A Comparative Study of Game Tree Searching Methods

Ahmed A. Elnaggar
Computer Science Department
Modern Academy in Maadi
Cairo, Egypt

Mostafa Abdel Aziem
College of Computing and IT
AAST
Cairo, Egypt

Mahmoud Gadallah
Computer Science Department
Modern Academy in Maadi
Cairo, Egypt

Hesham El-Deeb
Computer Science Faculty
MTI University
Cairo, Egypt

Abstract—In this paper, a comprehensive survey on gaming tree searching methods that can use to find the best move in two players zero-sum computer games was introduced. The purpose of this paper is to discuss, compares and analyzes various sequential and parallel algorithms of gaming tree, including some enhancement for them. Furthermore, a number of open research areas and suggestions of future work in this field are mentioned.

Keywords—game tree search; searching; evaluation; parallel; distributed; GPU

I. INTRODUCTION

Game playing; where a human play versus a computer have a long history. In the earlier game playing programs, the computer couldn't win a human because of the weakness of the game tree algorithms for finding the best next-move or the limitation of computer computation and memory space. The advances in this field and the field of computer architecture finally allowed computers to win humans in most complex games, including chess. Many algorithms have already been invented to find the best next-move for the computer, including sequential algorithms such as MiniMax [1], NegaMax [2], Negascout [3], SSS* [4] and B* [5] as well as parallel algorithms such as Parallel Alpha-Beta algorithm [6]. These Parallel algorithms are now modified to run not only on CPUs, but also on GPUs [7] to provide a faster solution.

Almost all game playing programs use game trees to find the next-best move. An example of a game tree for Tic-Tac-Toe game [8] is shown in Fig. 1 The figure shows the following:

- Each node represents a game state.

- The root represents the current game state.

- All the branches for a given node represent all the legal moves for that node.

- The node that doesn't have any successor called a leaf.

Evaluation function is used to determine whatever a leaf represents a win, lose, draw or just a score, in case the algorithm was stopped before any player won, lose or the game ended with a draw. The developers usually do that because in more complex games, there is no practical algorithm that can search in the entire tree in a reasonable amount of time even if it uses the power of parallel processing. An example for this is the checkers and chess game where they need to evaluate about 10^{20} and 10^{40} nodes respectively. W^D is used as an estimation of the number of nodes needs to be visited, where W represents the average number of legal moves for each node, and D represents the game length. Two solutions for this problem are to use a fixed depth "D" or to use a specific time to stop generating the tree.

There are many categorization methods for sequential game tree. However, the most common categorization is based on depth-first and breadth-first, which was used in this paper. Depth-first search "DFS" [9] means the algorithm will start from the root and explores as long as the depth limitation didn't meet along each branch before backtracking. Breadth-first search "BFS" [10] begins from the root and inspects all the children of the root node before it inspects all the children of each root children's node. The parallel algorithms are hard to categorize as depth or breadth first since some parallel algorithms work as follows: each core or each processor inspects a child of the root using DFS or BFS. In the first case, the distribution of the root's children uses a BFS algorithm, but each core or processor uses a DFS. In this case, it is called a hybrid-system.

To make it clear for the reader, the paper was organized as follows:

Section [II], presents a discussion for sequential game tree algorithms categorized into depth-first & breadth-first algorithms. Section [III], presents a discussion for parallel game tree algorithms from the programming point of view and from the hardware point of view. Section, [IV], provides an analysis for depth algorithms and breadth algorithms based on algorithm complexity for both time and memory. Section [V], concludes the paper with some future work that can enhance the current algorithms.

II. SEQUENTIAL GAME TREE ALGORITHMS

As mentioned in the previous section, sequential algorithms were categorized into depth-first search [9] and breadth-first search [10]. Furthermore, the depth-first search algorithms categorized again into brute-force search and selective search [11]. The brute-force search is looking at every variation to a given depth while the selective search is looking at important branches only. Section [A], presents a discussion for the brute-force search in depth-first search. Section [B], presents a discussion for the selective search in depth-first search. Section [C], presents a discussion for the breadth-first search.
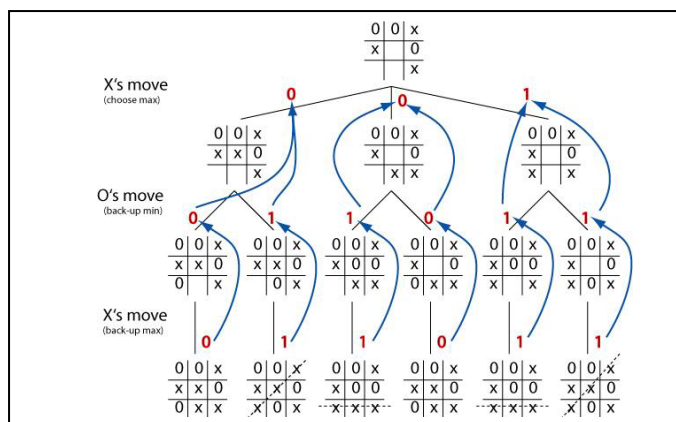

Fig. 1. Game tree for Tic-Tac-Toe game using MiniMax algorithm.

