    ------
| 17 |                    | 20 |
------                    ------
      =========================
      | 0 | 0 | 1 | 4 | 2 | 1 |
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player A's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      |12 | 12 | 0 | 2 | 0 | 0 |
      =========================
store_A                   store_B
------                    ------
| 19 |                    | 20 |
------                    ------
      =========================
      | 0 | 0 | 0 | 4 | 2 | 1 |
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
```

```
      ========================
      |12 | 12 | 0 | 2 | 0 | 0 |
      ========================
store_A                  store_B
------                   ------
| 19 |                    | 21 |
------                   ------
      ========================
      | 0 | 0 | 0 | 4 | 0 | 2 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B



It is player B's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      |12 | 12 | 0 | 2 | 0 | 1 |
      ========================
store_A                  store_B
------                   ------
| 19 |                    | 22 |
------                   ------
      ========================
      | 0 | 0 | 0 | 0 | 1 | 3 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

           PLAYER B



It is player A's turn!

           PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      |12 | 12 | 0 | 2 | 0 | 0 |
      ========================
store_A                  store_B
------                   ------
| 21 |                    | 22 |
```

```
------                    ------
      ==========================
      | 0 | 0 | 0 | 0 | 0 | 3 |
      ==========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ==========================
      |12 | 12 | 0 | 2 | 1 | 1 |
      ==========================
store_A                   store_B
------                    ------
| 21 |                    | 23 |
------                    ------
      ==========================
      | 0 | 0 | 0 | 0 | 0 | 0 |
      ==========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



Player B ran out of all the stones! All the pits in B are empty!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ==========================
      | 0 | 0 | 0 | 0 | 0 | 0 |
      ==========================
store_A                   store_B
------                    ------
| 49 |                    | 23 |
------                    ------
      ==========================
      | 0 | 0 | 0 | 0 | 0 | 0 |
      ==========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B
```

```
            PLAYER B


Player A won the GAME with 49 stones!

**************************************************************************************
**********************************
                              ***GAME OVER***
**************************************************************************************
**********************************

The program took '0.444' seconds of Execution time for the completed game!
1 node takes 81 bytes of Memory
The algorithm takes : '40338' bytes = '39' kb of memory!
The program generated '498' nodes for the completed game!
The program expanded '142' nodes for the completed game!
The total length of the Game path is '28' for the completed game!
Process returned 0 (0x0)   execution time : 32.890 s
Press any key to continue.
```

OUTPUT 2:
MINMAX AB(using evaluation 3 and depth 4)

```
**********************************************************************
**********************************************
                        ***KALAH BOARD DISPLAY***

**********************************************************************
**********************************************

            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       | 6 | 6 | 6 | 6 | 6 | 6 |
       ========================
store_A                    store_B
------                     ------
|  0 |                    |  0 |
------                     ------
       ========================
       | 6 | 6 | 6 | 6 | 6 | 6 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


 You now have the below choices :

Choose your option :

 ** Choose 1 for MinMaxAB:
 ** Choose 2 for AlphaBetaSearch:
 ** Choose 3 to QUIT:
1
Please enter the depth that you would want to check the results for:
```

```
2: Depth of 2
4: Depth of 4
4
Please enter the Evaluation function that you would want to check the results
for:
1: Evaluation Function 1 ( Vidhyashree Nagabhushana )
2: Evaluation Function 2 ( Akshatha Jain )
3: Evaluation Function 3 ( Akshatha Jain )
4: Evaluation Function 4 ( Sahana Sreenath)
3
Initial board

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       =========================
       | 6 | 6 | 6 | 6 | 6 | 6 |
       =========================
store_A                    store_B
------                     ------
|  0 |                    |  0 |
------                     ------
       =========================
       | 6 | 6 | 6 | 6 | 6 | 6 |
       =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player A's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       =========================
       | 7 | 7 | 7 | 7 | 7 | 0 |
       =========================
store_A                    store_B
```

```
------                    ------
|  1 |                   |  0 |
------                    ------
       ========================
       | 6 | 6 | 6 | 6 | 6 | 6 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

             PLAYER B


It is player A's turn!

             PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       | 8 | 8 | 8 | 8 | 0 | 0 |
       ========================
store_A                   store_B
------                    ------
|  2 |                   |  0 |
------                    ------
       ========================
       | 7 | 7 | 6 | 6 | 6 | 6 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

             PLAYER B


It is player B's turn!

             PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       | 8 | 8 | 8 | 8 | 0 | 1 |
```

```
store_A                    store_B
------                    ------
|  2 |                    |  1 |
------                    ------
        ========================
        | 0 | 8 | 7 | 7 | 7 | 7 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player A's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        | 8 | 8 | 8 | 8 | 0 | 0 |
        ========================
store_A                    store_B
------                    ------
| 10 |                    |  1 |
------                    ------
        ========================
        | 0 | 8 | 7 | 7 | 0 | 7 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player B's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
```

```
        ========================
        | 8 | 8 | 8 | 9 | 1 | 1 |
        ========================
store_A                    store_B
------                     ------
| 10 |                     |  2 |
------                     ------
        ========================
        | 0 | 0 | 8 | 8 | 1 | 8 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player A's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        | 8 | 8 | 8 | 9 | 2 | 0 |
        ========================
store_A                    store_B
------                     ------
| 10 |                     |  2 |
------                     ------
        ========================
        | 0 | 0 | 8 | 8 | 1 | 8 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player B's turn!

              PLAYER A
```

```
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        | 8 | 8 | 9 | 10 | 3 | 1 |
        ========================
store_A                    store_B
------                     ------
| 10 |                     |  3 |
------                     ------
        ========================
        | 0 | 0 | 0 | 9 | 2 | 9 |
        ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

             PLAYER B


It is player A's turn!

             PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        | 8 | 8 | 9 | 10 | 4 | 0 |
        ========================
store_A                    store_B
------                     ------
| 10 |                     |  3 |
------                     ------
        ========================
        | 0 | 0 | 0 | 9 | 2 | 9 |
        ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

             PLAYER B


It is player B's turn!
```

```
            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       | 9 | 9 | 10 | 11 | 5 | 1 |
       ========================
store_A                    store_B
------                     ------
| 10 |                     |  4 |
------                     ------
       ========================
       | 0 | 0 | 0 | 0 | 3 | 10 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


It is player A's turn!

            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       | 9 | 9 | 10 | 11 | 6 | 0 |
       ========================
store_A                    store_B
------                     ------
| 10 |                     |  4 |
------                     ------
       ========================
       | 0 | 0 | 0 | 0 | 3 | 10 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B
```

```
It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      | 9 | 9 | 10 | 11 | 6 | 1 |
      =========================
store_A                   store_B
------                    ------
| 10 |                    |  5 |
------                    ------
      =========================
      | 0 | 0 | 0 | 0 | 0 | 11 |
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


It is player A's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      =========================
      | 9 | 9 | 10 | 11 | 7 | 0 |
      =========================
store_A                   store_B
------                    ------
| 10 |                    |  5 |
------                    ------
      =========================
      | 0 | 0 | 0 | 0 | 0 | 11 |
      =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B
```

```
It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      |10 | 10 | 11 | 0 | 8 | 1 |
      ========================
store_A                   store_B
------                    ------
| 10 |                    | 19 |
------                    ------
      ========================
      | 1 | 1 | 1 | 0 | 0 | 0 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


It is player A's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      |10 | 10 | 11 | 0 | 9 | 0 |
      ========================
store_A                   store_B
------                    ------
| 10 |                    | 19 |
------                    ------
      ========================
      | 1 | 1 | 1 | 0 | 0 | 0 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B
```

```
            PLAYER B


It is player B's turn!

            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       |10 | 10 | 11 | 0 | 9 | 0 |
       ========================
store_A                    store_B
------                     ------
| 10 |                     | 19 |
------                     ------
       ========================
       | 0 | 2 | 1 | 0 | 0 | 0 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


It is player A's turn!

            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       |11 | 11 | 12 | 1 | 0 | 0 |
       ========================
store_A                    store_B
------                     ------
| 11 |                     | 19 |
------                     ------
       ========================
       | 1 | 3 | 2 | 1 | 0 | 0 |
```

```
        ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

                PLAYER B


It is player B's turn!

                PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        |11 | 11 | 12 | 1 | 0 | 0 |
        ========================
store_A                    store_B
------                     ------
| 11 |                     | 19 |
------                     ------
        ========================
        | 0 | 4 | 2 | 1 | 0 | 0 |
        ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

                PLAYER B


It is player A's turn!

                PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        |11 | 11 | 13 | 0 | 0 | 0 |
        ========================
store_A                    store_B
------                     ------
| 11 |                     | 19 |
------                     ------
```

```
        ========================
        | 0 | 4 | 2 | 1 | 0 | 0 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player B's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        |11 | 11 | 13 | 0 | 0 | 0 |
        ========================
store_A                    store_B
------                     ------
| 11 |                     | 20 |
------                     ------
        ========================
        | 0 | 0 | 3 | 2 | 1 | 0 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player A's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        |12 | 12 | 0 | 1 | 1 | 1 |
        ========================
store_A                    store_B
------                     ------
```

```
| 17 |                    | 20 |
------                   ------
       ========================
       | 1 | 1 | 0 | 3 | 2 | 1 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

             PLAYER B


It is player B's turn!

             PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       |12 | 12 | 0 | 1 | 1 | 1 |
       ========================
store_A                    store_B
------                   ------
| 17 |                   | 20 |
------                   ------
       ========================
       | 0 | 2 | 0 | 3 | 2 | 1 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

             PLAYER B


It is player A's turn!

             PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       |12 | 12 | 0 | 1 | 2 | 0 |
       ========================
```

```
store_A                    store_B
------                     ------
| 17 |                     | 20 |
------                     ------
        ========================
        | 0 | 2 | 0 | 3 | 2 | 1 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player B's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        |12 | 12 | 0 | 1 | 2 | 0 |
        ========================
store_A                    store_B
------                     ------
| 17 |                     | 21 |
------                     ------
        ========================
        | 0 | 2 | 0 | 3 | 0 | 2 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player B's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
```

```
        |12 | 12 | 0 | 1 | 2 | 0 |
        ========================
store_A                    store_B
------                     ------
| 17 |                     | 21 |
------                     ------
        ========================
        | 0 | 0 | 1 | 4 | 0 | 2 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B



It is player A's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        |12 | 12 | 0 | 2 | 0 | 0 |
        ========================
store_A                    store_B
------                     ------
| 19 |                     | 21 |
------                     ------
        ========================
        | 0 | 0 | 0 | 4 | 0 | 2 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B



It is player B's turn!

              PLAYER A
```

```
 | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
=========================
|12 | 12 | 0 | 2 | 0 | 1 |
=========================
store_A                 store_B
------                  ------
| 19 |                  | 22 |
------                  ------
=========================
| 0 | 0 | 0 | 0 | 1 | 3 |
=========================
 | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

        PLAYER B


It is player A's turn!

        PLAYER A

 | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
=========================
|12 | 12 | 0 | 2 | 0 | 0 |
=========================
store_A                 store_B
------                  ------
| 21 |                  | 22 |
------                  ------
=========================
| 0 | 0 | 0 | 0 | 0 | 3 |
=========================
 | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

        PLAYER B


It is player B's turn!
```

```
            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       =========================
       |12 | 12 | 0 | 2 | 1 | 1 |
       =========================
store_A                    store_B
------                     ------
| 21 |                     | 23 |
------                     ------
       =========================
       | 0 | 0 | 0 | 0 | 0 | 0 |
       =========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


Player B ran out of all the stones! All the pits in B are empty!

            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       =========================
       | 0 | 0 | 0 | 0 | 0 | 0 |
       =========================
store_A                    store_B
------                     ------
| 49 |                     | 23 |
------                     ------
       =========================
       | 0 | 0 | 0 | 0 | 0 | 0 |
       =========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B
```

```
**************************************************
                    ***GAME OVER***
**************************************************
**************************************************

The program took '0.437' seconds of Execution time for the completed game!

1 node takes 81 bytes of Memory

The algorithm takes : '326511' bytes = '318' kb of memory!

The program generated '4031' nodes for the completed game!

The program expanded '983' nodes for the completed game!

The total length of the Game path is '28' for the completed game!


