ECE – 457
Applied Artificial Intelligence

Project 2 – Adversarial Search

Prepared by

Fation Shahinas 20169057
Hasan Toplar 20162062

Oct 24, 2008

Introduction

This project analyzes and implements different adversarial search techniques and evaluation functions. In this project, an agent is implemented to play Conga, a simple board game, against a random playing opponent.

The dimensions for the board are set to be 4x4 and each player starts with 10 stones at their predefined positions (Player 1 @ 1x4 and Player 2 @ 4x1). The rules of the game are outlined on the project description.

Objective

The objective is to use different adversarial search algorithms such as Minimax and Alpha-Beta pruning to analyze each state that the agent is currently in and compute all the possible directions that it can move. After evaluation in possible movement with an evaluation function, the best move amongst all is chosen and the agent makes a move in this direction. The ultimate goal of the algorithm is the make the agent win the game by using adversarial search approaches.

Overview

This project was coded in C++ in Visual Studio.

Each of the 16 squares on board were declared as a structure called square. Below is the definition of this structure.

```
typedef struct Node_t {
int xpos;
int ypos;
int occby;
int s;
bool c;
} square;
```

Upon initialization, creategame() function is called and each square is updated with its position info (xpos,ypos), a occby variable which indicates the current occupant of the square (0= Unoccupied; 1=player 1; 2= player 2). Boolean c indicates if this particular has been visited by the search algorithm or not. This variable is very useful when performing the searches recursively. Variable s shows the number of stones that particular node currently has.

Also, there exists another structure type called a game. Below is the definition of this structure.

```
typedef struct game_t {
q que;
```

int h;
*struct game_t *parent;*
*struct game_t *child;*
} game;

Each instance of the game is created as a different structure. In other words, for each different possible movement that the agent can take, a game structure which holds the information for all its cells is created. Information associated with each cell is stored in to a 4x4 queue named *que*. After the initialization phase, the agents start playing the game. Player 2 always plays a random move. This is done by using the built in rand() function in C++. Player 1, or MAX, on the other hand, makes a decision to which direction to move by using Minimax with Alpha-Beta Pruning and one of the heuristic evaluation functions used below.

The search tree in both of the search algorithm cases are implemented recursively. This is the same structure used in project 1 since Minimax uses the depth first search approach.

Search Techniques Implementation

Minimax Algorithm

The Minimax algorithm works on the concept of picking the most favorable action at a current state. At any given state of the game, the algorithm analyzes all the legal moves that the agent can take. And for each of these moves, a game instance is created and evaluated by using evaluation functions. After all the possible moves have been generated and a heuristic corresponding to each one is calculated, the algorithm then picks the highest returned utility value and moves in that direction. At this instance, all the generated game instances are cleared from the memory in order not to cause a memory overflow. Following MAX's move, MIN gets to move in a totally random fashion and the whole cycle is repeated until the game ends.

Since the Minimax algorithm traverses the search tree in a DFS manner, a random depth limit of 4 was initially picked for recursion. This value was then changed at the testing phase to see if it would have any effects on the overall algorithm efficiency. The depth at each node is calculated by a function called pcount(*node). This function simply returns the number of parents of a given node, which is equivalent to the depth of the tree at that node.

Below is the pseudo-code for Minimax algorithm.

```
playgame_minimax()
{
        while(there is a node to move && pcount(node) < max_depth)
        {
        node = find an available node for player p.
          evaluate 8 directions (node);
        for (direction=0; direction <9; direction++)
                { copy current game;
                  make a legal move in copied game in direction if possibble;
                evaluate the heuristics at this instance;
                  put this game instance in a queue of games;
                direction ++;
               }
        node->c = true;
          playgame_minimax();
          }
          Analyze the whole game queue for the best heuristic;
          Move node in that direction;
}
```

As seen above the termination cases for the recursive function are when there are no more nodes to make a move which simply implies the end of a game, and when the depth of the current node is above the depth limit. At that point, the recursion is skipped and the children of that node are not evaluated.

Alpha Beta Pruning

It was seen that the problem with Minimax search is that the number of game states it has to examine is exponential in the number of moves. Although it is complete and optimal, computation wise it takes a long time. In Alpha Beta pruning, the algorithm returns the same moves as Minimax, but prunes away the branches that cannot possibly influence the final decision.

This is achieved by using two variables, namely alpha and beta.

alpha = the value of the highest value (i.e best) choice that the algorithm found so far at any choice point along the path for MAX

beta = the value of the lowest value choice the algorithm found so far at any choice point along the path for MIN

Both the Minimax search and Alpha-Beta pruning work on the assumption that the opponent (i.e MIN) is always making the best possible move.