A. Brute-Force algorithms in Depth-First Search

The most famous algorithms in brute-force search are MiniMax [1], NegaMax [2], Alpha-Beta [12], NegaScout [3], and Principle-Variation [13]. Following is a description of each of these algorithms.

MiniMax [1] algorithm is a game tree algorithm that is divided into two logically stages, the first one for the first player which is the computer and the second one for the second player which is the human. The algorithm tries to find the best legal move for the computer even if the human plays the best move for him. Which means, it maximizes the computer score when it chooses the computer move, while minimizing that score by choosing the best legal move for the human when it chooses the human move.

In Fig. 1 there is a simulation of MiniMax search for Tic-Tac-Toe game [8]. Every node has a value calculated by the evaluation function. For a given path, the value of the leaf nodes passed back to its parent. An example for that the value of any O's move will always choose the minimum value for the computer, while the value for any X's move will always choose the maximum value for the computer.

In Fig. 2 a pseudo code for the MiniMax algorithm [1] is presented. The program first calls the function MiniMax which starts the chain of calls for MaxMove and MinMove. Each time MaxMove function or MinMove function is called it automatically calls the other function until the game ended, or it reached the desired depth.

NegaMax [2] algorithm is an identical algorithm for MiniMax with only one slight difference. It uses only the

maximization function rather than using both maximization and minimization functions. This can be done by negating the value that is returned from the children from the opponent's point of view rather than searching for the minimum score. This is possible because of the following mathematical relation:

$$Max (a, b) == -Min (-a, -b) \qquad (1)$$

```
MinMax (GamePosition game) {
  return MaxMove (game);
}

MaxMove (GamePosition game) {
  if (GameEnded(game)) {
    return EvalGameState(game);
  }
  else {
    best_move < - {};
    moves <- GenerateMoves(game);
    ForEach moves {
      move <- MinMove(ApplyMove(game));
      if (Value(move) > Value(best_move)) {
        best_move < - move;
      }
    }
    return best_move;
  }
}

MinMove (GamePosition game) {
  best_move <- {};
  moves <- GenerateMoves(game);
  ForEach moves {
    move <- MaxMove(ApplyMove(game));
    if (Value(move) > Value(best_move)) {
      best_move < - move;
    }
  }

  return best_move;
}
```

Fig. 2. MiniMax Algorithm Pseduo Code

In Fig. 3 there is a pseudo code for NegaMax algorithm. Clearly, (1) was used to simplify the MiniMax algorithm.

Alpha-Beta [12] algorithm is a smart modification that can be applied to MiniMax or NegaMax algorithms. Kunth and Moore proved that many branches could be pruned away of the game tree which reduces the time needed to finish the tree, and it will give the same result as MiniMax or NegaMax. The main idea of the algorithm is cutting the uninteresting branches in the game tree. The following examples illustrating the idea: Max (8, Min (5, X)) and Min (3, Max (7, Y)) The result is always 8 in the first example and 3 in the second example, no matter the values of X or Y. This means the algorithm can cut the node X or Y with its branches. The Alpha-Beta algorithm uses two variables (alpha & beta) to detect these cases, so any value less than alpha or larger than

beta will automatically cutoff without affecting the result of the search tree.

The enhanced version of the NegaMax algorithm from Fig. 3 with Alpha-Beta property is shown in Fig. 4.

```
// Search game tree to given depth, and return evaluation of
// root node.
int NegaMax(gamePosition, depth)
{
        if (depth=0 || game is over)
                // evaluate leaf gamePositionition from
                // current player's standpoint
                return Eval (gamePosition);
        // present return value
        score = - INFINITY;
        // generate successor moves
        moves = Generate(gamePosition);
        // look over all moves
        for i =1 to sizeof(moves) do
        {
                // execute current move
                Make(moves[i]);
                // call other player, and switch sign of
                // returned value
                cur = -NegaMax(gamePosition, depth-1);
                // compare returned value and score
                // value, update it if necessary
                if (cur > score) score = cur;
                // retract current move
                Undo(moves[i]);
        }
        return score;
}
```

Fig. 3.   NegaMax Algorithm Pseudo Code

Several enhancements for the Alpha-Beta algorithm was published [13]; some of them is listed as follows:

- Move Ordering: The speed and the number of cutoffs of the Alpha-Beta algorithm can change dramatically depending on the moving search order. The best move should be examined first, and then the second best move and so on. This will maximize the effectiveness of the algorithm. Many techniques developed to solve this problem, including:

  o   Iterative deepening.

  o   Transposition tables.

  o   Killer Move Heuristic.

  o   History Heuristic.