Process returned 0 (0x0)   execution time : 8.789 s
Press any key to continue.
OUTPUT 3(Evaluation function3 AlphaBeta depth2):


**************************************************
**************************************************
                    ***KALAH BOARD DISPLAY***

**************************************************
**************************************************

        PLAYER A

    | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
    =========================
    | 6 | 6 | 6 | 6 | 6 | 6 |
    =========================
```

```
store_A                    store_B
------                    ------
|  0 |                   |  0 |
------                    ------
       ========================
       | 6 | 6 | 6 | 6 | 6 | 6 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

             PLAYER B



 You now have the below choices :

Choose your option :

 ** Choose 1 for MinMaxAB:
 ** Choose 2 for AlphaBetaSearch:
 ** Choose 3 to QUIT:
2
Please enter the depth that you would want to check the results for:
2: Depth of 2
4: Depth of 4
2
Please enter the Evaluation function that you would want to check the results
for:
1: Evaluation Function 1 ( Vidhyashree Nagabhushana )
2: Evaluation Function 2 ( Akshatha Jain )
3: Evaluation Function 3 ( Akshatha Jain )
4: Evaluation Function 4 ( Sahana Sreenath)
3
Initial board

             PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       | 6 | 6 | 6 | 6 | 6 | 6 |
       ========================
```

```
store_A                    store_B
------                     ------
|  0 |                    |  0 |
------                     ------
        ========================
        | 6 | 6 | 6 | 6 | 6 | 6 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B



It is player A's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        | 7 | 7 | 7 | 7 | 7 | 0 |
        ========================
store_A                    store_B
------                     ------
|  1 |                    |  0 |
------                     ------
        ========================
        | 6 | 6 | 6 | 6 | 6 | 6 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B



It is player A's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
```

```
        | 8 | 8 | 8 | 8 | 0 | 0 |
        ========================
store_A                    store_B
------                     ------
|  2 |                     |  0 |
------                     ------
        ========================
        | 7 | 7 | 6 | 6 | 6 | 6 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B



It is player B's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        | 8 | 8 | 8 | 8 | 0 | 1 |
        ========================
store_A                    store_B
------                     ------
|  2 |                     |  1 |
------                     ------
        ========================
        | 0 | 8 | 7 | 7 | 7 | 7 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B



It is player A's turn!

              PLAYER A
```

```
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        =========================
        | 8 | 8 | 8 | 8 | 0 | 0 |
        =========================
store_A                    store_B
------                     ------
| 10 |                     |  1 |
------                     ------
        =========================
        | 0 | 8 | 7 | 7 | 0 | 7 |
        =========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B



It is player B's turn!

              PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        =========================
        | 8 | 8 | 8 | 9 | 1 | 1 |
        =========================
store_A                    store_B
------                     ------
| 10 |                     |  2 |
------                     ------
        =========================
        | 0 | 0 | 8 | 8 | 1 | 8 |
        =========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B



It is player A's turn!
```

```
            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        =========================
        | 8 | 8 | 8 | 9 | 2 | 0 |
        =========================
store_A                    store_B
------                     ------
| 10 |                     |  2 |
------                     ------
        =========================
        | 0 | 0 | 8 | 8 | 1 | 8 |
        =========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


It is player B's turn!

            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        =========================
        | 8 | 8 | 9 | 10 | 3 | 1 |
        =========================
store_A                    store_B
------                     ------
| 10 |                     |  3 |
------                     ------
        =========================
        | 0 | 0 | 0 | 9 | 2 | 9 |
        =========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B
```

```
It is player A's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       =========================
       | 8 | 8 | 9 | 10 | 4 | 0 |
       =========================
store_A                    store_B
------                     ------
| 10 |                     |  3 |
------                     ------
       =========================
       | 0 | 0 | 0 | 9 | 2 | 9 |
       =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       =========================
       | 9 | 9 | 10 | 11 | 5 | 1 |
       =========================
store_A                    store_B
------                     ------
| 10 |                     |  4 |
------                     ------
       =========================
       | 0 | 0 | 0 | 0 | 3 | 10 |
       =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B
```

```
It is player A's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       =========================
       | 9 | 9 | 10 | 11 | 6 | 0 |
       =========================
store_A                    store_B
------                     ------
| 10 |                     |  4 |
------                     ------
       =========================
       | 0 | 0 | 0 | 0 | 3 | 10 |
       =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B



It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       =========================
       | 9 | 9 | 10 | 11 | 6 | 1 |
       =========================
store_A                    store_B
------                     ------
| 10 |                     |  5 |
------                     ------
       =========================
       | 0 | 0 | 0 | 0 | 0 | 11 |
       =========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B
```

```
            PLAYER B


It is player A's turn!

            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       | 9 | 9 | 10 | 11 | 7 | 0 |
       ========================
store_A                    store_B
------                     ------
| 10 |                     |  5 |
------                     ------
       ========================
       | 0 | 0 | 0 | 0 | 0 | 11 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


It is player B's turn!

