Alpha-Beta search in Pentalath

Benjamin Schnieders

21.12.2012

Abstract

This article presents general strategies and an implementation to play the board game Pentalath. Heuristics are presented, and pruning improvements to the alpha-beta framework are tested. The resulting program will be able to play Pentalath on a challenging level.

1 Introduction

Playing board game with search strategies is a wide field of research. This article describes techniques and an implementation to play the game of Pentalath using an Alpha-Beta search. Improvements are presented and discussed.

1.1 Pentalath

Pentalath is a two-player board game designed by a computer [1]. It is played on a hexagonal board with an edge length of five. Both players alternatingly put pieces of their colors, usually black and white, onto a free hex field on the board. The player first managing to get five pieces in a row wins the game. If a connected group of same-colored pieces has no free space neighboring it, it is considered trapped and the whole group is removed. A player may not sacrifice a piece by putting it into a position it or its connected pieces have no free neighbors, unless that move takes enemy pieces, thus restoring free fields next to the group.

The different fields of the board may have different strategic value, especially covering the middle position allows many more combinations to have 5 pieces in a row than a border position. To prevent the first player from always playing in the middle first, the swap rule allows the second player to swap colors in his first turn; forcing the first player to select a mediocre move first.

Common strategies involve starting many lines simultaneously or creating one long row, then capture the enemy's piece blocking it. An enemy can easily be put into zugzwang, as for example ignoring an open line of three can easily become an unbeatable open line of four the next turn. The enemy now has just four possible positions to play to, unless he can put the first player into zugzwang as well.

1.2 Alpha-Beta

The alpha-beta pruning technique is an improvement of the MiniMax search [3]. It enhances MiniMax by the option to keep track of the best outcome both

players can achieve. These values, alpha and beta, allow cutting off branches in the search tree if they are outside the bounds. Whenever a value inside the bounds is encountered, the search window is adapted accordingly.

1.2.1 Improvements

There is a multitude of enhancements known for the Alpha-Beta search. Most of them aim to improve the number of cutoffs made, other more general approaches prevent work from being done multiple times. Some possible approaches are listed here:

- Transposition Tables store information gained in an earlier branch or search. They are designed in a way that the lookup is much faster than calculating the according information.
- Move ordering tries to order the search tree in a manner that cutoffs happen earlier, thus reducing the number of nodes visited. The killer move strategy places nodes, that happened to cause cutoffs earlier, at the beginning of the move order. Moves that lead to a certain win or prevent certain loss should also be places at the front.
- Iterative Deepening Search effectively increases the number of nodes searched, but the use of transposition tables and the fact that the deepest iteration contains more moves than all earlier ones reduces the the overhead to an acceptable level. ID allows using the search results gained from shallower searches to construct better move orderings for deeper searches, which can cause a very good cutoff ratio.
- Search Windows can be used if the heuristic is known to deliver relatively constant values, regardless of the search depth. With ID, a small search window can be constructed around the result of search depth n, to be used for search depth $n + 1$. The small search window allows cutting off any value outside it. This technique is known as aspiration search.
- The Null Move Heuristic evaluates a situation in which a player does not make a move at all, although this being illegal. If a good state can be reached even after passing, any action can be chosen. This assumption will not work in games where zugzwang can lead to a loss.

2 Implementation

The game was implemented in C++, using some parts of the boost library. The game rules are hardcoded into the program, and the user can configure the modes of play using commandline arguments. The game interface is console-based, human inputs are read from the standard input and AI moves and the current game state are printed out to the standard output stream. For information about how to run the game, consult the provided readme document; for implementation details consult the HTML documentation or the sourcecode itself.

2.1 Board representation

Despite the game board having two dimensions, it is stored in a one-dimensional array internally. This makes copying and allocation tasks faster. For certain operations like printing the state, it is important to know how many fields there are in each row. Also, other operations intended to be on a graph structure, in which all neighboring pieces are connected. This information, row indices and neighbors, is precomputed and available with $O(1)$ lookup time. This allows for memory saving and efficient usage. Frequently performed recursive searches on the board, like the methods called by the heuristic functions, are mostly unwrapped to loops using a bitset of fields already visited, lowering the overhead of stack frame changes.