- Minimal Window Search: Alpha-Beta algorithms depend on the values of alpha and beta to cutoff the branches, so by narrowing the search window by changing the values of alpha and beta; it will increase the possibilities of the cutoffs. Many algorithms such as NegaScout [3] and MTD (f) [14] used this property to

improve the Alpha-Beta algorithm which discussed below.

```
// Search game tree to given depth, and return evaluation of
// root node.
int AlphaBeta(gamePosition, depth, alpha, beta)
{
        if (depth=0 || game is over)
        // evaluate leaf gamePositionition from
        // current player's standpoint
                return Eval (gamePosition);
         // present return value
        score = - INFINITY;
        // generate successor moves
        moves = Generate(gamePosition);
        // look over all moves
        for i =1 to sizeof(moves) do
        {
                // execute current move
                Make(moves[i]);
                // call other player, and switch sign of
                // returned value
                cur = -AlphaBeta(gamePosition, depth-1,
                                -beta, -alpha);
                // compare returned value and score
                // value, update it if necessary
                if (cur > score) score = cur;
                // adjust the search window
                if (score > alpha) alpha = score;
                // retract current move
                Undo(moves[i]);
                 // cut off
                if (alpha >= beta) return alpha;
        }
        return score;
}
```

Fig. 4.   Enhanced NegaMax with Alpha-Beta Property Pseudo Code

The NegScout [3] and Principal Variation Search [13] algorithms were based on the scout algorithm which was an enhanced version of the Alpha-Beta algorithm that can make more cutoffs in the game tree. It contains a new test condition that checks whatever the first node in the siblings is either less than or equal to beta value or greater than or equal to the alpha value. If the result of the condition is true, then the algorithm cuts off the root node for these siblings, and if it is false, then it searches the rest of the siblings to get the new values of alpha and beta.

In Fig. 5 there is a pseudo code for the NegaScout [3]. It looks like the same algorithm in Fig. 4 with the modification of the minimal window search.

B. Selectivity algorithms in Depth-First Search

The main difference between the brute-force algorithms and the selectivity algorithms; it doesn't depend on fixed depth to stop looking in each branch. The most common techniques in this category are Quiescence Search and Forward Pruning.

Below is a discussion of the implementation of the Quiescence Search technique [15] and ProbCut [16] algorithm that is based on the Forward Pruning technique.

```
// Search game tree to given depth, and return evaluation of
// root node.
int NegaScout(gamePosition, depth, alpha, beta)
{
        if (depth=0 || game is over)
        // evaluate leaf gamePositionition from
        // current player's standpoint
                return Eval (gamePosition);
         // present return value
        score = - INFINITY;
        n = beta;
        // generate successor moves
        moves = Generate(gamePosition);
        // look over all moves
        for i =1 to sizeof(moves) do
        {
                // execute current move
                Make(moves[i]);
                // call other player, and switch sign of
                // returned value
                cur = -NegaScout(gamePosition, depth-1,
                            -n, -alpha);
                if (cur > score) {
                        if (n = beta ) OR (d <= 2)
                        // compare returned value and
                        // score value, update it if
                        // necessary
                                score = cur;
                        else
                                score = -NegaScout
                                (gamePosition,depth-1,
                                 -beta, -cur);
                }
                // adjust the search window
                if (score > alpha) alpha = score;
                // retract current move
                Undo(moves[i]);
                 // cut off
                if (alpha >= beta) return alpha;
                n = alpha+1;
        }
        return score;
}
```

Fig. 5. NegaScout Algorithm Pseudo Code Using the Minimal Window Search Principle

Quiescence Search [15] based on the idea of variable depth searching. The algorithm follows the normal fixed depth in most branches. However, in some branches the algorithm takes a deeper look and increases the search depth. An example of that is the chess game where in critical moves like checks or promotions, the algorithms extend the depth to make sure there is no threat exists.

In Fig. 6 there is an abstract pseudo code for the Quiescence Search that extends the depth and checks if there is any capture for pieces after a specific move or not.

```
int Quiesce( int alpha, int beta ) {
    int stand_pat = Evaluate();
    if( stand_pat >= beta )
        return beta;
    if( alpha < stand_pat )
        alpha = stand_pat;

    until( every_capture_has_been_examined )  {
        MakeCapture();
        score = -Quiesce( -beta, -alpha );
        TakeBackMove();

        if( score >= beta )
            return beta;
        if( score > alpha )
            alpha = score;
    }
    return alpha;
}
```

Fig. 6. Abstract Pseudo Code Version for Quiescence Search

Forward Pruning technique completes the idea of the variable depth, where it cuts-off unpromising branches. However, this can lead to errors in the result. Many algorithms implemented the idea of this technique, including N-Best Selective Search, ProbCut and Multi-ProCut [16].

N-Best Selective Search only looks for the best N-best moves at each node. All other siblings for the N-best moves will automatically cutoff.

Both ProbCut Multi-ProCut algorithms use the result of shallow search to determine the possibility that a deeper search will change the value of alpha and beta or not.

ProbCut [16] algorithm uses the statistics' correlation techniques to cutoff branches, because it was discovered that there is a strong correlation between values obtained from different depth. The relation was described by Micheal Buro as follows:

$$V\_D = a * V\_D' + b + e \qquad (2)$$

Where V_D means the value of a given depth, a & b are real numbers and e is a normally distributed error with zero mean.

