CS405

Game Playing

The search routines we have covered so far are excellent methods to use for single player games (such as the 8 puzzle). We must modify our methods for two or more player games.

Ideally: Use a search procedure to find a solution by generating moves through the problem space until a goal state is reached (i.e., we win). This is not realistic; the branching factor and depth of most games is too high. In many games, b is about 35, and number of moves (or ply) is about 100. 35^{100} too large to search!

Idea: Use a heuristic evaluation function to evaluate the board state and estimate the distance to a win. Look as many moves ahead as we can within allotted time to get a better estimate. For example, if we are playing checkers, a simple heuristic might be:

# of my pieces / # of his pieces

The larger the ratio, the better. If the ratio approaches 0, then we're in trouble. The idea is to choose moves that maximize this ratio.

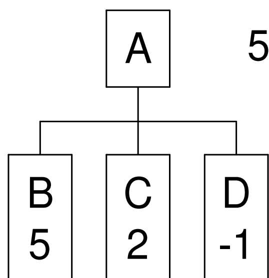
We will discuss example heuristics in more detail later.

Search procedure: what to use? Can't use A* very effectively since it doesn't take into account the adversarial nature of the game. Must use a modified search procedure. The most often used procedure is called Minimax.

Minimax Search

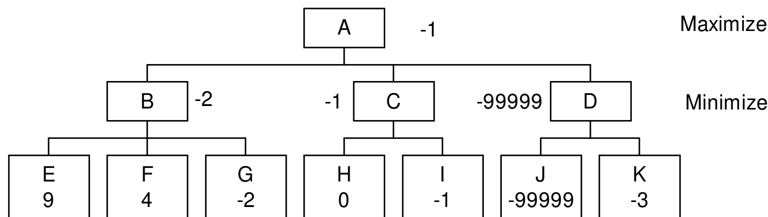
Idea is simple: look ahead from current game state as many moves as possible in a depth-first manner. Apply the heuristic function to these positions and choose the best one.

Example of 1-ply search:



Say we are playing a game like checkers and we are at board state 'A'. We can make three moves to either state B, C, or D. Applying the heuristic to each of these states results in the values shown. Large values are good, negative values are bad. Looks like choice "B" is the best, so node A would get the value of 5 as the best and select B as the move.

What if we want to look ahead another move? We have to take into account that the next move will be our opponents move. Instead of picking a state with a high value, we will assume our opponent will use the same heuristic function as us and pick the state leading to the smallest value. This is called a *minimizing* move; when it is our move, this is called a *maximizing* move.



In the example above, if we search one more ply and then apply the heuristic function, we get different results. If we assume that the opponent will choose to pick the move resulting in the smallest heuristic value, then we have to propagate back the minimum of the children at a min node. State B turns out to be not so good due to G, while C is slightly better. Move D turns out to lead to a lost game for us at state J.

Note: Minimax assumes an opponent as smart as we are. Sometimes we may want to make a move like B, and hope that the opponent won't choose G, but will instead choose E or F. Since G is close to I, but there are better other moves at B, we could take move B and hope the opponent will pick one of the other moves.

