Connect-K Hints, Caveats, and Heuristics

In order to inspire your creativity as AI algorithm designers, collected below are some of the hints, caveats, and heuristics that previous Connect-K students have contributed. They may be useful to your reflective introspection as you think deeply about the problem and its constraints. Remember, your goal is to write an AI that will defeat all of the heuristics mentioned below.

A Few Hints and Caveats

General Hints

1. There were actually a lot of difficulties and surprises with the implementation. Book keeping is not as easy as it looks. Generating children, making sure we start with the best found before so we can prune more, hashing and trying to speed things up, alpha beta errors and others.

2. The biggest hurdle was wading through the early parts of development when we weren't sure whether the AI was performing poorly because the evaluation function was terrible or there was a bug in the search functions. We realized that we had to thoroughly test all of our functions to be sure everything was performing the way we wanted, as is the case with most development, but because search and evaluation are dependent on one another it's hard to decide which to debug first.

3. ... one of us had an earlier version of the provided shell with an unfixed bug, and it took us some time to realize that we needed to have pulled the latest version from the website.

4. Start earlier.

Technical Hints --- Language Related

1. Warn students ahead of time that it will be significantly harder to debug when using C++. I had to attach my debugger to the running java GUI process every time I wanted to debug my code while playing the game. It wasn't bad once I got used to it but still added a significant amount of time compared to debugging it in Java. The benefit of writing it in C++ is that given a Java AI and a C++ AI, everything else equal, the C++ AI should do better since C++ runs faster than Java so you can search deeper.

2. Specify NOT to use Windows specific libraries or code. *(Note: We may run your tournament code on a Linux cluster.)* Let the C++ students know which platform they should code against, for java it doesn't matter. *(Note: We'll run the tournament on SGE or a lab machine. The C++ target platform should be x86. You should write your code to run on any x86 machine. The OS is CentOS 6. We most likely will need to compile your code with CentOS 6 (RHEL 6) x86_64. Machines in the openlab.ics.uci.edu (family-guy.ics.uci.edu) are CentOS 6.)*

3. We wrote our program in C++, which made timing somewhat more difficult than in Java, and we decided not to include any libraries that would have made it a lot easier to work in - which ended up being a good thing for the tournament, as we had considered using Windows specific libraries or code. However, not being able to use gdb or another debugger (or perhaps we could, and we simply didn't do it correctly) with this project was a major hindrance, and we were plagued with errors that would have been simple to solve had we had more ability to look at the search state outside of print messages.

4. Memory allocation/deallocation by the OS is generally quite slow compared to purely internal computation. This may be especially true if you make many small allocate/deallocate requests, or you force the Java garbage collector to collect lots of garbage. Try to allocate what you need at the beginning, and then reuse/recycle it internally yourself instead of going through the OS.

Technical Hints --- Algorithm Related

1. ... we decided to record the best result of each depthLimit and return the move we got from the max depthLimit before time was over. [Need to emphasize more strongly that they should always do this.]

2. The one thing that sticks out as being difficult was keeping everything straight with alpha beta pruning. The MIN-MAX portion was fairly simple to get working, but once we added alpha beta pruning we ran into a few problems with children not getting pruned correctly or at all.

3. In my implementation, having an object that stored actions together with the evaluation score that they garnered proved much simpler. I consulted the pseudocode in the book to implement the IDS. The most difficult portion was returning an answer after 5 seconds, rather than the IDS itself.

4. We combined the Iterative Deepening Search with Alpha-Beta pruning algorithm. The speed improved to some extent.

5. We also pre-pruned those moves that do not make any sense. For example, we don't calculate moves like very far-away from “battle field”. Only the cells with stones nearby will be evaluated.

6. We store the child node information every step of the way. We create a class IDNode to store information like the current board state, the total moves played, which player is playing and so on. Every node in the game tree is an instance of IDNode. However there is a downside to it: it is memory consumptive since each time we go deeper with IDS, a bunch of new child nodes will be created.

We also sort the stored nodes using Collections.sort. The sorting is a big help to Alpha-Beta pruning. However it is worth noting that the sorting itself is also time consumptive.