            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       |10 | 10 | 11 | 0 | 8 | 1 |
       ========================
store_A                    store_B
------                     ------
| 10 |                     | 19 |
------                     ------
       ========================
       | 1 | 1 | 1 | 0 | 0 | 0 |
       ========================
```

```
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player A's turn!

              PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       |10 | 10 | 11 | 0 | 9 | 0 |
       ========================
store_A                    store_B
------                     ------
| 10 |                     | 19 |
------                     ------
       ========================
       | 1 | 1 | 1 | 0 | 0 | 0 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player B's turn!

              PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       |10 | 10 | 11 | 0 | 9 | 0 |
       ========================
store_A                    store_B
------                     ------
| 10 |                     | 19 |
------                     ------
       ========================
```

```
        ==========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

                PLAYER B


It is player A's turn!

                PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ==========================
        |11 | 11 | 12 | 1 | 0 | 0 |
        ==========================
store_A                    store_B
------                     ------
| 11 |                     | 19 |
------                     ------
        ==========================
        | 1 | 3 | 2 | 1 | 0 | 0 |
        ==========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

                PLAYER B


It is player B's turn!

                PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ==========================
        |11 | 11 | 12 | 1 | 0 | 0 |
        ==========================
store_A                    store_B
------                     ------
| 11 |                     | 19 |
```

```
------                          ------
       ==========================
       | 0 | 4 | 2 | 1 | 0 | 0 |
       ==========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player A's turn!

              PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ==========================
       |11 | 11 | 13 | 0 | 0 | 0 |
       ==========================
store_A                      store_B
------                       ------
| 11 |                       | 19 |
------                       ------
       ==========================
       | 0 | 4 | 2 | 1 | 0 | 0 |
       ==========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player B's turn!

              PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ==========================
       |11 | 11 | 13 | 0 | 0 | 0 |
       ==========================
store_A                      store_B
```

```
------                    ------
| 11 |                    | 20 |
------                    ------
      ========================
      | 0 | 0 | 3 | 2 | 1 | 0 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


It is player A's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      |12 | 12 | 0 | 1 | 1 | 1 |
      ========================
store_A                   store_B
------                    ------
| 17 |                    | 20 |
------                    ------
      ========================
      | 1 | 1 | 0 | 3 | 2 | 1 |
      ========================
       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


It is player B's turn!

            PLAYER A

       | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
      ========================
      |12 | 12 | 0 | 1 | 1 | 1 |
```