The search algorithm updates values of alpha and beta as it goes along and prunes the remaining branches at a node by terminating the recursive

function call. So as soon as a node is found to have an alpha value less than the current alpha value for MAX, that node including all the neighboring nodes that are at the same depth and coming from the same parent are pruned. Below is the pseudo code for Alpha-Beta Pruning

```
playgame_alpha_beta(int alpha, int beta)
{
        while(there is a node to move && pcount(node) < max_depth)
        {
        node = find an available node for player p.
          evaluate alpha_new for 8 directions (node);
          for all legal directions{
          if alpha_new > alpha then
                {alpha = alpha_new;
                        Move in that direction
                         { copy current game;
                           make a legal move in copied game in direction if
possibble;
                    evaluate the heuristics at this instance;
                           put this game instance in a queue of games;
                   }
                  }
          Else prune;
        node->c = true;
          playgame_alpha_beta();
          }
          Analyze the whole game queue for the best heuristic;
          Move node in that direction;
}
```

Evaluation Functions

Squares Occupied

This heuristic calculates the difference between the number of squares occupied by MAX and those occupied by MIN. The idea is that occupying as many squares as possible is favorable as it will restrict the opponent to only a few squares, thus making it easier to trap him and win the game.

This heuristic leads to MAX filling up the majority of the board very quickly. This allows MAX to have more moves available and increases the chances of MIN running out of moves. While the idea is good, it was found that in practice this heuristic was not quite efficient. MAX can occupy a maximum of 10 out of 16 squares on the board which give MIN the remaining 6, plenty of room to move. Often the heuristic causes MAX to move away from a winning end condition, and not winning the game.

Adjacent Squares

This heuristic calculates the number of squares that one player has adjacent to the other. The idea is to maximize this number for MAX. A higher number of adjacent squares leads to the opponent getting chased, effectively, and is indicative that the opponent is about go get trapped and run out of moves.

While sound in principal, this heuristic did not produce any decent results in practice. Often, MAX will surround one or two squares of MIN while other squares of MIN are free to move in all directions. The heuristic was unable to produce end of game conditions to win the game even after hundreds of moves.

Number of Moves

This heuristic focuses on evaluating the number of moves that each player can make at any state in the game. MAX searches moves to minimize the number of moves that MIN can make, hopefully reaching a state where that number is 0 and MAX is the winner. The heuristic gives a nonlinear relationship, giving a much higher number of close to end and end game conditions to speed up the winning process.

This was found to be the most effective heuristic implemented. More and more game states were being pruned as a result of this algorithm which means that it took less time and moves to lead the game to an end state condition. The game was finished in 72 moves when using this heuristic.

Summary of Results

Below is a summary of the evaluation of the performance of each heuristic implemented in the game. The tables show results for two different search tree depths: 4 and 6.

Depth: 4

Heuristic	# Moves to win	Nodes Explored (x 1000)			% Pruned
		Searched	Pruned	Total	
Squares Occupied	152	9.1	3.4	12.5	27%
Adjacent Squares	446	7.2	2.1	9.8	21%
Number of moves	72	6.2	3.2	8.5	52%

Depth: 6

Heuristic	# Moves to win	Nodes Explored (x 1000)			% Pruned
		Searched	Pruned	Total	
Squares Occupied	173	101	71	172	41%
Adjacent Squares	642	82	40	122	33%
Number of moves	81	48	37	85	77%

Sample Output

The figures below show the strategy behind our most successful heuristic. The strategy is shown in the series of moves below where MAX (shown in black) plays to reduce the number of available moves for the random agent.

10			
			10

	1	2	7
			10

7	1	2	7
	2		
		1	

7		2	7
1	2		
		1	

7		2	7
1			
	1	1	
	1		

7			7
1	1		
1	1	1	
	1		

	1	6	7
1	1		
1	1	1	
	1		

1	1	6	7
	1		
1	1	1	
	1		

1	1	6	7
1	1		
1		1	
	1		

1	1	6	7
1	1	1	
1	2	1	
4	1		

Conclusions

As mentioned above, it was found that the Number of Moves heuristic gave the best results when implementing the adversarial search. This is when the lowest number of nodes were explored as a lot of nodes in the game state tree were pruned. The number of available moves for MIN (or the random agent) drops consistently until it becomes 0, putting an end to the game.

It was realized that the most pruning occurs when the heuristic function assigns values that are substantially different from one another. This is one of the reasons why this method worked so well.

The ‘Adjacent Squares’ and ‘Number of Squares’ start off at the beginning spreading MAX out on the board very quickly but are unable to put a quick end to the game by trapping the random agent. A significantly higher number of nodes are explored which makes the process take longer. The reason for this is that these heuristics assign similar utility values so not a lot of nodes are pruned.

It was found that the difference in tree depth used did not cause a major change in the results. Of course, more nodes are explored when the tree depth increases. A higher rate of nodes were also pruned when this happened, which makes sense because alpha-beta pruning works better in larger trees. This increases the chances of MAX winning in fewer moves. The downside for this, however, is time taken to search a few thousand more nodes.