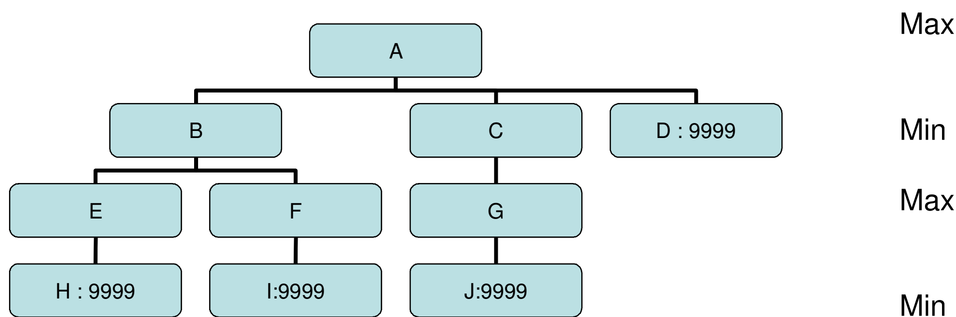
Algorithm:

```
Function Minimax-Decision() returns Move
        Move_List = MoveGenerator(game_state)
        For each move M in Move_List do
                Value[M] ← Minimax_Value(Apply_Move(M,game_state))
        Return M with the highest Value[M]

Function Minimax-Value(state) returns a heuristic Value
        If current_search_depth == Desired_Depth or OutOfTime or Terminal(state) then
                Return Heuristic(state)
        Else
                Move_List = MoveGenerator(state)
                For each move M in Move_List do
                        Value[M] ← Minimax_Value(Apply_Move(M, state))
                if Whose_Turn==MyTurn then
                        return Max of Value[]
                else
                        return Min of Value[]
```

This algorithm assumes that the heuristic function doesn't flip for the opponent. Some algorithms assume that it does. For example, in this algorithm, a state that is bad for us would have a small heuristic value whether or not we are at a min or max node. In other algorithms, this state would be low at a max node, but it would be high at a min node since its being evaluated from the point of view of the opponent.

Note: May want to explicitly check for WIN state; if you can win, make the move. Consider the following:



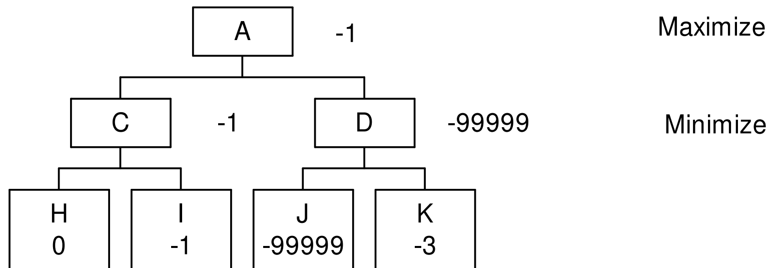
If the maximizing player takes the move to B, then looking ahead we see we are guaranteed a win. But we could directly win if we make move D. The downside is we may take the move to B instead, which could potentially result in an infinite loop if this type of move is repeatable. A similar problem occurs if we don't check and exit for win states (the maximizing player may then go for multiple-win states if that increases our heuristic value)

Complexity: Essentially just doing a depth-first search

Ways to search deeper in the same time: Alpha-Beta Cutoffs

A strategy called alpha-beta pruning can significantly reduce search time on minimax, allowing your program to search deeper in the same amount of time. In general, this can allow your program to search up to twice as deep compared to standard minimax. The modified strategy also returns the exact same value that standard minimax would return.

Consider the tree below:

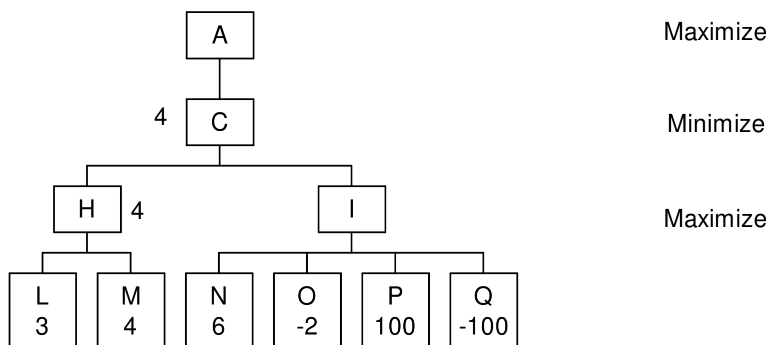


The minimax algorithm will search the tree in a DFS manner: A to C to H, to I, A to D to J, then A to D to K. Notice however, when what happens when we're at node D and have just examined J. We know that A wants to maximize its move. It can already get a value of -1 by making move C. D wants to minimize; it can choose at LEAST -99999, and maybe less. So we don't even have to examine node K, because there is no way A will pick branch D, since D can choose a value < -1.

