Knights Isolation Project Report

Scope

The main section of this report answers the questions required for an implementation of a Custom Agent with an advanced heuristic.

Optional appendices provide a little more detail

Brief experimental data is provided to support the choices made.

Unless otherwise stated , results tables show competitions between two players of 100 matches - consisting of 25 rounds of fair matches with 150 millisecond turn timeouts:

```
run_match.py –f –r 25 –t 150
```

The *baseline* heuristic for comparison is the *relative liberties* heuristic provided in the sample `MimimaxPlayer` code.

Contents

Search Algorithm

For my custom search algorithm I implement basic Minimax with Alphabeta Pruning and Iterative Deepening (as described in the lectures).

Results

The table below shows competition results which demonstrate the effectiveness of

a) Iterative Deepening over a fixed search depth
b) Alphabeta pruning over vanilla Minimax

Player One	Player Two	Player One Wins	Player Two Wins
Minimax – fixed 3 ply search depth *Baseline heuristic*	Iterative Deepening Minimax *Baseline heuristic*	23%	**77%**
Iterative Deepening Minimax *Baseline heuristic*	Iterative Deeping Alphabeta *Baseline heuristic*	17%	**83%**

Custom Heuristic

My heuristic is the unweighted linear combination of the two heuristics described below:

Relative Control Heuristic

This heuristic counts the number of empty squares that a player can reach before his opponent can (or that his opponent cannot reach at all).

I call the squares a player can reach before his opponent, the squares under his 'control'.

A player's *relative* control score is the number of squares he controls minus the number his opponent controls.

Relative Minimum Remaining Moves Heuristic

This heuristic calculates a lower bound (i.e.it is always less than or equal to the actual value) on the minimum number of moves a player can make from the current position against any play by his opponent.

That is, the player can provably make at least this number of moves from the current position regardless of what his opponent does.

A player's *relative* minimum remaining moves is his minimum remaining moves minus his opponent's minimum remaining moves.

Combination Heuristic

I use these two heuristics in combination because

a) They can both be calculated using a single pass of the same algorithm (see Appendix Two) which has a worst case time complexity linear in the number of remaining free squares.
b) They are complementary heuristics
    i. *Relative Control* can be regarded as a *strategic* evaluation. The more squares a player can reach first the more likely he can block the squares that improve his position relative to his opponent. *Its usefulness is greatest* in open positions (high branching factor) with many reachable squares, that is, *earlier in the game*.
    ii. *Minimum Remaining Moves* can be regarded as a *tactical* evaluation. It attempts to estimate indirectly the game theoretic question: who can make at least one more move than their opponent against any play. The algorithm used is very inaccurate (provides a very loose lower bound) on open boards and increasingly accurate (provides a much tighter lower

bound) on closed boards (low branching factor) with fewer remaining squares. *Its usefulness is greatest later in the game*[1].

Results

The table below shows competition result s which demonstrate

- with high confidence that either custom heuristic separately is more effective than the baseline,
- with medium confidence that the combination is more effective than either separately

Player One	Player Two	Player One Wins	Player Two Wins
Iterative Deeping Alphabeta *Baseline heuristic*	Iterative Deeping Alphabeta *Minimum Moves heuristic*	37%	**63%**
Iterative Deeping Alphabeta *Baseline heuristic*	Iterative Deeping Alphabeta *Control Heuristic*	19%	**81%**
Iterative Deeping Alphabeta *Baseline heuristic*	Iterative Deeping Alphabeta *Combo heuristic*	13%	**87%**

Analysis

I speculate that my combo heuristic is more accurate than the baseline heuristic, but takes longer on average to evaluate a given position. Therefore my combo heuristic's increased accuracy compensates for its decreased search speed.

If this is true then we would hypothesise that my heuristic would perform

a) even better relative to the baseline if search speed was not relevant
b) less well if search speed was made more relevant.