Technical Hints --- Weighting

1. ...Since we now had two features that were used to form the utility value, we used the dot product of a vector of weights, giving each feature a weight of 1 to start. We saw a change in AI's behavior in a way that it now was not focused on making moves towards the middle which would benefit its own final configuration. Instead, the AI positioned its moves around the opponent's pieces, so as to eliminate possible threats made by the opponent. We thought that the focus was shifted too much from trying to win, towards making sure that the opponent doesn't win, so we adjusted the weights to give the win paths feature a weight of 1, and the threats feature a weight of 0.5. This seemed to produce a smarter, more optimal play in our opinion.

2. To figure out the weights for our evaluation there was a lot of trial and error. We actually completely scrapped our evaluation function 3 times.

3. We tried a few variations (including initializing the weights to all zeros and initializing the weights to randomly-chosen values), but we had the best success when we set the initial values manually, based on what we thought might be the most important features.

Technical Hints --- Quiescence

1. We included the quiescence test inside the evaluation function. This is because both functions were doing similar things and it made our code more efficient.

2. Our quiescence test checks whether the opponent is about to win and if so, continues the tree expansion one more level to see if the game may resolve itself. Specifically, the test consists of checking for any opponent threats, and letting the search continue if the opponent has one or more threats. A threat consists of any line of k cells on the game board which has a single empty piece with the rest of the cells filled by opponent's pieces. That way, if there is a cell which the opponent can occupy in order to win, we want to make sure to allow heuristic calculation on the next level, so that the node can be dismissed as a losing node, instead of a regular calculation based on the Eval(S) function. Unfortunately, we did not see much improvement in the AI's logic when adding the quiescence test. Perhaps our board was too simple to notice any changes, and did not suffer from the horizon effect to begin with. On more complex boards, it was difficult to tell whether the AI was making a better or worse move with or without quiescence, since we could not beat the AI either way.

3. We implemented the Quiescence search in a method called game_over(). This function enforces a few situations in which the game is already decided or will be immediately decided. If the advantage is on our side, this test allows further search down this path (which will have large utilities by the heuristics anyway). If the advantage is on the opponent's side this test would immediate forward the search to the next parent branch that does not fail the test. Indeed this is a very important feature of the AI and without it the AI is prone to making moves that would result in a loss in 1-2 pieces. The test situations do have to be hard coded in our cases.

Technical Hints --- Heuristics (and some weighting)

1. Locality in search

One clever thing we did with our heuristic evaluation function was to update our tallies considering only the piece that was just placed. We would start by calculating the tallies for the entire board, but after we got these tallies, we would run a different function to consider only the new piece for each new node. By only looking at the frames that were directly connected to this piece, we could update the board tally without needing to look at the entire board. This method was much more efficient, and without it, we would probably be unable to run this heuristic evaluation function at a reasonable speed.

2. Win move

This feature simply returns 1 if the current player has won and -1 if the opponent has won. The weight is large enough to outweigh all other features since winning is a game-ending state.

3. Partly-connected pieces

This feature scores the board based on the size of the partially connected pieces. Basically, we only consider connected components that have between floor(K/2) and K-1 pieces. For example, when K=6, connected components of 3, 4, and 5 pieces are scored relative to their size. For example, given a K=5 game, we assign a 3-connect configuration 250, and a 2 connect 83.3. A diagonal configuration is weighted 20% more than a vertical or horizontal one.

4. Threat move

Any move that leads to possible win is called a threat move. We assign a “threat score” (i.e., 750) for each threat move. This case generally happens when there is a K-2 component on the board where the next move would guarantee the current player a win. Conversely, blocking threat moves from the opponent is weighted twice as much. Finally, moves that would result in the opponent winning are also weighted heavily. Note that for the sake of efficiency, the partly-connected pieces and threat move features are computed in one function.

5. Pieces near center

We found that pieces placed closer to the center of the board tend to have more winning paths. The feature is simply a sum of the pieces in the center, weighted relative to their distance to the center.

6. Occupy empty rows

An empty row is defined as a row where the player has at least one piece and the opponent has none. Empty rows can be vertical, horizontal, or diagonal. Each row is given a weighted score relative to their distance to the center of the board.