This is called an alpha prune; on a min node, we came across a value LESS than the current max. At this point we can stop searching because we won't want to make a move where the opponent can force a worse board state.

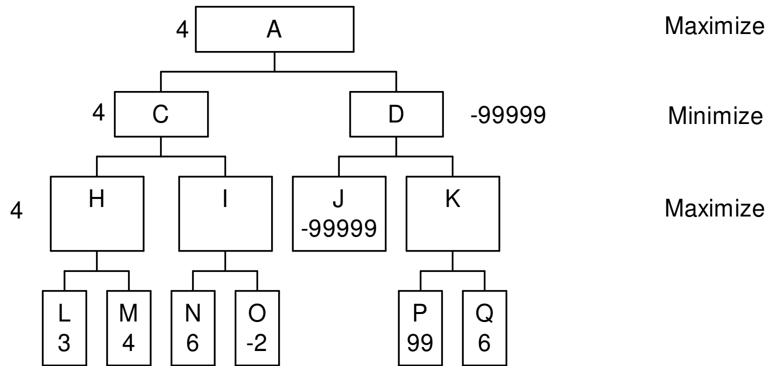
We can also do the opposite prune, a beta prune; on a max node, if we come across a value GREATER than the current min, then we can stop searching because our opponent won't want to make a move where we can get a better board state.

Here is an example of a beta prune:

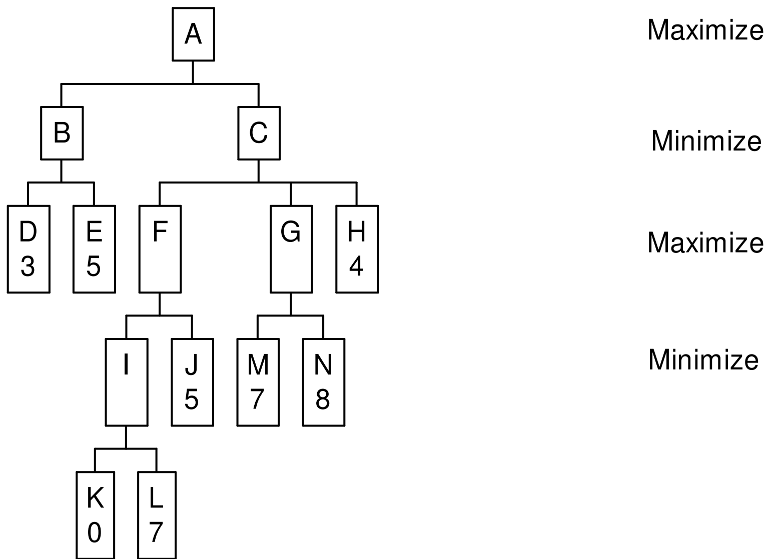


In this case, after examining node N we don't have to examine nodes O,P, or Q at all, since we can choose a value of at least 6. However, the opponent can limit us to 4 by choosing move H. So moving to I would be bad and any other values can be discarded.

Here is another example with both an alpha and a beta prune:



One more example:



At min nodes, compare to alpha (max so far). At max nodes, compare to beta (min). Can prune out nodes L and N. Why? What is the final backed-up value? Note that in these examples only a few nodes were pruned, but if the pruned nodes are entire trees, then a significant amount of pruning can be achieved, enough to search several layers deeper in search.

