Iterative Deepening A*

Algorithm A* has memory demands that increase exponentially with the depth of the goal node (unless our estimates are perfect).

You remember that we improved the space efficiency of the breadth-first search algorithm by applying iterative deepening.

Can we do a similar thing for the Algorithm A*?

Sure!

Iterative Deepening A*

- In the first iteration, we determine a “cost cut-off” \( f'(n_0) = g'(n_0) + h'(n_0) = h'(n_0) \), where \( n_0 \) is the start node.
- We expand nodes using the depth-first algorithm and backtrack whenever \( f(n) \) for an expanded node \( n \) exceeds the cut-off value.
- If this search does not succeed, determine the lowest \( f' \)-value among the nodes that were visited but not expanded.
- Use this \( f' \)-value as the new cut-off value and do another depth-first search.
- Repeat this procedure until a goal node is found.

Let us now investigate...

Two-Player Games

Two-Player Games with Complete Trees

We can use search algorithms to write “intelligent” programs that play games against a human opponent.

Just consider this extremely simple (and not very exciting) game:

- At the beginning of the game, there are seven coins on a table.
- Player 1 makes the first move, then player 2, then player 1 again, and so on.
- One move consists of removing 1, 2, or 3 coins.
- The player who removes all remaining coins wins.

Let us assume that the computer has the first move. Then, the game can be described as a series of decisions, where the first decision is made by the computer, the second one by the human, the third one by the computer, and so on, until all coins are gone.

The computer wants to make decisions that guarantee its victory (in this simple game).

The underlying assumption is that the human always finds the optimal move.
Two-Player Games with Complete Trees

So the computer will start the game by taking three coins and is guaranteed to win the game.

The most practical way of implementing such an algorithm is the Minimax procedure:

• Call the two players MIN and MAX.
• Mark each leaf of the search tree with -1, if it shows a victory of MIN, and with 1, if it shows a victory of MAX.
• Propagate these values up the tree using the rules:
  – If the parent state is a MAX node, give it the maximum value among its children.
  – If the parent state is a MIN node, give it the minimum value among its children.

The previous example shows how we can use the Minimax procedure to determine the computer’s best move.

It also shows how we can apply depth-first search and a variant of backtracking to prune the search tree.

Before we formalize the idea for pruning, let us move on to more interesting games.

For such games, it is impossible to check every possible sequence of moves. The computer player then only looks ahead a certain number of moves and estimates the chance of winning after each possible sequence.

Therefore, we need to define a static evaluation function \( e(p) \) that tells the computer how favorable the current game position \( p \) is from its perspective.

In other words, \( e(p) \) will assume large values if a position is likely to result in a win for the computer, and low values if it predicts its defeat.

In any given situation, the computer will make a move that guarantees a maximum value for \( e(p) \) after a certain number of moves.

For this purpose, we can use the Minimax procedure with a specific maximum search depth (ply-depth \( k \) for \( k \) moves of each player).

Two-Player Games

For example, let us consider Tic-Tac-Toe (although it would still be possible to search the complete game tree for this game).

What would be a suitable evaluation function for this game?

We could use the number of lines that are still open for the computer (X) minus the ones that are still open for its opponent (O).
Two-Player Games

The Alpha-Beta Procedure

Now let us specify how to prune the Minimax tree in the case of a static evaluation function.

- Use two variables alpha (associated with MAX nodes) and beta (associated with MIN nodes).
- These variables contain the best (highest or lowest, resp.) $e(p)$ value at a node $p$ that has been found so far.
- Notice that alpha can never decrease, and beta can never increase.

The Alpha-Beta Procedure

There are two rules for terminating search:
- Search can be stopped below any MIN node having a beta value less than or equal to the alpha value of any of its MAX ancestors.
- Search can be stopped below any MAX node having an alpha value greater than or equal to the beta value of any of its MIN ancestors.

Alpha-beta pruning thus expresses a relation between nodes at level $n$ and level $n+2$ under which entire subtrees rooted at level $n+1$ can be eliminated from consideration.