2.2 Transposition Table

The transposition table capacity can be controlled by altering the number of bits used for the index part of the hash. 25 bits will lead to a memory consumption of about 700 megabytes, whereas 27 bits already lead to 3.5 gigabytes allocation size. The replacement scheme is set to prefer the searched depth below the node.

2.3 Move Ordering

THe move ordering for each search step is returned from a MoveOrdering class. Depending on parameters set, different heuristcs are taken into account to order the moves. To test for improvements, simple move orders just taking the geometry of the board into account, (e.g by assuming that pieces played in the middle of the board are in general placed better than on the border) and more sophisticated ones, regarding the last played move, the best move of a former iteration and the best move recorded so far in the transposition table are tested against each other.

- "Naive" move ordering uses the order of fields on the board.
- "InsideOut" move ordering assumes that as moves in the middle have most dynamic, they are to prefer in general.
- "LastMove" sorts the possible moves around the last played move, assuming that a reaction on a move is going to be close to the placed piece.
- "PreviousBest" sorts the remaining moves around the move that was found by a previous search iteration or saved in the transition table.
- "History" makes use of a history heuristic table tracking previously considered killer moves. Still, the "PreviousBest" move is placed in front. As the move structure in Pentalath is extremely simple, every empty field is a potential move, and any situation is highly unlikely to be reachable by both players, no butterfly table is needed, a simple table with 61 entries for both colors suffices.

2.4 History Heuristic Table

A history heuristic table was implemented to improve move ordering. One table holds a separate table for each color, as killer moves do not lead necessarily to a

cutoff move for the other player. As found by Winands et al. [2], the increment of the table entries when encountering a cutoff is not hat important, thus, the value is simply incremented by one. After a move is played, it can only be played again if the piece was removed. Else, it is immediately rejected, not causing much overhead.

2.5 Matches and Tournaments

A very simple Match class was added, to let two players automatically play against another, where both players would be the first to move one time. Between the two games, the player information is reset. This is especially important for tournaments, as players might otherwise gain extraordinary knowledge in their transposition tables.

A Tournament class allows automated tournaments between agents. The two main parameters list a number of players wanting to compete and a number of matches to be played between each two players, to minimize random effects. The tournament features multi-threading using OpenMP. After all matches have been played, a result matrix can be retrieved.

3 Playing Pentalath

Pentalath has an initial branching factor of 61, as there are 61 fields one can put a piece on. However, mirrored and rotated versions of the board are of equal gameplay value, which reduces the branching factor if the search can account for that. A rotation and mirror invariant hashing function for the transposition table could accomplish this. With ongoing gameplay, the branching factor reduces gradually, as fewer and fewer positions can be played. The average branching factor can be estimated to be around 30, assuming the game will end shortly after the first inevitable taking move.

3.1 Game Start

The first move for the second player might be a color swap. This is the only possibility such an extraordinary move can happen, but the first player will have to consider the second one "stealing" his move, so he will have to pick a less good starting position.

In order not to confuse the AI with this detail, the results for the first two moves are precomputed. The AI will assume that only pieces on the outer two rings will be spared. The outer ring is however a quite bad starting position, so it places its first move randomly in the second outer ring. If the first move went to inner 3 rings, the AI will swap colors, as shown in Figure 1.

3.2 Gameplay Variables

Pentalath is a rather simple game, besides the first move equalizer, there are no special rules for placement or special rules for different pieces. Heuristic functions have to cover at least the main aspects in the game, which are reaching the goal condition to have 5 pieces in a row and the taking move, which removes enclosed pieces. These heuristic features are then weighted and the result is added up.

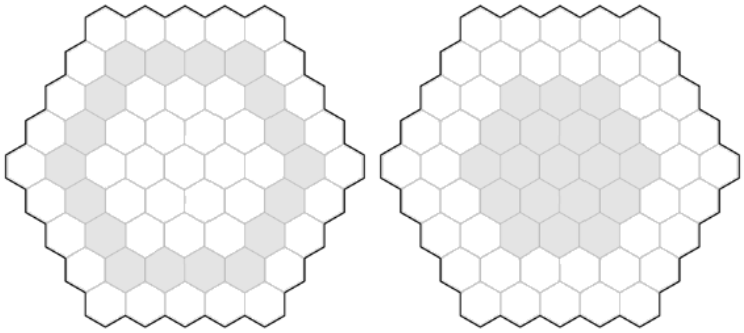

Figure 1:
Left: The shaded area shows possible starting moves of the AI. Right: An opponent's first move inside the shaded area will be taken.