Algorithm: Call with Max-Value(state, -MaxValue, MaxValue)

```
Function Max-Value(state, alpha_max, beta_min) returns pair: minimax value of state, move
    If current_search_depth == Desired_Depth or OutOfTime or Terminal(state) then
        Return Heuristic(state), any move
    Move_List = MoveGenerator(state)
    BestMove ← Move_List[0]
    For each move M in Move_List do
        Value ← Min-Value(Apply_Move(M,game_state), alpha_max, beta_min)
        If Value > Alpha_max then
            Alpha_max ← Value
            BestMove ← M
        If Alpha_max >= Beta_min then return Alpha_Max, BestMove
    Return Alpha_Max, BestMove

Function Min-Value(state, alpha_max, beta_min) returns minimax value of state
    If current_search_depth == Desired_Depth or OutOfTime or Terminal(state) then
        Return Heuristic(state)
    Move_List = MoveGenerator(state)
    For each move M in Move_List do
        Value ← Max-Value(Apply_Move(M,game_state), alpha_max, beta_min)
        Beta_min ← Min(Value, Beta_min)
        If Alpha_max >= Beta_min then return Beta_Min
    Return Beta_Min
```

In this algorithm, we only care about the best move from the initial invocation to Max-Value. The rest of the algorithm only requires the heuristic values. Consequently, the BestMove code is only present in the Max-Value routine.

To-do in class: revisit previous tree example and show pruning using the algorithm.

Some strategies to increase speed:

1. Search the potentially best moves first. If you can find good moves early on, these moves help prune more branches.
2. Prune search tree by eliminating some moves that are obviously bad to make.
3. Copying board may be time consuming; consider incremental approach, undoing moves on the same board. If you are copying a board, consider using memcpy instead of a loop copying each element.
4. Use opening library of book moves, if you have analyzed and found opening moves. In book moves, you store an exact board state and the move you have determined to be best for that state. If the state arises, make that move. Use a hash or lookup table to retrieve the moves quickly.
5. Perform search while opponent is moving (predict what move they will make, do squashing). This is not allowed in the tournament, but if you can get it to work I will give you extra points for the project!
6. Optimize your code for speed wherever possible ☺

Other Hints:

Sometimes heuristics are tuned to only when it is your move, not your opponents; in this case you may wish to force the search to only search an even number of levels.

You may want to search an entire level one at a time to avoid a “horizon” effect. For example, if your search runs out of time before it has examined all the possible moves to depth 1, then you could possibly miss a win state. Also, depending on how the heuristic is implemented, the value may change dramatically from one depth to another. This means you can’t really compare a heuristic value from one depth with the heuristic value from another. Consequently, you should explore all moves to a certain depth and pick the best move from the completed depth over the best from an incomplete search. In other words, consider an “iterative deepening minimax”. Search depth 1 and save the best move, search depth 2 and save the best move, etc. until time runs out and then pick the best move from the completed depth. Not only does this avoid the problem of an incomplete search, it also avoids the problem of not making a winning move if all moves lead to winning states if your program stops after depth 1 if there is a win.

Alternatives to Minimax?

Some alternatives so far have been:

1. Rule-based systems. These essentially do no search or lookahead but rely on a large number of rules to make the move. This is analogous to a complex heuristic function and doing only one move lookahead.
2. Machine learning systems such as neural networks. The machine plays many games, adjusting its weights and parameters of what makes a move good; Tesauro has implemented this technique into a world class backgammon program.

Machine Learning with Games

Let's briefly examine an early technique to apply machine learning with games. We'll look at Samuel's Checkers program that he created in 1959. Samuel was a better-than-average checkers player who was able to write a program that learned to consistently beat its author.

Samuel's basic program operated upon minimax with a linear, weighted heuristic. He was able to search ahead to a ply of 20 before exhausting memory on his IBM 704. (How could he look this far ahead? Consider the branching factor of checkers; it is fairly small).