7. Winning paths

Since we needed to enumerate through all possible lines on the game board and check each one for potential win paths, we pre-computed all winning lines of the board in a pre-processing step. That way, when the heuristic calculation runs, we just need to traverse the vector of points that make up each line and track the current values in the game state. We took the difference between the AI winning lines, and the opponent winning line to come up with a utility value. In the same piece of code we also added the check for whether the AI or the opponent have won the game, outputting a maximum heuristic value if the AI won or a minimum heuristic value if the opponent won. This heuristic dramatically improved the performance of the AI in practice, and we saw the UI making "smarter" moves where it would block the opponent's line formation and try to favor its own line building. Since the heuristic is general for a game with gravity off, we saw that when gravity was on, the AI had the tendency to fill up the middle row all the way to the top.

8. Number of connected lines of shorter lengths than K

The heuristic scoring method (evaluation function) we adopted is based on the number of connected lines of shorter lengths than K. Initially, we formulate heuristic scores for all types of combinations in a line, and whether lines are blocked by opponent or not (see attached heuristic scoring table). We used 7 features to evaluate the board state, and the weight vector of each features were selected to address the importance of its role. 6 features counts number of connected line in length of K-4, K-3, K-2, and K-1. The last feature counts potential number of ways to win given board state. Also, the length of K-1, and K-2 lines are so special that we treat them as an indicator of winning or losing. By experiment, we found several rules that lead a player to win based on such K-1 and K-2 length lines.

9. Example features of which to keep track

Our vector of features is designed as follows:

1) K or above K continuous pieces existed on the row/column/diagonal (ImmediateThreats)
2) Number of row/column/diagonals that a player wins (WinningLines)
3) There are K-1 pieces on row/column/diagonal, however, there are only on free position for putting pieces forward and behind (IndirectThreats)
4) K-1 continuous pieces on the same row/column/diagonal, there are two pieces outside the (MustWinThreats)
5) K-2 continuous pieces on the same row/column/diagonal, however there is one free piece forward and one player's piece. This means when the player put the key on the free piece, he will win the game (IndirectThreat_withPosition)
6) K-2 continuous pieces on the same row/colum/diagonal. There are two unoccupied pieces beside the k- 2 pieces (K_mins2_free)

10. Example weighting of features
We scan the board and rate each of the patterns like below:
If a pattern is blocked on one side, it's called “dead”. If both sides are not blocked, it's called alive.
1. Winning state. 100 pts
2. (Alive k-1) or (double dead k-1) or (dead k-1+alive k-2) 90 pts.
3. Double alive k-2 80
4. Dead k-2+Alive k-2 70
5. Dead k-1 60
6. Alive k-2 50
7. Double dead k-3 30
8. Alive k-4 20
9. Dead k-4 10
10. Other 0

11. Number of the connected pieces
Our heuristic evaluation function is simple and clear: counting the number of the connected pieces. We start scanning the board from (0,0), row by row while column by column for each row, to build a max-min tree from three axises : horizontal, vertical and diagonal. Every time a new piece that will be added to the consecutively connected pieces for the starter side is found, we increase the heuristic value (denoted as max) by 1. If the max value for the next step is expected to be K, it would be given K*2 to signal that it is about to win. The same logic is applied to the opposition side, with the min value monitored instead of max value. The Eval(S) function will return the value of max-min.

12. Spaces
This is a very lightly weighted heuristics (by a factor of 1/pieces played) that simply prefers a move with more empty spaces around than a play close to the opponent. If a more complicated heuristic returns a non-trivial value it will not matter.

13. Winning lines
Our heuristic function uses two features: Firstly, we care about how far pieces are from the center, which helps the AI choose a move at the center of the board in the early game, and is lightly weighted as not to overwhelm our second feature, which is potential winning lines and their lengths. Winning lines are lines that contain k of the following in some combination:
* At least one AI PIECE
* Any number of NO PIECE
* At most zero HUMAN PIECE
They can occur horizontally, vertically, and diagonally.

14a. Ranking features (aka weighting)
In our case, we decided to only calculate the value around the most recently placed piece so the heuristic isn't being calculated on the entire board. The heuristic score is calculated by looking at the newly formed connections in the 8 directions around the piece that was placed. The score itself is classified by tiers. A “Tier 1” score corresponds to a winning move while tiers 2-9 get progressively less valuable. The tiers are separated by a gap large enough that it is near impossible to bridge between them; even with multiple tier 3 connections you will not beat a tier 2 score.