$$\begin{aligned} Heuristic(state) &= w_1 \cdot Combinatorial\,Heuristic + \\ &\quad w_2 \cdot Enclosing\,Heuristic + \\ &\quad w_3 \cdot Piece\,taken\,penalty \end{aligned}$$

3.2.1 Combinatorial Heuristic

As the winning condition is to put five pieces in a row, obviously the number of pieces in a row correlates with winning. However, only rows that can possibly be extended to five should be counted, and whether the ends of the row are empty or not plays a major role. Additionally, the number of different rows is important.

The heuristic calculates for every possible placement of five pieces on the board if they can form five in a row (without being blocked by border or enemy pieces) and how many pieces of 5 are already placed. The number of already available pieces is multiplied with the number of occurrences, then, regarding the already present line length, the partial results are weighted.

Having two possibilities to complete 4 in a row to 5 is regarded equally good as having 5 in a row already, as the enemy is only able to block one possibility. From that reasoning, the function is extended, two possibilities to complete rows of three to rows of four are regarded equally good as having one row of four already, and so on. The weighted values are summed up to a single integer number.

$$Combinatorial\,Heuristic = \sum \begin{cases} \#1of5 \cdot 1 \\ \#2of5 \cdot 2 \\ \#3of5 \cdot 4 \\ \#4of5 \cdot 8 \\ \#5of5 \cdot 16 \end{cases}$$

3.2.2 Enclosing heuristic

To understand taking pieces, the enclosing heuristic calculates the "degrees of freedom" a connected group of pieces has. This is defined inversely by the

number of enemy pieces or border fields directly around any connected group of own pieces divided by the number of free spots around it. Thus, the more covered by enemy pieces a group of stones is, the higher the enclosing criticality value will rise. The value for the most critical group is then multiplied by the number of pieces being in danger, so that the AI might actually sacrifice a piece in order to get to a better state.

$$Enclosing\,Heuristic = max_{\epsilon own\,pieces}(\frac{blocked\,neighbors}{empty\,neighbors})\cdot\#endangered\,pieces$$

3.2.3 Piece taken penalty

The enclosed pieces criticality value has a flaw - it drops back to zero or the next critical value after pieces have been captured. An AI thinking ahead might thus plan over the taking move, then forget that it was in a better state once. Here, the piece taken heuristic returns just the number of pieces that have been captured from the player so far.

$$Piece\,taken\,penalty = \#pieces\,lost$$

3.3 Time limit

Without a time limit, an AI may be tempted to search through the whole search tree in order to win. As the AI should be able to fight in a tournament, a time limit is set for the whole game, allowing both players an equal amount of calculation time. The thinking times for both players are measured and subtracted from their own time rations. At the end of the game, the time left is printed out, if the time left is below zero, the AI might be disqualified for not complying with the time limit.

3.3.1 Estimating the number of moves

To convert the total time left for a game to a move time limit, the AI must make assumptions about the number of moves in the whole game. As the AI was not trained against a number of other AIs, the only obvious move limit is reached when the board is getting full - at this point, at least one capture happens, and after that, it is probably easy to win, so planning long seach times after 61 moves might be a waste of time needed earlier.

The AI subsequently assumes that the game will be over after 61 plys, or, if 51 plys are reached, constantly assumes it will take 10 more plys.

3.3.2 Estimating search depth

From the estimated number of moves and the time left, the average duration of the remaining plys is created. The AI uses an averaged history of search durations to estimate how long the next search of depth n will take. As the search time rises exponentially with search depth, often the AI has to decide between a search depth that finishes very quickly and the next search depth, greatly exceeding the planned soft-threshold duration.

Instead of always choosing cautiously the lower search depth, or optimistically choosing the deeper search, the AI lets a weighted random value decide.

The ratio of differences between the move time threshold and the quicker and slower search depth is calculated, and a random number is generated. The error ratio is inverted to produce a positive measure, then the random number is compared to it. If it exceeds the ratio, the upper bound will be taken, the lower bound in any other case. This leads to a statistically ideal fit to the duration planned to use, as long as enough samples are drawn, i.e., the game has enough moves. Also, in the early game, the averaged move times might not return a good estimation, as the search time heavily depends on the board configuration.