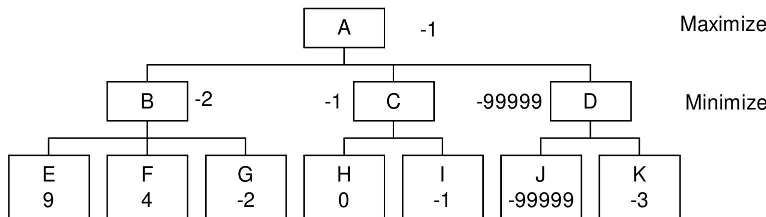
His heuristic was a weighted polynomial of the form $Ax + By + Cz + \ldots$
Where x, y, and z are variables calculated from the game state, and A, B, and C are constants. He had up to 35 variables in his experiments. The most important he found

were variables that characterized the 1) piece advantage, 2) denial of occupancy, 3) mobility, and 4) a hybrid term combining control of the center and piece advancement.

Rote Learning

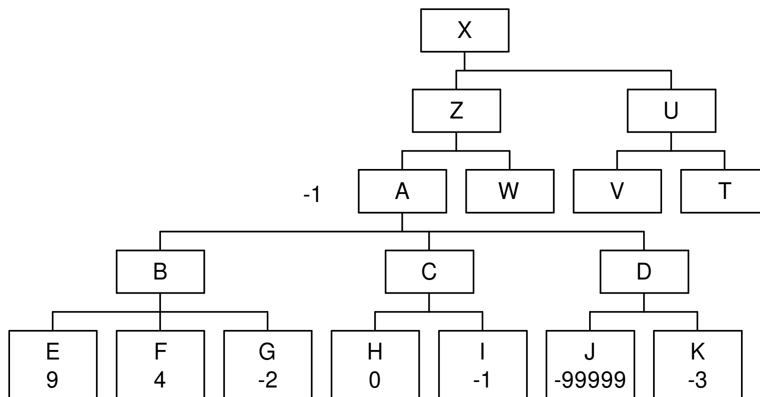
Rote learning is the most elementary form of learning. To implement rote learning, the program would simply save all of the board positions encountered in play along with the computed lookahead score. Reference could then be made to this record and then a certain amount of computing time might be saved.

For example:



We had to look two ply to compute a value for node A. Samuel's program would store the representation for node A along with the value of -1, and if this state reappears then we can just immediately return -1 without having to do any search.

This can let us look ahead farther than we normally might. Let's say that our program normally looks ahead only 2 ply. If we come across a case like the following:



Normally, search would stop at A, W, V, and T. However, since we remembered node A with its heuristic value searching two moves ahead, we get a more accurate heuristic value for A which could help us play better. When we save the heuristic value for X, and if X repeats, then we can incorporate moves looking even further ahead (by using A's value).

One problem that Samuel encountered was that of progress in the end game. For example, consider two Kings vs. one King. This is a winning combination for the two

Kings in almost all combinations. In time, the program can be assumed to store all variations, each associated with a winning move. Now it looks like the program can take any move it wants in order to win. However such a move might be wasting time, not taking a direct path. The problem might turn into loops or the pieces meandering back and forth since every move appears to result in a win. Samuel's solution to this problem was that if there was a tie among the heuristic value, select the move with the smallest backed-up ply; i.e. the move resulting most directly to the desired state.

Using these technique, Samuel was able to take a poor program and turn it into a better-than-novice program. It played very well in openings, not so well in the middle game, and average in the end game.

Generalized Learning

To perform more generalized learning, Samuel attempted to have the program alter its heuristic constants after every move. Samuel had the program play itself as Alpha and Beta. The Alpha player would alter his constants, while the Beta player used a fixed heuristic function.

The idea is for Alpha to make a move based upon the backed-up heuristic. Then, Alpha has to somehow figure out if that move was good or not. This can be problematic since the only measurement we have of goodness is the same heuristic we want to alter!

The solution is to rely upon lookahead to fix the heuristic. Alpha stored the heuristic value for the last move. Then for the current move, Alpha computes the heuristic using lookahead and compares the result to the previous value. Note that the new value is now looking ahead farther than before, so its heuristic should be more accurate. Delta is set to the difference between these scores. If negative, the heuristic is evaluated and those values that made the value positive are given less weight and those that made the value negative are given more weight. The converse is true if delta is positive; the constants leading to a large heuristic are reinforced and those that decreased the value are made smaller. Overall this is a very difficult problem – where to assign credit.

