Hierarchical Heuristic Search Techniques for Empire-Based Games

Kenrick Mock
University of Alaska Anchorage
Dept. of Mathematical Sciences, Computer Science
3211 Providence Dr. Anchorage, AK 99508
kenrick@uaa.alaska.edu

Abstract

Computer games have long been a fruitful and challenging area for the application of AI technologies. The empire-based strategy game is one genre that presents unique challenges to the implementation of an AI-based player due to the extremely large search space. Consequently, a majority of AI systems utilize ad hoc strategies such as hand-built finite state automata. While practical, this approach limits the computer player to the strategies designed by the programmer. A more flexible approach allows the computer to adaptively search for promising moves. This paper proposes a hierarchical architecture that breaks the problem space into manageable units. Through aggressive pruning strategies heuristic search may be employed across the hierarchical levels. The new search space examines approximately (nm^{d+1}) states for m moves, n pieces, and a search depth of d. In comparison, full search requires the examination of approximately $(m^n)^d$ states. This technique was implemented on a simple empire-style game that generated expected results

1. Introduction

The public has often viewed the ability of a computer to play strategy games as a measuring stick for the progress of artificial intelligence. For example, great attention has been placed on the ability of computer programs to beat the top humans in chess, checkers, or go. While this view is not representative of AI as a field, computer games do offer an excellent venue to test and apply various AI techniques. These techniques range from machine learning to navigation, natural language processing, or character behavior modeling [1]. Recently, with competitions such as RoboCup and extensible software interfaces for commercial programs, there has been more interest from the academic community to pursue computer games as an application area for research [2].

One genre of computer games is the empire-based strategy game. The empire strategy game typically involves conquest of the world through the control of combat units and economic resources. The world is often a hexagonal or grid-based map filled with combat units, resources, land, water, etc. Players take turns moving their pieces or may play in near real-time. Examples of such games include Age of Empires, Command and Conquer, Warcraft, or Civilization. I denote these as empire-based games named after the freely available UNIX game "empire," which was one of the first such games in this genre.

While the format and rules will vary from game to game, all empire-based games allow a player to move more than one combat unit per turn, or move no unit at all. This is notably different from a "classical" board game like chess or checkers, where a player may only move a single unit per turn. The ability to move multiple units per turn creates an extremely large problem space that makes traditional search techniques like minimax difficult to apply.

For example, consider a 2-player scenario where each player has *n* units and each unit always has *m* moves available. Since each player may move any or all of his units during their move, we must consider the combination

of moving all n pieces as a single move for minimax. This results in a total of m^n combinations or m^n different moves. The branching factor b for a minimax game tree is the number of available moves from each state in the game tree. If $b = m^n$ and we wish to search ahead to a depth of d (starting at d=0) then we must evaluate a total of $(m^n)^d$ states. Clearly this is a prohibitively large search space even for small values of n and m.

Given this large search space, most commercial games use hard-coded finite state machines (FSM) to implement the artificial intelligence of a computer opponent [5, 8]. For example, the FSM may model the internal state of its units, and if a unit has low strength that unit might be instructed to retreat. While a large FSM may be complex, this technique does allow the computer opponent to play as intelligently as the programmer can encode directly. However, the obvious drawback is that this technique is static. Once a human opponent learns the strategies encoded in the FSM, the human can easily devise a strategy to thwart the computer and the computer will never be capable of countering the human's strategy due to the programming limitations. To address this problem, many game programs "cheat" to compensate for their lack of adaptive strategic skills [9].

Other AI techniques used in games include genetic algorithms, neural networks and autonomous agents [9]. As noted by Woodcock, the neural network and genetic algorithm approaches have been difficult to implement successfully. Autonomous agents are typically implemented via FSM and it becomes difficult to centrally coordinate agents without additional communication mechanisms.

2. Proposed Methodology - Hierarchical, Heuristic Search

Using the FSM method, a computer program will never be better than its creator. Obviously this is not the case for many games, such as chess, where computer programs can play at a level far superior to that of their programmer.