Since the value of $a \approx 1$, $b \approx 0$ and σ^2 is small in most stable evaluation function, the probability of V_D >= beta could be predicted from the following equivalent equation:

$$V\_D' >= ((1/ \Phi(P))*\sigma + beta - b) / a \qquad (3)$$

Furthermore, the probability of V_D <= alpha could predicted from the following equivalent equation:

$$V\_D' <= (-(1/\Phi(P)*\sigma + alpha - b) / a \qquad (4)$$

In Fig. 7 there is an abstract implementation for the ProbCut algorithm. Remember it's up to you to choose the values of D, D', the cutoff threshold $(1/\Phi(P))$, a, b and σ. The algorithm can provide a faster result than any brute-force algorithm. However, it needs many accurate parameters, which may be difficult to choose and may lead to errors in the results.

```
int alphaBetaProbCut(int α, int β, int depth) {
        const float T(1.5);
        const int DP(4);
        const int D(8);

        if ( depth == 0 ) return evaluate();

        if ( depth == D ) {
                int bound;

                // v >= β with prob. of at least p?
                // yes => cutoff */
                bound = round( ( T * σ + β - b) / a );
                if ( alphaBetaProbCut( bound-1, bound,
                                DP) >= bound )
                        return β;

                // v <= α with prob. of at least p?
                // yes => cutoff */
                bound = round( (-T * σ + α - b) / a );
                if ( alphaBetaProbCut( bound, bound+1,
                                DP) <= bound )
                        return α;
        }
        // the remainder of alpha-beta goes here
        ...
}
```

Fig. 7. Abstract Pseudo Code Version for ProbCut Search without the alpha-beta implementation

Multi-ProbCut [16] algorithms generalize the idea of ProbCut by using additional cutoff thresholds and checks, including allowing more regression parameters and cutoff thresholds, using many depth pair and using internal iterative deepening for shallow searches.

C. Breadth-First Search

As mentioned before the BFS begins from the root node then it visits its first child after that it visits all its siblings from the same depth before it moves to the next depth. One of the problems with this technique; it requires huge memory to store node's data. Many algorithms use this technique like NegaC* [17], MTD (f) [14], SSS* [4], B* [5] and Monte-Carlo search [18] algorithms, which discussed below.

NegaC* [17] algorithm uses the minimal-window with fail-soft Alpha-Beta algorithm like NegaScout [3] algorithm, but it parses the tree in Breadth-First way. The fail-soft technique uses two more variables than the Alpha-Beta algorithms to cutoff more branches.

In Fig. 8 there is an abstract pseudo code implementation of the NegaC* algorithm.

```
int negaCStar (int min, int max, int depth) {
        int score = min;
        while (min != max) {
                alpha = (min + max) / 2;
                score = failSoftAlphaBeta (alpha,
                                alpha + 1, depth);
                if ( score > alpha)
                        min = score;
                else
                        max = score;
        }
        return score;
}
```

Fig. 8. An Abstract Pseudo Code Implementation of the NegaC* Algorithm

MTD (f) [14] algorithm, which is an abbreviation for "Memory-enhanced Test Driver" also uses the minimal-window technique like NegaScout [3] algorithm, but it does it efficiently. It was introduced as an enhancement to the Alpha-Beta Algorithm as mentioned before. It also uses two more variables to determine the upper-bound and lower-bound. The normal Alpha-Beta algorithm uses only alpha and beta variables with $-\infty$ & ∞ as a start and the values are updated one time at each call to make the only one returning value lies between the alpha and beta values. However, MTD (f) may search more than one time at each Alpha-Beta [12] call and use the returned bounds to converge toward it using the lower-bound and upper-bound to make faster cutoffs of the tree. Furthermore, the algorithm uses a transposition table to store and retrieve data about portions of the search tree to use it later to reduce the over-head of re-examining same game state. However, it uses memory space to store this data, which required more memory space.

Fig. 9 shows a pseudo code for the MTD (f) algorithm without the implementation of the Alpha-Beta algorithm which was described in Fig. 4.

SSS* [4] is another famous breadth-first search algorithm, which is non-directorial search algorithm. The algorithm expands into multiple paths at the same time to get global-information of the search tree. However, it searches fewer nodes than fixed depth-first algorithms like Alpha-Beta algorithm.

The algorithm stores' information for all active nodes which didn't solve yet in a list in decreasing order depends on their importance. The information consists of three parts:

- N: a unique identifier for each node.

- S: a status of each node whatever it's live or has been solved.

- H: an important value for the node.

```
function MTDF(root, f, d){
        g := f
        upperBound := +∞
        lowerBound := -∞
        while lowerBound < upperBound{
                if g = lowerBound then
                        β := g+1
                else
                        β := g
                g := AlphaBetaWithMemory (root,  β-1,
                                          β, d)
                if g < β then
                        upperBound := g
                else
                        lowerBound := g
        }
        return g
}
```

Fig. 9. Pseudo Code for the MTD (f) Algorithm Without the Implementation of the Alpha-Beta algorithm Which was Described Previously in Fig. 4

The core of the SSS* [4] algorithm depends on two phases:

- Node Expansion: Top-down expansion of a Min strategy.

- Solution: Bottom-up search for the best Max strategy.

Fig. 10 shows the pseudo code for the SSS* algorithm using three function push, pop and insert to store, remove and update node information.

Monte-Carlo Tree Search "MCTS" [18] algorithm made a breakthrough in game theory and computer science field. The algorithm is based on randomized exploration of the game tree. The algorithm also uses the results of previous examined values for nodes. Every time the algorithm runs it produces a better estimation of values. However, the game tree gradually grows in the memory, which is the main disadvantages of breadth-first algorithms.