Example:

$$\begin{cases}
\text{max} & \beta = 4 \\
\text{min} & \\
\text{max} & \\
\text{min} & \\
\text{max} & \\
\text{min} & \\
\end{cases}$$
The Alpha-Beta Procedure

Example:

\[
\begin{align*}
\alpha &= 3 \\
\beta &= 3 \\
4 &\quad 5 &\quad 3 &\quad 1
\end{align*}
\]

\[
\begin{align*}
\alpha &= 3 \\
\beta &= 3 \\
\beta &= 1 \\
\beta &= 8 \\
4 &\quad 5 &\quad 3 &\quad 1
\end{align*}
\]

\[
\begin{align*}
\alpha &= 3 \\
\beta &= 3 \\
\beta &= 1 \\
\beta &= 8 \\
4 &\quad 5 &\quad 3 &\quad 1
\end{align*}
\]

\[
\begin{align*}
\alpha &= 3 \\
\beta &= 3 \\
\beta &= 1 \\
\beta &= 8 \\
\beta &= 6 \\
4 &\quad 5 &\quad 3 &\quad 1 &\quad 8 &\quad 6
\end{align*}
\]

\[
\begin{align*}
\alpha &= 3 \\
\beta &= 3 \\
\beta &= 1 \\
\beta &= 8 \\
\beta &= 6 \\
\alpha &= 6 \\
\alpha &= 3 \\
4 &\quad 5 &\quad 3 &\quad 1 &\quad 8 &\quad 6 &\quad 7
\end{align*}
\]
The Alpha-Beta Procedure

Example:

\[
\begin{array}{l}
\alpha = 3 \\
\beta = 3 \\
\beta = 3 \Rightarrow \beta = 1 \Rightarrow \beta = 6 \\
\beta = 2 \Rightarrow \beta = 3 \\
\alpha = 6 \\
\alpha = 3 \\
\end{array}
\]

Propagated from grandparent – no values below 3 can influence MAX's decision any more.

\[
\begin{array}{l}
\alpha = 3 \\
\beta = 3 \\
\beta = 3 \Rightarrow \beta = 1 \Rightarrow \beta = 6 \\
\beta = 2 \Rightarrow \beta = 4 \\
\alpha = 6 \\
\alpha = 4 \\
\end{array}
\]

\[
\begin{array}{l}
\alpha = 3 \\
\beta = 3 \\
\beta = 3 \Rightarrow \beta = 1 \Rightarrow \beta = 6 \\
\beta = 2 \Rightarrow \beta = 4 \\
\alpha = 6 \\
\alpha = 4 \\
\end{array}
\]

\[
\begin{array}{l}
\alpha = 3 \\
\beta = 4 \\
\beta = 3 \Rightarrow \beta = 2 \Rightarrow \beta = 5 \\
\beta = 4 \Rightarrow \beta = 4 \\
\alpha = 6 \\
\alpha = 4 \\
\end{array}
\]

\[
\begin{array}{l}
\alpha = 3 \\
\beta = 4 \\
\beta = 3 \Rightarrow \beta = 2 \Rightarrow \beta = 5 \\
\beta = 4 \Rightarrow \beta = 4 \\
\alpha = 6 \\
\alpha = 4 \\
\end{array}
\]
The Alpha-Beta Procedure

Example:

\[
\begin{array}{c}
\alpha = 4 \\
\beta = 3 \\
\alpha = 3 \\
\beta = 3 \\
\beta = 1 \\
\beta = 6 \\
\beta = 2 \\
\alpha = 4 \\
\alpha = 6 \\
\beta = 6 \\
\alpha = 4 \\
\beta = 4 \\
\beta = 4 \\
\beta = 6 \\
\beta = 6 \\
\beta = 6 \\
\alpha = 6 \\
\beta = 6 \\
\alpha = 6 \\
\beta = 6 \\
\end{array}
\]

\[
\begin{array}{c}
\max \\
\min \\
\max \\
\min \\
\max \\
\min \\
\min \\
\max \\
\min \\
\max \\
\min \\
\min \\
\end{array}
\]

Done!