The solution is to rely upon the brute force power of the computer to perform heuristic search to find good moves. In this project I propose a similar approach for empire-based games. By using heuristic search combined with lookahead, a computer player will be able to make the appropriate move that maximizes the heuristic instead of relying on pre-encoded strategies.

The difficulty with heuristic search in empire games lies in the complexity of the search space. To address this problem, techniques must be established that prune large sections of the search tree [4]. These techniques are likely to be domain-specific. In this work we propose some fairly general techniques that are likely to apply to most empire-based games.

2.1 Hierarchical AI

The first technique is to split the problem space up into manageable chunks via a hierarchical organization much like the chain of command in modern military forces [3]. As described by Luppnow, in actual military forces a general does not concern himself with the movements of individual soldiers. It does not make sense for a computer program to do the same. For example, we might control the AI with a general at the top of the hierarchy, several commanders in the middle of the hierarchy, and a large number of soldiers at the bottom of the hierarchy as shown in figure 1.

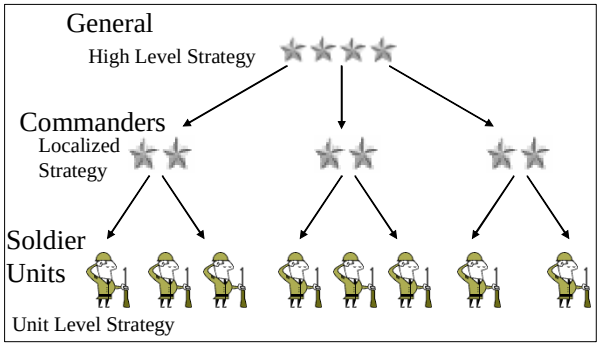

Figure 1. Hierarchical Problem Space

The general operates at a very high strategic level. For example, moves might be as

abstract as “defend cities”, “all-out offensive”, “re-supply front line”, “control sector A”, or “control bridges”. The actual set of strategic moves will be specific to a particular game and also challenging to determine and implement. Equally challenging are heuristics that can be applied at the strategic level. For example, heuristics might favor the control of resources or the control of enemy territory. The heuristic might be adjusted during the course of the game by rules encoded by the programmer.

By encoding heuristics and moves at a high, generalized level, we can now perform minimax search at the level of the generals. That is, each general makes abstract strategic moves to generate a search tree, and we select the move that leads to a state that maximizes our heuristic. This type of search is feasible since the problem space of the generals is of much lower complexity than the problem space of all individual units combined. Note that a single “move” at the level of the general likely encompasses many moves for a single soldier, so in effect we are performing a deep lookahead for the individual soldiers.

After the general has performed its minimax routine and selected a move, the results of the move are passed down the hierarchy to the commanders in the form of orders. Each commander performs a similar job to the general, but only within the scope of each commander’s domain. For example, a commander might be in charge of a small region of the map and will only concern itself with units within that region. The order passed down from the general specifies what the heuristic function should be for each commander.

For example, if the general has selected “all-out-offensive” then the heuristic for the commander will maximize those states that attack enemy units. If the general has selected “defend cities” then the heuristic will maximize those states that place units inside or near to cities. Using minimax, the commander will then search through its available moves to find the move that maximizes the heuristic. Moves at the level of the commander might be as specific as “Move unit A to coordinate 1,3” or “Attack enemy unit Z with unit A.”

Finally, the orders from the commanders propagate down to the individual soldier units. These units may either execute the order directly or perform a unit-level minimax search. For example, if a unit is instructed to engage another unit in combat and has multiple options available, the soldier may perform a shallow search through the available options to find the most effective one.

2.2 Minimax Pruning Strategies

While the strategy outlined above will help break the problem space up into smaller and more manageable chunks, we still must deal with the combinatorial explosion of searching through multiple units that may be moved simultaneously. For example, consider a commander with a modest three units under its control where each unit has 9 moves available. If the enemy also has three units in the commander’s region, then by the analysis from section one we will require $(9^3)^d$ or 729^d states to examine, where *d* is the search depth. With a branching factor as large as 729 we will need to create and examine $2.8 * 10^{11}$ states for a modest lookahead of four moves!