The algorithm consists of four phases, which is repeated as long as there is still time for the computer to think:

- Selection phase, it starts from the root node; it traverses the game tree by selecting the most promising move until reaching a leaf node.

- Expansion phase, if the number of visits reaches a pre-determined threshold, the leaf is expanded to build a larger tree.

- Simulation phase, calculates the outcome value of the leaf by performing a play-out at it.

- Back-propagation phase, it traces back along the game tree path from the leaf to the root to update the values changed in the simulation phase.

```
int SSS* (node n; int bound)
{
   push (n, LIVE, bound);
   while ( true ) {
     pop (node);
     switch ( node.status ) {
     case LIVE:
        if (node == LEAF)
          insert (node, SOLVED, min(eval(node),h));
        if (node == MIN_NODE)
          push (node.1, LIVE, h);
        if (node == MAX_NODE)
          for (j=w; j; j--)
            push (node.j, LIVE, h);
        break;
     case SOLVED:
        if (node == ROOT_NODE)
          return (h);
        if (node == MIN_NODE) {
          purge (parent(node));
          push (parent(node), SOLVED, h);
        }
        if (node == MAX_NODE) {
          if (node has an unexamined brother)
            push (brother(node), LIVE, h);
          else
            push (parent(node), SOLVED, h);
        }
        break;
     }
   }
}
```

Fig. 10. Pseudo Code for the SSS* Algorithm

Fig. 11 shows the pseudo-code for MCTS algorithm using the four phases described before.

B* [5] is the final algorithm that will be described in the BFS. It finds the least-cost path from the node to any goal node out of one of more possible goals. The main idea of this algorithm is based on:

- Stop when one path is better than all the others.

- Focus the exploration on paths that will lead to stopping.

The algorithm expands the searching based on prove-best and disprove-rest strategies. In prove-best strategy, the algorithm chooses the node with the highest upper-bound because it has a high probability to raise its lower bound higher than any other nodes' upper-bound when it expands. On the other hand, the disprove-rest strategy chooses the next highest upper-bound node because it has a good probability to reduce the upper-bound to less than the lower-bound of the best child when it expands.

```
Data: root node
Result: best move
while (has time) do
    current node ← root node

    /* The tree is traversed
    while (current node ϵ T ) do
        last node ← current node
        current node ← Select(current node)
    end

    /* A node is added
    last node ← Expand(last node)

    /* A simulated game is played
    R ← P lay simulated game(last node)

    /* The result is backpropagated
    current node ← last node
    while (current node ϵ T ) do
        Backpropagation(current node, R)
        current node ← current node.parent
    end

end

return best move = argmaxN ϵ Nc (root node)
```

Fig. 11. Pseudo-Code for MCTS Algorithm

In the next section, a discussion of most famous parallel game tree search algorithms is presented.

III. PARALLELISM IN GAME TREE SEARCH

The technological advances of the computer architecture and the release of multiprocessors and multi-core computers, allows algorithms that can be partitioned into independent segments to be solved faster. Many enhancements were done on the sequential algorithms to make it run in parallel as well as new algorithms are designed for parallel computing. The problem of parallel computing is the trade-off between the overhead of communication and synchronization and the benefits of exploring many nodes in the same time in parallel. This made the speedup is sub-linear rather than linear. Section [A], presents a discussion of various techniques & algorithms made to solve these problems. Section [0], presents a discussion for the parallelism of the game search tree from the hardware point of view.

A. Game Tree Parallelism Techniques & Algorithms

As mentioned before many techniques were designed to solve the overhead problem. One of them is the "Shared Hash Table" technique, which stores information about nodes in the game tree so it could be used by any processor or core in the system. This reduces the communication over-head between processors or cores, especially if the processors are not on the same physical computer.

Many algorithms use the previous technique as well as other techniques, including ABDADA, Parallel Alpha-Beta, Parallel PVS, YBWC and Jamboree and Dynamic Tree Splitting. Next, a description of each algorithm is presented.

ABDADA is a loosely synchronized and distributed search algorithm designed by Jean-Christophe. The algorithm uses the shared hash table technique as well as adding more information for each node like the number of processors searching this node.

Parallel Alpha-Beta [6] is the parallel version of the previously discussed Alpha-Beta algorithm. The basic idea is splitting the search tree into sub-search trees and run each one in specific core or more in case of multi-core and one or more processor in case of multi-processor system. The problem of this algorithm is the complexity of implementing it. However, it can maximize the utilization of the cores or processors. The two methods of splitting the tree are showing in Fig. 12.
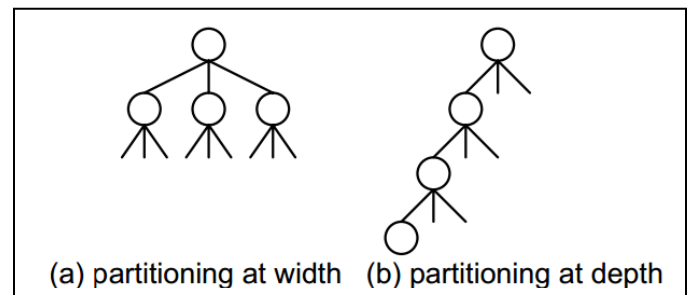