3.3.3 Soft and hard limit

The previous approaches describe soft limits, that the AI should take into account and try to stick to. In addition to these soft limits, hard limits should be implemented. There is one hard limit, switching the max search depth to 4, which proved to be a search that is still extremely quick, that is, it is barely measurable by a human. This hard limit is reached if there are only 10 seconds left for the rest of the game, assuming that with such quick moves, it can stay below the time limit indefinitely.

Another hard limit may be implemented by running the search in an own thread, prematurely returning the result of an earlier iterative deepened search, and then trying to stop the still searching thread before the enemy finishes the move. This requires more checkpoints inside the search, which may lower the performance noticeably.

3.3.4 Selective search depth

If the search returned before the time is over, the AI should be able to decide whether to save the surplus time for later moves, or if it is not yet contend with the search results obtained so far. For that, it may observe the principal variation of search results, and in the case of heavy fluctuations, decide to do another search step, but a better way might be to skip the deepest search depth if the principal variation does vary only slightly.

4 Results & Discussion

The AI plays Pentalath using search depth 6 on a challenging level. Constantly using depth 7 showed to be too slow for humans to have an interesting game. Deeper search seems to be correlated with better gameplay, however, the odd-even effect of the heuristic distorts the results.

4.1 Heuristic weighting

Different heuristic weights were tried and AI competitions held between all combinations of heuristics. Superior weights were chosen and refined and the tournament was repeated. This way, a good heuristic setting could be found for playing against other heuristic settings. However, tests against humans and AIs with entirely different heuristics functions are needed to provide truly good heuristic functions weights.

4.2 Enhancements quality

AIs with and without any combination of enhancements were tested in a tournament against another. Although some combinations seem to be superior against other combinations, this could not be reduced to a single factor. Even though the results are inconclusive, one can at least choose the fastest AI against an unknown opponent, placing reliance on the fact that a faster AI may search deeper. The following results were achieved with a fixed depth and without timeout, with all random factors disabled. Still, hash collisions might produce slightly different gameplay if a transition table is used.

4.2.1 Transposition Table results

The following findings were produced playing a game against an AI with and without using a transposition table. The search depth was limited to 5, no iterative deepening search was used. The move ordering was statically set to order moves inside out. Table 4.2.1 shows that even a smaller transposition table with about a million entries can speed up the search significantly.

No Transposition Table			Transposition table with 20 bit key, 64bit hash			
Ply	#Nodes	Search Time (s)	#Nodes	Search Time (s)	Savings (%)	% Key collision
2	286.140	1.40375	161.210	0.790422	43,66%	19,87%
4	245.630	1.21277	136.932	0.671332	44,25%	33,26%
6	178.778	0.914445	95.235	0.481575	46,73%	39,65%
8	222.024	1.16595	132.081	0.699028	40,51%	49,53%
10	450.795	2.37601	316.073	1.66693	29,89%	71,39%
12	370.187	1.93338	251.183	1.28362	32,15%	74,53%
14	443.795	2.29888	301.659	1.55834	32,03%	77,60%
16	5.364.066	30.4163	1.967.902	10.9326	63,31%	73,37%
18	461.413	2.46244	271.638	1.41432	41,13%	93,57%
20	1.177.212	5.72802	365.458	1.79572	68,96%	95,21%
22	2.319.394	11.4165	177.894	0.850502	92,33%	94,43%
Avg.	1.047.221	5.57532	379.751	2.01312	63,74%	65,67%

Using larger transposition tables does not seem to improve performance with low search depths. Although the average number of index key collisions lowers from 65,67% to 5,40%, the number of searched nodes does not drop much. An explanation for this is that although many more nodes are stored, most of them do not lead to a window shrinkage. For deeper searches and longer matches, larger tables are to be preferred.

4.2.2 Iterative Deepening

As iterative deepening causes overhead initially, a transition table is used to minimize the overhead by providing a move ordering for the next iteration. Note that only the history heuristic table move ordering and the “previous best” ordering can benefit from this information. The previous best move is used as first node here. A 6-ply deep search was used to produce the following results.