14b. Preemptive pruning
We decided to implement the heuristic in two parts: a cutoff check, and the actual heuristic evaluation itself. The cutoff check is a check we used before even pruning. It decides if a node was so good or so bad (for instance, a winning move or a losing move) that it didn't even need to be expanded, thus saving a huge number of nodes from being expanded and often cutting off very large sub-trees. The check itself is done by calculating the heuristic on the move and then seeing which tier the move fell into. If there were no tier 1 moves, we would then take all of the moves from the next best tier and move forward with an evaluation. Since we were able to know, based on the board configuration and rules of the game, that some moves are inherently better than others we exploited this fact to provide the greatest pruning possible. The heuristic for determining the tiers was based on a combination of connection lengths and open spaces to provide an accurate ranking of each move.

15. Connected lines
We used 7 features to evaluate the board state, and the weight vector of each features were selected to address the importance of its role. 6 features counts number of connected line in length of K-4, K-3, K-2, and K-1. The last feature counts potential number of ways to win given board state. Also, the length of K-1, and K-2 lines are so special that we treat them as an indicator of winning or losing. By experiment, we found several rules that lead a player to win based on such K-1 and K-2 length lines....

16. In a row
My heuristic function awards positive points for the maximal number of pieces that max has “in a row” and negative points for the maximal number of pieces min has “in a row.” Except in states where a player has achieved the goal, it sums these scores (max + min) to produce the total evaluation score for a state. In states where a player has achieved the goal, it awards +k*2 if max has won or -k*2 points if min has won.

The algorithm does not merely count the maximal number of contiguous pieces. That technique would fail, because k-1 pieces in a row that are blocked from expanding, by borders or the opponent, to the full win length, k, are not of any utility to the AI. Instead, the algorithm counts the maximal number of pieces in every possible k-length path that is uninterrupted by pieces from the other player. Finally, the algorithm takes into account one special case of a state that will result in a forced loss.

17a. Alternative board representations

We used a simple heuristic that looked at every frame of the board. A frame is defined as K consecutive board spaces, where K is the connect length. The heuristic uses two two-dimensional array as “scratch paper” for each player. The heuristic analyzes every horizontal, vertical, up-right diagonal, and up-left diagonal frame in the board. This is represented by the first dimension in the heuristic array. In a single frame, the heuristic counts the total number of X's and O's (1's and 2's in the program). If the frame has both, it is disregarded. Otherwise, the second dimension of the heuristic array is incremented at the index of the size. For instance, if there are 2 X's in a horizontal frame, ie (X_X_), the heuristic would index the array position [0][2] by 1. At the end, each heuristic array is converted to a number by the simple formula of score+=i*array[j][i]*5^i, for all i and j.

17b. Weighting features

There are 3 aspects of this formula. The first is for any number of pieces in a frame, the base score is five to the power of that number. We initially used the number of X's (or O's) in a frame squared, but the heuristic valued clusters over large pieces-in-a-row. With a connect length of 4, this behavior is not optimal. We then used two to the power of pieces-in-a-row, but the clustering problem persisted. Based on the fact that given any space is adjacent to eight other spaces and if there are more than four occupied neighbors, then there has to be a consecutive three pieces-in-a-row. The second part of the equation multiplies base score by the number of in-a-rows found. This makes two two-in-a-rows worth exactly double a single two-in-a-row. The final piece of the equation is the scalar based on the number-in-a-row. This is to zero out empty frames. This is necessary due to the bias implemented in converting the score for X and Y into one score.

17c. Considerations of player ordering

Based on the last piece placed, a biased is given to the other player. We used a bias multiplier of 1.4. This means that if both players complete one three-in-a-row at the same time, the player who goes next has the advantage. Zeroing out empty squares is necessary in order to avoid overly biasing the center of the board. The final score is simply the score of the player who plays next, multiplied by 1.4 and then added to the negative score of the other player. Due to this, the search is maximin (instead of minimax) for even depths (or searching plies).