Fig. 12. Partitioning of the Game Tree for Alpha-Beta Search

PVS [13] algorithm expresses each node as a thread which can run in parallel. However, before running them in parallel, the problem of data dependency that exists among threads must be solved. A simple solution is to get the initial required value from the first node among any siblings then run the remaining siblings in parallel. The sequential and parallel tasks for PVS algorithm using two processors is showing in Fig. 13 while a pseudo code for the algorithm is showing in Fig. 14.
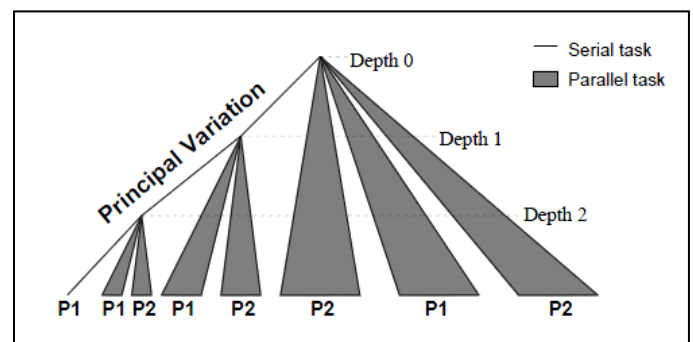

Fig. 13. Sequential and Parallel Tasks by Two Processors using PVS

YBWC and Jamboree algorithm [19] is shown in ,which based on a recursive algorithm that visits the first node in siblings before spawning the remaining sibling's nodes in parallel. It uses this technique because the first node may produce a cutoff so it does not waste the others processors' time by searching in the sibling nodes or will produce better

bounds, then the algorithm search the remaining siblings in parallel. A pseudo code of the algorithm is showing in Fig. 15.

```
PVSplit (Node curnode, int alpha, int beta, int result)
{
        if(cur_node.is_leaf)
                return Evaluate(cur_node);

        succ_node = GetFirstSucc(cur_node);
        score = PVSplit(curnode, alpha, beta);

        if(score > beta)
                return beta;
        if(score > alpha)
                alpha = score;

        //Begin parallel loop
        while(HasMoreSuccessors(curnode))
        {
                succ_node = GetNextSucc(curnode);
                score = AlphaBeta(succnode,alpha,beta);

                if(score > beta)
                        return beta;
                if(score > alpha)
                        alpha = score;
        }//End parallel loop
        return alpha;
}
```

Fig. 14. Pseudo Code for PVS Algorithm Using Alpha-Beta Function

Finally, DTS algorithm [20] is the most complex parallel game tree search algorithm and there are a few implantation of it. However, it gives the best performance in symmetric multi-processors systems. A pseudo-code of the DTS algorithm presented in Fig. 16.

Table I shows the speedup for the three algorithms compared using 1, 2, 4, 8, and 16 processors.

TABLE I.   POPULAR GAME TREE PARALLEL ALGORITHMS SPEEDUP

Algorithm	Number of Processors				
	1	*2*	*4*	*8*	*16*
PVS	1.0	1.8	3.0	4.1	4.6
YBWC	1.0	1.9	3.4	6.1	10.9
DTS	1.0	2.0	3.7	6.6	11.1

All the previous parallel algorithms need a parallel programming language or library that can handle threads and parallel computing. Many libraries and programming languages were released to support CPU parallelism in general. However, the most famous library that used in parallel game tree algorithms is MPI (Message Passing Interface) [21] which is an extension of C programming language. MPI handles the burden of synchronization, communication and distributed resources management. The latest version of MPI is version 2 which supports C++ and object-oriented programming.

```
jamboree(CNode n, int α, int β, int b)
{
        if (n is leaf)
                return static_eval(n);

        c[] = the childen of n;
        b = -jamboree(c[0], -β, -α);
        if (b >= β) return b;
        if (b > α) α = b;

        In Parallel: for (i=1; i < |c[]|; i++)
        {
                s = -jamboree(c[i], -α - 1, -α);

                if (s > b)
                        b = s;
                if (s >= β
                        abort_and_return s;
                if (s > α)
                {
                //Wait for the completion of previous
                iterations of the parallel loop
                        s = -jamboree(c[i], -β, -α);

                        if (s >= β)
                                abort_and_return s;
                        if (s > α)
                                α = s;
                        if (s > b)
                                b = s;
                }
        }
        return b;
}
```

Fig. 15. Pseudo Code for Jamboree Algorithm

```
DTS(root)
{
        while (Stopping_criterion() == false)
        {
                //One processor search to ply = N
                SearchRoot(root);

                //Detect free processors, and begin tree split
                Split(node v);

                //Initialize new threads.
                ThreadInit();

                //Copy a "split block" to begin a new search
                CopytoSMP(node v);
                SearchSMP(node v);
        }
        ThreadStop();
}
```

Fig. 16. Pseudo Code for DTS Algorithm

Another trending library for artificial intelligence algorithms that runs on CPU is Microsoft Task Parallel Library (TPL) [22]. This library uses the concept of finite CPU-bound computation based on task notation and the concept of replicating task using work-stealing technique. It is more effective to develop parallel algorithms such as DTS and YBWC.