On one hand, a deep search depth may not be necessary at the level of the commanders or below since a deep search is performed implicitly at higher levels in the command hierarchy. Nevertheless, an extremely shallow search of 2 to 4 moves may be insufficient lookahead to generate good moves. To address this issue we must implement further pruning of the problem space. One technique is to:

- Select a unit controlled by the commander
- Apply minimax to determine a move for that piece and apply the move
- Repeat for the next unit controlled by the commander

After all pieces have moved then we switch turns to the opponent. For real-time games, we will need to either complete the entire turn within a time limit or have the capability of applying the search across multiple, short turns.

To apply minimax we must now reduce the complexity of the problem space. The technique proposed here is to perform a full search of all possible moves only for selected units on the map (e.g., the selected piece and selected critical opponent(s)). We must not ignore all other pieces, however, as the moves they make can certainly influence the moves our selected piece should make.

However, instead of performing a full search for the other pieces, we instead perform a search to a depth of 1 moving that piece only (i.e. pick the single move for the piece that best maximizes the heuristic for us, or best minimizes the heuristic for the opponent). An alternate technique is to remember the prior move selected when this piece searched ahead, and apply it during this phase.

Figure 2 shows a sample problem of 3 units with 2 moves each. Each move is either "X" or "Y" and unit 3 is the selected piece to expand the tree for both min and max:

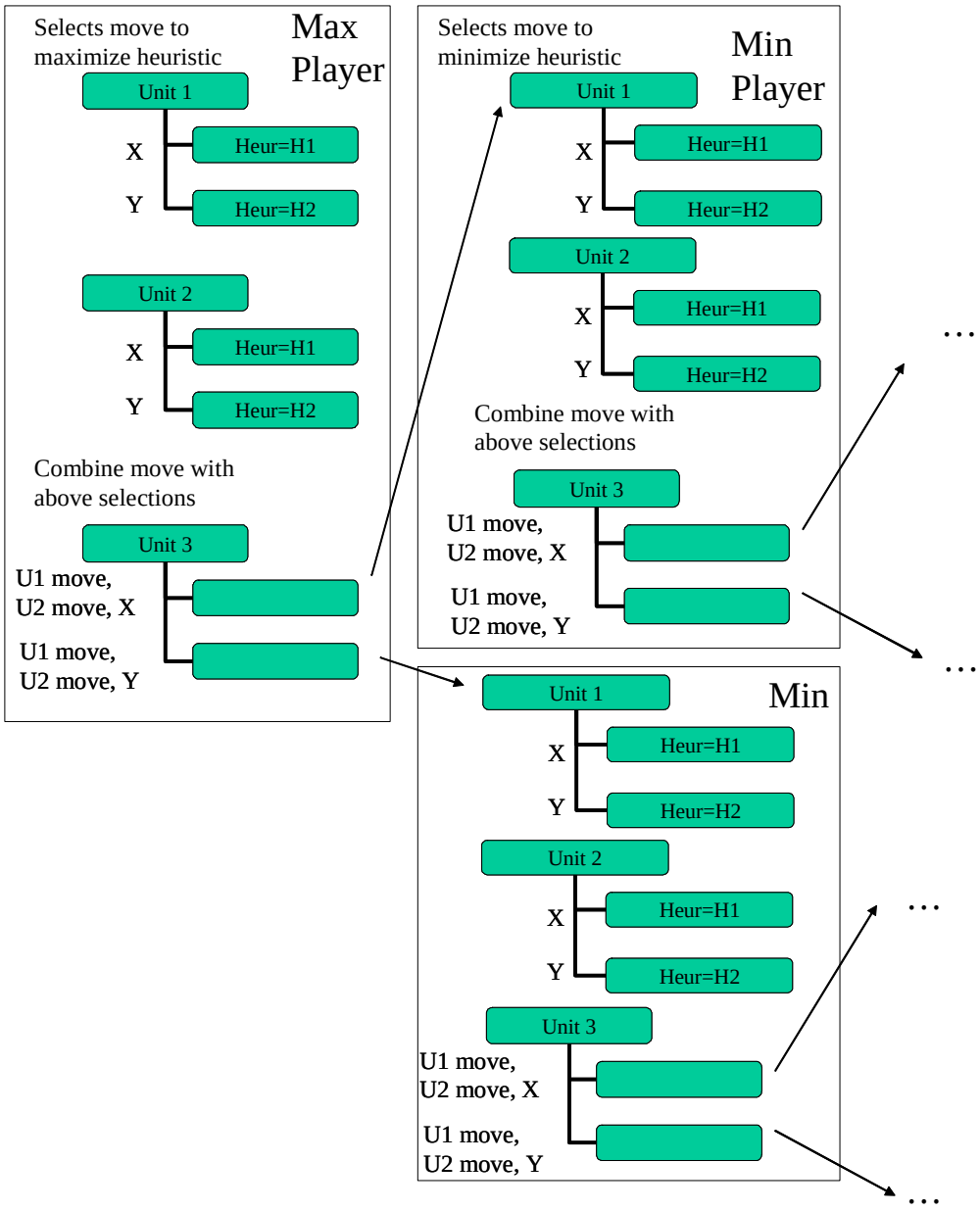

Figure 2: Pruning minimax states

In the general case, given n pieces and m moves per piece, if we limit ourselves to selecting a single piece for which we generate all moves for both the maximizing and minimizing player, then at each node in the tree we examine mn states as we apply the heuristic to each individual piece. The branching factor, however, has been reduced to m instead of m^n. If we now search to a depth of d, we will examine a total of $\left(mn(m^d - 1)\right)$ internal states plus m^d leaf states or approximately $\left(nm^{d+1}\right)$ states. While still compute intensive, this is much smaller than the space of $(m^n)^d$ states required for the full search. Given our hypothetical example where $n=3$, $m=9$, and $d=4$ we originally needed to examine $2.8 * 10^{11}$ states. Using the pruning algorithm we instead examine approximately $2.4 * 10^5$ states.

Unfortunately, we are not quite finished yet – the above analysis only applies to the evaluation to select a move for a single piece. Based on the algorithm we must apply this routine for all pieces, resulting in an additional factor of n for an overall complexity of $O(n^2m^{d+1})$.

While certainly compute-intensive, this complexity is feasible for small values of n and m. Additionally, up to half of the states may be pruned using alpha-beta pruning [6].

3. Experiment – Empire Lite

To gauge the effectiveness of the pruning algorithm, a simple empire-style game, Empire Lite, was created. Empire Lite simulates the work that may be done at the level of an individual commander using the pruning algorithm. Implementation of the hierarchical architecture is left for future work.

Empire Lite is played by two players on a 10x10 rectangular grid. Units may move one square left, right, up, down, diagonally, or nowhere for a total of 9 possible moves. Diagonal moves allow movement over a farther distance by $\sqrt{2}$, so in the future a hexagonal board is preferable. If two units engage in combat with equal strength, both lose one point value. If one unit has a higher strength than the other, the unit with higher strength loses one point value and the lower strength unit loses two point values. Units are removed when their strength drops to zero or below.

The only economic resource in Empire Lite is cities. Initially unoccupied, once a unit moves into a city it becomes owned by that player. All of the unit's pieces then increase in strength by a value of one. If an occupied city is captured, the original owner loses a strength point for all units and the captor gains a point for all units.