After just 10 hours of playing, the program was capable of playing well. The middle and end games played very well, while the opening game was poorer.

The best program resulted by combining both Generalized and Rote learning.

You are not required to implement any learning of this nature for your project, but feel free if you have time. Much twiddling is required to get everything to work!

Another form of learning that has been implemented for games and problems in general is the neural network and statistical approach. We'll discuss this in a later lecture.

State of the Art with Game-Playing Programs

Solved Games:

Connect-4, Go-Moku (5 stones in a row), 3D tic-tac-toe (on a 4x4x4 matrix) have all been solved by computers. They are capable of looking ahead all the way to the end, so they are won by the first player with correct moves.

Backgammon:

See above. Applying AI to games involving chance is a recent research area that is only beginning to be examined in more detail. We need to modify our algorithms so that they can properly search under uncertainty. The random roll of the dice creates so many possibilities at each move (branching factor up to 400) that only a shallow look-ahead is possible.

One of the first competitive computer programs to play backgammon was G. Tesauro's TD-gammon. This program uses temporal-difference learning with a neural network. In backgammon, the dice introduces a branching factor of 400. This makes traditional brute-force search intractable. Instead, TD-Gammon uses a neural network that was trained offline by having two different copies of the program to compete with one another, with some help by human experts. It only searches ahead 3 ply. In a 1998 AAAI contest, TD-Gammon lost to world champion Malcolm Davis by 8 points in over 100 contests. Other programs such as Snowy and Jellyfish are playing at championship levels.

Scrabble

The 1998 AAAI contest also featured a program called Maven that plays scrabble. Other programs that competed in the tournament include Maven, a program that plays scrabble. Most Scrabble programs rely on a massive dictionary to consider each playable word in each position on the board. Unfortunately for computers, the Scrabble community does not normally allow computer programs to compete in their tournaments.

Checkers:

Samuel applied machine learning to checkers in conjunction with minimax search. He had the machine play itself to learn a heuristic function and the best weights to make the heuristic better. His checkers program was ranked world class; currently, the Chinook system has claimed the world title, beating human Marion Tinsley in 1994. Chinook, developed at the University of Alberta, has an opening book of some 80,000 position and a closing book of some 443 billion position, comprising over 10 gigabytes of moves. Every position in which no more than eight checkers remaining on the board were entered into the closing book!

Go:

Go has a branching factor of about 250! Regular search methods fail. To give a concrete example, if we assume a branching factor of 35 in chess, then four moves in chess evaluates to 35^4 or 1.5 million states. In Go, four moves would evaluate to 200^4 or 1.6 trillion states.

Systems today use a knowledge base to suggest plausible rules to narrow search, but these systems still perform much poorer than world class humans. Amateur Go players are ranked by kyu, from 30 to 1. More experienced players are ranked 1-6 dan, and professional players are ranked from 1-9 dan. The Ing cup was established in 1986 with a cash prize of $1.8 million to the first program to defeat Taiwan's three best 14-16 year old players before the year 2000. This goal seemed reachable; after all, hundreds of people achieve this level of ability, which is about 3 dan professionally. However, up to 2000 the best Go-playing programs ranked only around 4 kyu and nobody claimed the prize (the Taiwainese businessman has since died). Programs today have improved. On August 7, 2008, the computer program MoGo running on 25 nodes (800 cores) of the Huygens cluster in Amsterdam beat professional Go player Myungwan Kim (8p) in a handicap game on the 19x19 board. Kim estimated the playing strength of this machine as being in the range of 2-3 amateur dan. (Wikipedia). There is (was?) a 21st Century Championship Cup where the winning computer program can win $5,000.