Other programming libraries were designed to run parallel algorithms in GPU rather than CPU, including CUDA [23]. However, few algorithms were implemented using these libraries because of the complexity of programming search trees using these libraries. On the other hand, the implemented algorithms that tested on GPU showed a better speedup than the CPU.

B. CPU & GPU for Parallel Game Tree Search

In the previous fifteen years, all researches focused on designing parallel algorithms that can run in parallel on multi-cores or multi-processors. However, the new trend of search trees field is to design and implement algorithms that can run on parallel on the GPU. Early the GPU was built just for graphic computing.

However, in the last 10 years the GPU became a platform for general parallel processing computing. The idea of GPU is to have hundreds or thousands of simple cores that can run threads in parallel with higher GFLOPS than the CPU. On the other hand, the CPU contains few powerful cores or few powerful multi-processors that can run more instructions and have a faster clock speed than the GPU. As mentioned before few algorithms were modified to support GPU. However, in the next five years many AI algorithms will designed to use the power of GPUs. Fig. 17 shows the CPU and GPU architecture.
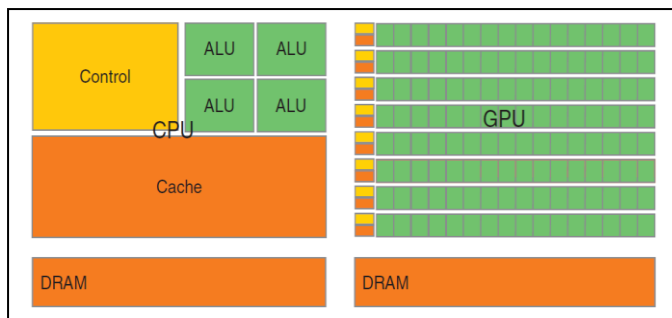

Fig. 17. Difference Between CPU and GPU Architecture

In the next section, an analysis of the previous algorithms based on several criteria is presented.

IV. GAME TREE ALGORITHMS ANALYSIS

As the game tree was categorized according to specific criteria, the evaluation of the previous algorithms is categorized according to specific criteria ; which are:

- Completeness: whatever if the algorithm finds a solution if one exists.
- Time Complexity: the number of nodes generated.

- Space Complexity: the maximum number of nodes in memory during the search.
- Optimality: whatever if the algorithm always finds the least-cost or the best solution.

The following terms were used to measure the time and space complexity:

- B: maximum branches factor.
- D: depth of the best solution.
- M: maximum depth of the state space.
- L: depth cut-off

Table II compares the various sequential algorithm categories based on the previous criteria.

TABLE II.   SEQUENTIAL ALGORITHMS ANALYSIS

Criterion	Breadth-First	Depth-First	Depth-Limited	Iterative Depending
Completeness	Yes	No	Yes, if l >= d	Yes
Time	b^d	b^m	b^l	b^d
Space	b^d	bm	bl	bd
Optimality	Maybe	No	No	Maybe

All algorithms that fall into any of the categories should match the mentioned table criteria value. However, this is the worst-case scenario. In most cases, a better space and time values may be found. An example of that is the Alpha-Beta algorithm where the average time is equal $b^{3m/4}$.

The complexity analysis of parallel game tree searching algorithms is more difficult than the sequential algorithms. Usually, the parallel game tree searching algorithms could be analyzed in terms of:

- Time complexity T (n): How many times steps are needed?
- Processor complexity P (n): How many processors are used?
- Work complexity W (n): What is the total work done by all the processors?

The sequential Minimax algorithm with alpha-beta modification has a time complexity of $O(b^m)$ in the worst case, $O(b^{3m/4})$ in the average case and $O(b^{m/2})$ in the best case; where b is the branching factor and m is the maximum depth of the search tree. The total work done by the parallel alpha-beta algorithm equals the total work done by the sequential algorithm. Hence, table III summarizes the time complexity of the parallel alpha-beta algorithm in the best, average and worst case.By increasing the number of processors or cores, the algorithm could provide a better speed-up. However, it will never reach the ideal speedup, because of the communication overhead between nodes for sharing alpha and beta values as well as the synchronization overhead.

TABLE III.　　PARALLEL ALGORITHMS ANALYSIS

Criterion	Case	Complexity
Time Complexity	worst	$T(n) = O\left(\dfrac{b^m}{p}\right)$
	average	$T(n) = O\left(\dfrac{b^{3m/4}}{p}\right)$
	best	$T(n) = O\left(\dfrac{b^{m/2}}{p}\right)$
Processor Compleixty	worst	$P(n) = O(n)$
	average	
	best	
Work Compleixty	worst	$W(n) = O(b^m)$
	average	$W(n) = O(b^{3m/4})$
	best	$W(n) = O(b^{m/2})$

Besides the normal criteria that will make you choice any game searching algorithms; which includes the amount of memory you have, if you need an optimal solution or not, and the time you need to solve, etc.. An important criterion is the nature of the problem or the nature of the game and the nature of the hardware architecture you have.