No Iterative Deepening			Iterative Deepening		
Ply	#Nodes	Search Time (s)	#Nodes	Search Time (s)	Savings (%)
2	296.825	1.58056	537.063	2.77354	-80,94%
4	280.841	1.50791	453.338	2.43694	-61,42%
6	342.068	1.85501	342.059	1.86098	0,00%
8	818.088	4.41454	395.021	2.12578	51,71%
10	883.207	4.90314	590.703	3.29848	33,12%
12	921.054	5.16992	555.301	3.13974	39,71%
14	2.808.208	15.3428	1.771.998	9.66481	36,90%
Avg.	907.184	4.96769	663.640	3.61432	26,85%

Table 4.2.2 illustrates the overhead of iterative deepening: During the first searches, the transition table is not yet filled well, thus does give fewer support. Also, for the very first moves, there is no particular best move, so that the best move achieved from the last iteration provides less cutoff. At later searches, however, ID shows its strength and in average saved nearly 27% in just a 14 ply game.

4.2.3 Aspiration search results

It proved to be hard to find an acceptable Δ to construct a search window $(Value - \Delta, Value + \Delta)$ around the $Value$ returned from the last ID search, as the heuristic varies heavily with search depth. For the heuristic used, the odd-even-effect is very strong, also, Pentalath is a game in which every move might be game outcome changing, meaning that not only the heuristic, but the actual game value varies when looking one step further ahead.

For all Δ values tested, aspiration search either did not result in an improvement, or the search failed too often to produce better pruning results, see Figure 2 for results. Aspiration search is not used in the final version of the game.

4.2.4 Move ordering results

Using a different move ordering strategy leads to quickly diverging gameplay. Because of that, only the results of the dynamic move orderings are compared from a complete game. Table 4.2.4 presents the improvement of using the best move of a previous iteration or found in the transition table compared to a more naive order around the last move. Again, the results were generated from 6-ply deep search. Additionally, all AIs used a transition table of exact same layout.

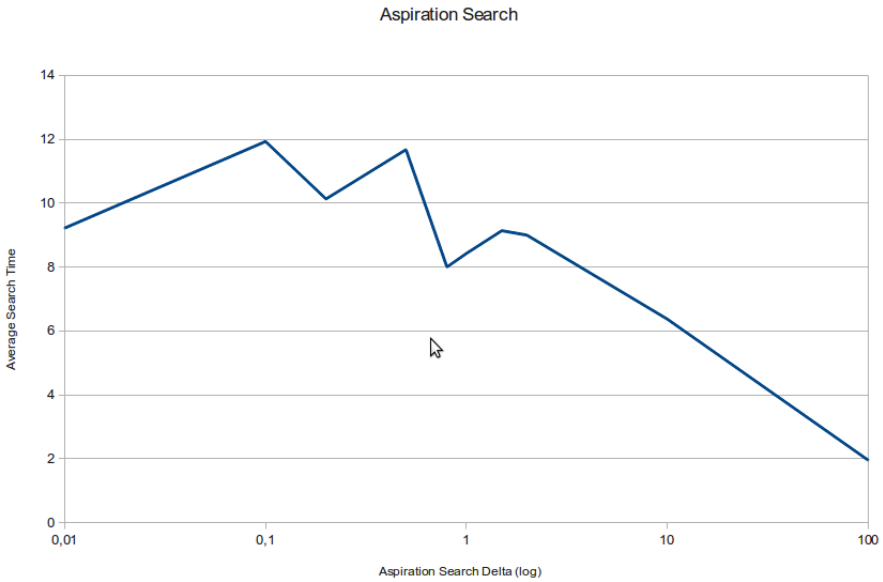

Figure 2:
Graph of aspiration search windows ($\triangle$ as a factor of the last search result) versus average search time. Note that $\triangle = \infty$, equaling a search without window, is listed at $\triangle = 100$, and that the X-Axis is logarithmically scaled.