Othello:

Othello has a small branch factor; consequently computers are much better than the best humans. Logistello is the reigning computer champion, spanking world Othello champion Takeshi Murakami in 1997 by a score of 6 games to 0. It uses a weighted heuristic function for positions on the board. Due to the relatively small branching factor, strong Othello programs can now look 25 or more moves ahead. This means that, when the game is not quite two-thirds over, a program can see to the very end and thus play perfectly.

Chess:

A longstanding area of research. Deep Blue is the current state of the art, a massive effort by IBM involving custom built parallel processors that can evaluate billions of positions per second and search 13-15 moves ahead. It has beaten the world champion, Kasparov, although some debate the outcome; computers are already world champion at speed chess. Both Deep Blue and Chinook are examples of brute force search programs on a massive, heretofore unprecedented scale. Each explored enormous subtrees and supported tremendous opening and closing books that were carefully tuned by humans. Contrast this approach with the way humans play these games. In chess, humans consider only a handful of good moves and rarely look ahead more than 8 or 9 ply. Go experts

only look ahead 3 or 4 ply, and are remarkably able to zero in on reasonably good moves in less than a second. Some type of pattern recognition and experience with selecting candidate moves likely plays a role, and researchers may be able to use these techniques to improve their programs.

Timer Implementation

If you need any sample code to implement a timer in either C, C++, or Java then let me know. The easiest technique is to use a global variable that is set when time is up. Your code would have to check this variable to see if it should continue execution.

Hints on Designing Heuristic Functions for Board Games

Ideally you want a heuristic to give an accurate measure of how far away you are from a win. This can be very difficult for some games, especially ones where the board state may change radically in a single move. For example, in the game Quixo, it is possible in a single move to go from being close to a win, to being close to a loss. (explain how to play Quixo).

Admissible heuristics, which were so important in single player games, are often not so useful in multi-player games. Consider an admissible heuristic for Quixo. An excellent admissible heuristic would be the number of blocks left we need to get 5 in a row. However, this will often be useless, since it is very common to get 3 or 4 blocks in a row. Consequently, we will almost always get back "1" or "2", not very useful information to make a move.

Here are some simple sample heuristics used for various games in previous AI classes:

Othello:

A surprisingly good heuristic is just the ratio of pieces: Mine/His. Since the goal is to win by maximizing your pieces until the board is full, this works very well. Strategic locations are given a higher weight. For example, the corners are very strategic since they can never be captured again. The sides are slightly less strategic, and pieces in the middle even less strategic. A better scheme is a weighted heuristic based on board placement:

Value = A*#corner_pieces + B*#edge_pieces + C*#other_pieces
Heuristic=Value(Mine)/Value(His)

In general, this approach is very common. A weighted heuristic based upon different features of varying importance:

Heuristic = Const1*Feature1 + Const2*Feature2 + …

Quoridor:

Quoridor is played on a 10x10 board with a pawn starting at each end. Each opponent has a number of fences that can be placed (each occupying two squares) to block the pieces. Each player may either play a fence or move the pawn one square. The first player to reach the other side of the board wins.

My heuristic was to compute the shortest path for each player to the end of the board. This is actually a somewhat expensive heuristic; it requires finding the shortest path. I used A* to find this path, using a subheuristic of the current Y coordinate. The final heuristic was:
Heuristic = C1(C2*MyShortestPath – C3*OpponentShortestPath) + C4(C5*MyFences-C6*OpponentFences)

Pente:

The goal of Pente is to get 5 in a row on a 19x19 grid. If two stones are sandwiched, then they are removed. If 5 pairs are captured, then you win. Our heuristic was a weighted heuristic:

Value = A*(HisCaptured-MyCaptured) + B*(1_to_win) - C*(1_to_lose)
+ D*(2_to_win) - E*(2_to_lose) + F*(3_to_win) - G*(3_to_lose)
+ H*(4_to_win)

We had different weights for moves away from winning and moves away from losing. We made the moves-away-from losing weight to be high, so that the program would be more defensive. If we captured 4 of his pieces, or if 4 of our pieces were captured, then the weight A was made very high so that our program would go for the fifth capture.