The heuristic employed in this experiment was a weighted polynomial that favored occupying cities, having stronger pieces, being close to empty cities, and then being close to enemy units:

H = 20(Num_A_Cities – Num_B_Cities) +
10(Total_A_Strength – Total_B_Strength) +
5($\sum_{all\_units}$(10-Closest_Dist_To_City))+
1($\sum_{all\_units}$(10-Closest_Dist_To_Enemy_Unit))

3.1 Experimental Results

Using a 1.6 GHz Pentium IV processor on a game with three pieces per player, the computer was able to search 6 moves ahead in about 4 seconds for all three pieces. The end result was a computer player capable of reasonable play, sometimes achieving a stalemate. A few examples of play that benefited from the minimax search are shown in figure 3.

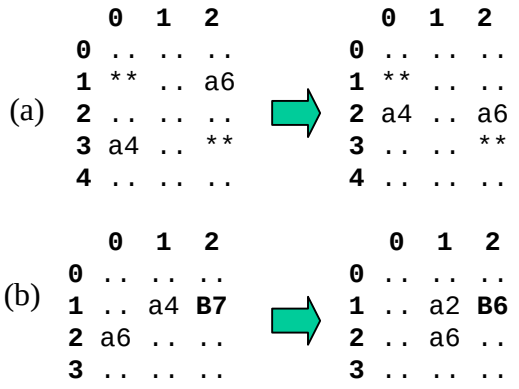

Figure 3 : Example Game Playing Scenarios

In figure 3a, the **'s indicate empty cities. The two pieces belonging to player A have strength of 4 and 6 respectively. If these pieces were operating independently, they would both head toward the city at (0,1) to occupy it. However, using minimax search, piece a6 recognizes that a4 will occupy the city and instead heads toward the city at (2,3).

In figure 3b, the bold B7 indicates a city currently occupied by enemy piece B with strength of 7. If pieces a4 and a6 were operating independently, they would not attack B7 since it is stronger than each individually. However, through minimax lookahead, piece a4 attacks B7 recognizing that it may sacrifice itself allowing a6 to later overwhelm the city.

4. Future Work

Although the examples shown above are not particularly brilliant moves, they do illustrate intelligent play without the use of hard-coded rules. Hopefully this work illustrates the potential for heuristic search in empire-based games that may one day lead to computer opponents that can rival the best human players.

Much additional work needs to be done to accomplish this feat. As described previously, it is difficult to describe and implement the search space at the level of the general. For example, what is available to move? What is a unit at this level? A combination of heuristic search and rule-based methods may be most practical. Another problem is the issue of how to break the problem space up into meaningful chunks dynamically. For example, a commander should have control of applicable units, not necessarily all units within a fixed radius. Techniques such as influence mapping may be helpful in determining strategic dispositions [7]. Finally, the runtime of the pruning algorithm may still be too large for mainstream programs, and require further pruning mechanisms.

5. References

[1] Funge, John. (1999). AI for Computer Games and Animation: A Cognitive Modeling Approach. A K Peters Ltd.

[2] Laird, J. and Van Lent, M. (2001). Human-Level AI's Killer Application: Interactive Computer Games. Communications of the ACM, 22(2), 2001.

[3] Luppnow, Andrew. (December, 1994). Hierarchical AI. Retrieved 2/20/2002 from http://www.gamedev.net/reference/articles/article199.asp

[4] Nahr, Christoph. (1999). Computer Players in the Star Chess Game. Technical Report, 17th August 1999.

[5] Patel, Amit (Ed.). (July, 1993). AI In Empire-Based Games . Retrieved 2/20/2002 from http://www.gamedev.net/reference/articles/article196.asp

[6] Pearl, Judea. (September, 1980). Asymptotic properties of minimax trees and game-searching procedures, AI Journal 14(2), pp.113-138.

[7] Woodcock, Steven (Ed.). (July, 1995). Recognizing Strategic Dispositions thread. Retrieved 2/20/2002 from http://www.gamedev.net/reference/articles/article1085.asp

[8] Woodcock, Steven. Game AI: The State of the Industry. Game Developer Magazine, August 2001.

[9] Woodcock, Steven. (n.d.) Games Making Interesting Use of Artificial Intelligence Techniques. Retrieved 2/20/2002 from http://www.gameai.com/games.html