V. CONCLUSIONS & FUTURE WORK

A discussion of various game tree-searching algorithms was presented in this paper, including sequential and parallel algorithms. The popular sequential algorithms were covered in details the common algorithms in both depth-first and breadth-first. Furthermore, an overview of common parallel algorithms as well as the hardware architecture for parallel game tree searching was presented. In the end, an analysis the algorithms based on four criteria was discussed.

The use of service-oriented approach to expand the searching trees into a distributed system will solve many distributed issues, while using tasks technology rather than threads to implement the parallel algorithms to maximize the utilization of multi-core computers. Another suggestion is to implement the search algorithms using the OpenCL library which allows the code to run on both GPU and CPU, or CUDA library to produce a massive parallel game tree searching algorithm.

The new feature of dynamic parallelism in CUDA v5.5 allows recursion based algorithms to run faster on the GPU, by eliminating the CPU initialization time of each kernel. Furthermore, the unified memory in CUDA 6.0 creates a pool of management memory that is shared between the CPU and GPU, which make the development of complex games easier. Both the dynamic parallelism and unified memory features can improve the speedup of current AI game tree searching algorithms.

REFERENCES

[1] S. Russell and P. Norvig, Artificial intelligence: a modern approach, 3rd ed. Prentice Hall Press, 2009, p. 1152.

[2] G. T. Heineman, G. Pollice, and S. Selkow, "Path Finding in AI," in Algorithms in a Nutshell, 1st ed., O'Reilly Media, 2008, pp. 213–217.

[3] H.-J. Chang, M.-T. Tsai, and T. Hsu, "Game Tree Search with Adaptive Resolution," in Advances in Computer Games SE - 26, vol. 7168, H. J. Herik and A. Plaat, Eds. Springer Berlin Heidelberg, 2012, pp. 306–319.

[4] [4] U. Lorenz and T. Tscheuschner, "Player Modeling, Search Algorithms and Strategies in Multi-player Games," in Advances in Computer Games SE - 16, vol. 4250, H. J. Herik, S.-C. Hsu, T. Hsu, and H. H. L. M. (Jeroen. Donkers, Eds. Springer Berlin Heidelberg, 2006, pp. 210–224.

[5] Y. Tsuruoka, D. Yokoyama, and T. Chikayama, "Game-Tree Search Algorithm Based on Realization," ICGA J., vol. 25, no. 3, pp. 145–152, 2002.

[6] V. Manohararajah, "Parallel Alpha-Beta Search on Shared Memory Multiprocessor," Computer Engineering University of Toronto, 2001.

[7] D. Strnad and N. Guid, "Parallel alpha-beta algorithm on the GPU," CIT, vol. 19, no. 4, pp. 269–274, 2011.

[8] J. Habgood and M. Overmars, "Clever Computers: Playing Tic-Tac-Toe," in The Game Maker's Apprentice SE - 13, Apress, 2006, pp. 245–257.

[9] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Depth-first search," in Introduction to Algorithms, 3rd ed., MIT Press, 2009, pp. 603–612.

[10] W. Ertel, "Search, Games and Problem Solving," in Introduction to Artificial Intelligence SE - 6, Springer London, 2011, pp. 83–111.

[11] M. Schadd, "Selective Search in Games of Different Complexity," Maastricht University, 2011.

[12] D. Knuth, Selected Papers on Analysis of Algorithms. California: Center for the Study of Language and Information, 2000.

[13] M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and E. C. D. van der Werf, "Enhanced forward pruning," Inf. Sci. (Ny)., vol. 175, no. 4, pp. 315–329, 2005.

[14] K. Shibahara, N. Inui, and Y. Kotani, "Adaptive Strategies of MTD-f for Actual Games," in CIG, 2005.

[15] M. Schadd and M. Winands, "Quiescence Search for Stratego," in BNAIC, 2009, pp. 225–232.

[16] A. X. Jiang and M. Buro, "First Experimental Results of ProbCut Applied to Chess," Adv. Comput. Games, vol. 10, 2003.

[17] D. Rutko, "Fuzzified Tree Search in Real Domain Games," in Advances in Artificial Intelligence SE - 13, vol. 7094, I. Batyrshin and G. Sidorov, Eds. Springer Berlin Heidelberg, 2011, pp. 149–161.

[18] J. Hashimoto, A. Kishimoto, K. Yoshizoe, and K. Ikeda, "Accelerated UCT and Its Application to Two-Player Games," Adv. Comput. Games, 2011.

[19] [19] J. Steenhuisen, "Transposition-Driven Scheduling in Parallel Two-Player State-Space Search," Delft University of Technology, 2005.

[20] T. A. N. Ying, L. U. O. Ke-lu, C. Yu-rong, and Z. Yi-min, "Performance Characterization of Parallel Game-tree Search Application Crafty," vol. 4, no. 2, pp. 2–7, 2006.

[21] D. Jakimovska, G. Jakimovski, A. Tentov, and D. Bojchev, "Performance estimation of parallel processing techniques on various platforms," in Telecommunications Forum (TELFOR), 2012 20th, 2012, pp. 1409–1412.

[22] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," Acm Sigplan Not., vol. 44, no. 10, pp. 227–242, Oct. 2009.

[23] D. B. Kirk and W. M. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2012, p. 496.