”LastMove”			“PreviousBest”		
Ply	#Nodes	Search Time (s)	#Nodes	Search Time (s)	Savings (%)
2	2.952.245	16.8167	537.063	2.95432	81,81%
4	4.333.205	25.3447	410.206	2.27277	90,53%
6	2.799.612	15.9292	641.355	3.58721	77,09%
8	5.834.460	33.7243	485.404	2.72457	91,68%
10	2.386.468	14.6336	526.188	3.01307	77,95%
12	3.586.191	21.6713	1.024.846	5.99585	71,42%
14	3.694.822	22.5610	1.360.684	7.50856	63,17%
16	6.342.506	39.4383	1.804.225	10.7666	71,55%
Avg.	3.243.130	19.3073	931.178	5,4000	71,29%

Using a previously considered best node first in the move ordering leads to many cutoffs. However, if the first move proves not to be the best one, ideally the second move should be. Instead of just ordering moves spatially around the best one, the history heuristic table counts the times a particular move generated a cutoff. Below, Table 4.2.4 presents the improvement when adding a history heuristic table to “PreviousBest”

”PreviousBest”			“History”		
Ply	#Nodes	Search Time (s)	#Nodes	Search Time (s)	Savings (%)
2	537.063	2.95432	605.751	3.26131	-12,79%
4	410.206	2.27277	567.101	3.16332	-38,25%
6	641.355	3.58721	591.662	3.30183	7,75%
8	485.404	2.72457	374.716	2.13307	22,80%
10	526.188	3.01307	497.326	2.74505	5,49%
12	1.024.846	5.99585	643.269	3.58895	37,23%
14	1.360.684	7.50856	365.604	2.01105	73,13%
16	1.804.225	10.7666	454.608	2.58329	74,80%
Avg.	931.178	5,4000	495.071	2,742596	46,83%

For the first moves, the history is not able to generate meaningful results, leading to a worse move ordering past the best move than a spatial one. After the table contains some entries, the move order quality quickly rises and improves “PreviousBest”. A possibility to fight the startup problems might be a static initialization of the table, or a combination of the approaches with a variable weight that switches to History fully after some plys. Not all searches gain the same performance boost through better move ordering, for example if the best move really is the already supplied one.

4.2.5 Null Move results

Using a null move non-recursively for the max player on a subtree with a search depth reduced by two produces best gameplay results. On a sum of games, the regular AI searched an average of 1.998.558 nodes per move. Null move reduced this number to 613.085, which is a saving of 69,32%. Of 20 Matches (40 Games), the AI not using null move won 24 times, so there seems to be a tradeoff between better gameplay and performance improvement, which should be explored further.

5 Conclusions

Alpha-beta has proven to be an effective framework to build an AI for Pentalath in. However, against certain human strategies, the achieved 6 ply lookahead does not suffice. Humans often can understand a possible future structure of pieces, seeing a win easily up to 8 or 10 moves ahead. If humans however are unable to play such “exploit moves”, often the AI wins, because 6 ply lookahead is already pretty far. Combining a transition table with iterative deepening search and using it for move ordering provides dramatic search speedup. Implementation of a history heuristic table is quickly done for Pentalath, and gains some more performance. Using null move not very aggressively provides some more performance, but at a lowered gameplay quality.

To improve gameplay further, some possible improvements might be the following:

- Implement more heuristic features
- Learn optimal heuristic weightings using Temporal Difference Learning against humans and other AIs.

- Search extensions for promising search states.
- Learn losing moves from frequent plays and save them in a separate transposition table.
- Implement a transition table that can be accessed from multiple threads lock-free to allow multi threading
- Rotation and mirror invariant hashing method to reduce search overhead
- B* seems like an applicable approach, if one can write a heuristic function returning a pessimistic an and optimistic measure.

5.1 Future Work

Pentalath is a game that - if playing against humans - either requires very subtle heuristics, or a search depth up to 20 plys, as humans provably can plan moves vaguely 20 plys ahead. Heuristics that restrict the gameplay possibilities further, i.e., making the AI more defensive, might lead to easier wins against humans, however, deep-searching AIs might be an even harder opponent then. Using B* search with a heuristic obtained from 2-ply-deep search might overcome these effects, as there is no more search horizon. It should be investigated if B* can beat alpha-beta search with common memory restrictions.

References

[1] Cameron Browne. Pentalath. Retrieved December 20, 2012, from http://cameronius.com/games/Pentalath/, 2007.

[2] H.J. van den Herik M.H.M. Winands, E.C.D. van der Werf and J.W.H.M. Uiterwijk. The relative history heuristic. *Lecture Notes in Computer Science*, 3846:262–272, 2006.

[3] Stuart Russell & Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, second edition edition, 2002.