Omnigon:

Hexagonal board with pieces that could move in different directions. Three pieces that could move in two directions, two pieces that could move in three directions, one piece that could move in any direction but could be captured from any direction. If a piece is pointing in a direction, it cannot be captured in that direction. Game is won when the Helios (piece that can move in any direction) is captured.

Heuristic was simple: assign value to each piece.

Value=A*(MyHelios) + B*(MyThreeWay) + C*(MyTwoWay)
Heuristic = Value(MyPieces)/Value(HisPieces)

Upthrust:

The board was setup as follows:

```
60 . . . .    10
40 . . . .    9
30 . . . .    8
20 . . . .    7
10 . . . .    6
   . . . .    5
   . . . .    4
   G R B Y    3
   R B Y G    2
   B Y G R    1
   Y G R B    0
------------------------------
   0  1  2 3
```

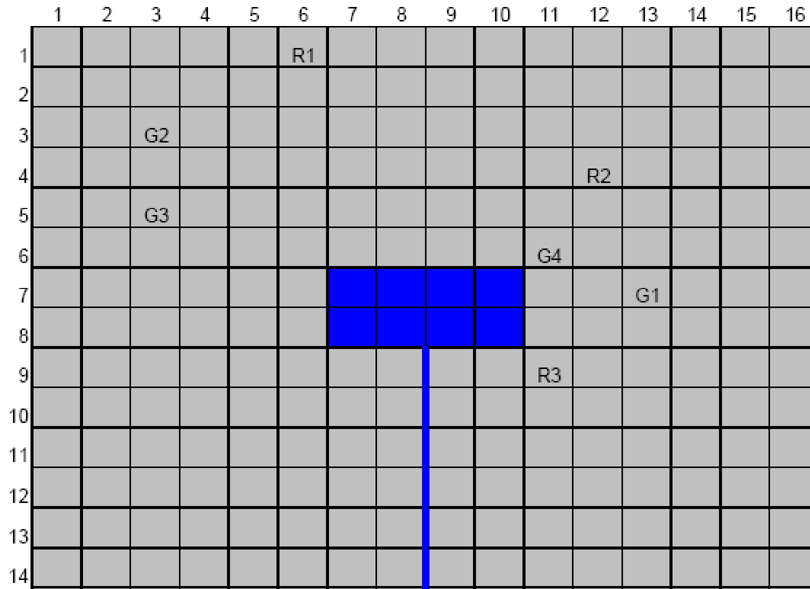
This game allowed pieces to be moved a distance depending upon how many pieces were in a row. Points were allotted for pieces that reached the other end of the board.
A simple heuristic was just:

A*(Myscore – Hiscore) + B*(sum of Y distance of all my pieces)

This is about the simplest heuristic we could get! The Y distance encourages our pieces to “move forward” and after that, the scoring kicks in to determine what makes a move good.

Billabong:

The object of billabong is to move all of your frogs around the billabong before your opponent. Frogs can jump over other frogs an equidistant amount, or move one square:



The following move would jump kangaroo G3 all the way around the billabong:

```
J 3,5 3,3 6,1 12,4 13,7 11,6 13,7 11,9
```

A simple heuristic is just the sum of the distances to the goal for all our pieces, minus the sum of the distances to the goal for all the opponent's pieces. Pieces that have not crossed the start get a value of 0. Pieces in the billabong get a value of 20.

In all cases, we had special cases checking for a win. If a winning state occurred, the heuristic should return MAXINT or -MAXINT, depending on the winner.

The general tradeoff in all heuristics is an accurate expensive heuristic vs. a more inaccurate, but inexpensive heuristic. In general the cheaper heuristics win out if this lets you search one or two ply farther than the expensive heuristic. Since you will not be given a lot of time to make a move, it is up to you to determine the tradeoff that will be best for you!