```
        ========================
store_A                  store_B
------                  ------
| 17 |                  | 20 |
------                  ------
        ========================
        | 0 | 2 | 0 | 3 | 2 | 1 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player A's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        |12 | 12 | 0 | 1 | 2 | 0 |
        ========================
store_A                  store_B
------                  ------
| 17 |                  | 20 |
------                  ------
        ========================
        | 0 | 2 | 0 | 3 | 2 | 1 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player B's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
```

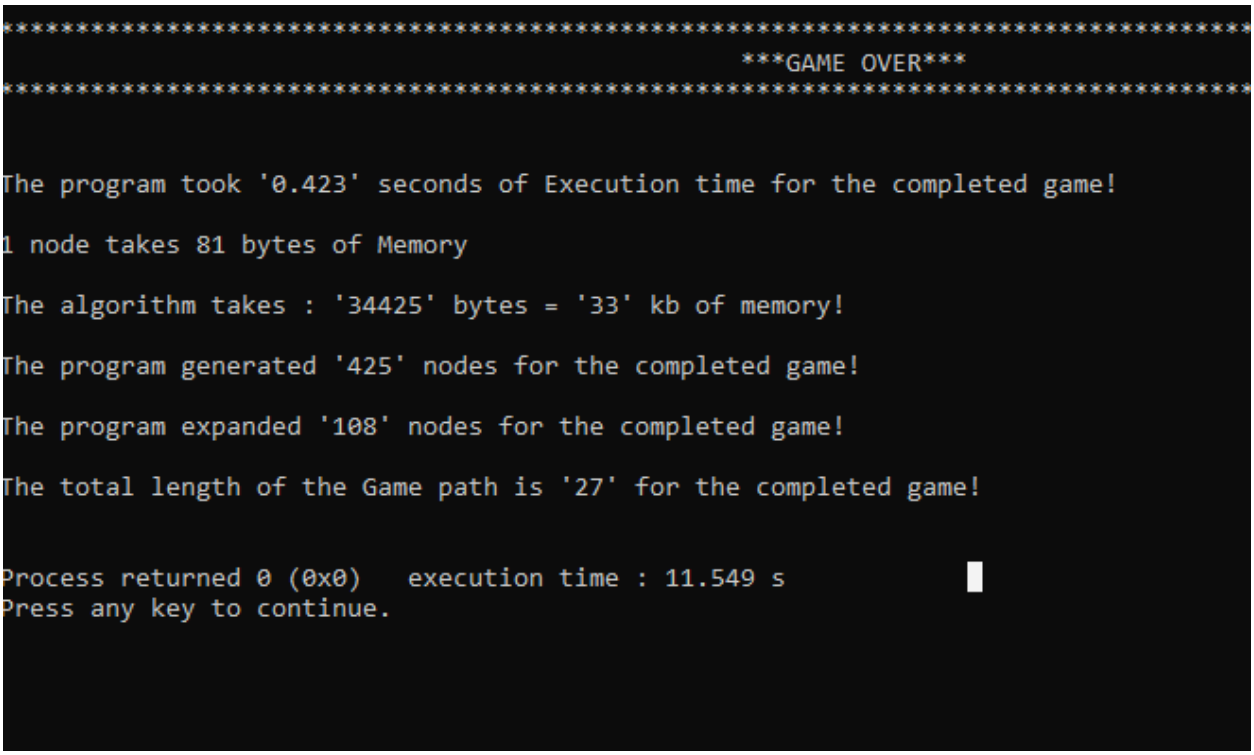
```
        ========================
        |12 | 12 | 0 | 1 | 2 | 0 |
        ========================
store_A                    store_B
------                     ------
| 17 |                     | 20 |
------                     ------
        ========================
        | 0 | 0 | 1 | 4 | 2 | 1 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player A's turn!

              PLAYER A

         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
        ========================
        |12 | 12 | 0 | 2 | 0 | 0 |
        ========================
store_A                    store_B
------                     ------
| 19 |                     | 20 |
------                     ------
        ========================
        | 0 | 0 | 0 | 4 | 2 | 1 |
        ========================
         | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

              PLAYER B


It is player B's turn!

              PLAYER A
```

```
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       |12 | 12 | 0 | 2 | 0 | 1 |
       ========================
store_A                  store_B
------                   ------
| 19 |                   | 21 |
------                   ------
       ========================
       | 0 | 0 | 0 | 0 | 3 | 2 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

             PLAYER B


It is player A's turn!

             PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       |12 | 12 | 0 | 2 | 0 | 0 |
       ========================
store_A                  store_B
------                   ------
| 23 |                   | 21 |
------                   ------
       ========================
       | 0 | 0 | 0 | 0 | 0 | 2 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

             PLAYER B


It is player B's turn!
```

```
            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       |12 | 12 | 0 | 2 | 0 | 1 |
       ========================
store_A                  store_B
------                   ------
| 23 |                   | 22 |
------                   ------
       ========================
       | 0 | 0 | 0 | 0 | 0 | 0 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B


Player B ran out of all the stones! All the pits in B are empty!

            PLAYER A

        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of A
       ========================
       | 0 | 0 | 0 | 0 | 0 | 0 |
       ========================
store_A                  store_B
------                   ------
| 50 |                   | 22 |
------                   ------
       ========================
       | 0 | 0 | 0 | 0 | 0 | 0 |
       ========================
        | 0 | 1 | 2 | 3 | 4 | 5 | ----------> Pit Numbers of B

            PLAYER B
```

```
The program took '0.427' seconds of Execution time for the completed game!

1 node takes 81 bytes of Memory

The algorithm takes : '16686' bytes = '16' kb of memory!

The program generated '206' nodes for the completed game!

The program expanded '54' nodes for the completed game!

The total length of the Game path is '27' for the completed game!


Process returned 0 (0x0)   execution time : 6.981 s
Press any key to continue.
```

OUTPUT 4(Evaluation function4 depth4)



13. References:

Artificial intelligence, A modern approach by S.J.Russel and P.Norvig

Artificial intelligence, by E.Rich, K.Knight, K.B.Nair

https://en.wikipedia.org/wiki/Alpha–beta_pruning

http://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-

pruning/