The table below shows competition results which support the hypotheses

a) by limiting search depth to a single fixed ply
b) by doubling the timeout from 150 milliseconds to 300 milliseconds

Player One	Player Two	Player One Wins	Player Two Wins
Alphabeta – Single Ply search *Baseline heuristic*	Alphabeta – Single Ply search *Combo heuristic*	5%	**95%** (> 87%)
Iterative Deeping Alphabeta *300 millisecond search* *Baseline heuristic*	Iterative Deeping Alphabeta *300 millisecond search* *Minimum Moves Heuristic*	20%	**80%** (<87%)

[1] Pleasingly then, because the *control* heuristic tends to score higher the more reachable squares remain in the game, on average its weight in the combination heuristic relative to *minimum remaining moves* tends to decline as the game progresses.

The two hypotheses can also be demonstrated explicitly by instrumenting the time-cost of both heuristics and the search depth reached by the different search algorithms.

The results below were obtained by feeding each player all the moves from the same 100 randomly generated games and averaging across all the moves.

METRICS TABLE

Player	Average Completed Search Depth	Average Score Evaluation Time (microseconds)	Average Branching Factor (excluding first two moves)
Iterative Deepening Minimax *Baseline heuristic*	6.4	12	4.1
Iterative Deeping Alphabeta *Baseline heuristic*	8.3	12	4.1
Iterative Deeping Alphabeta *Combo heuristic*	7.5	29	4.1

These results show that Alphabeta node pruning enables a significant advantage in average search depth over vanilla Minimax.

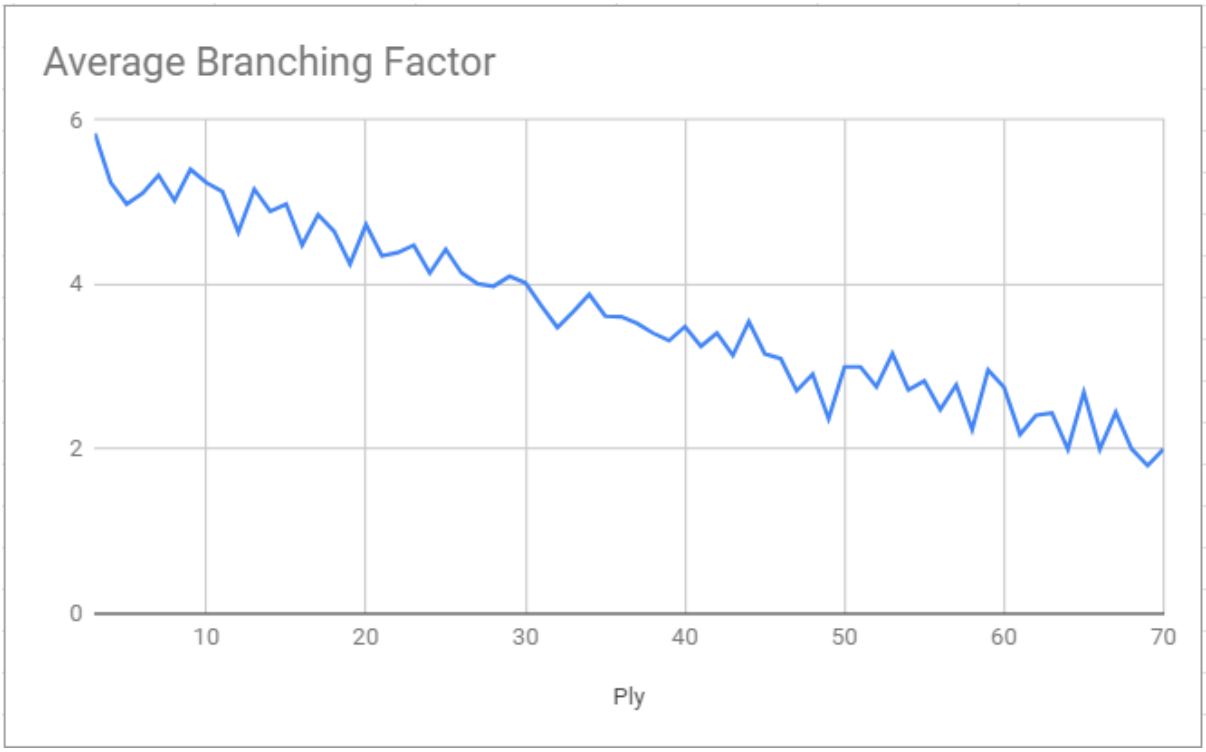
The results show that the increased average evaluation cost of the combo heuristic over the baseline heuristic means sacrificing some average search depth. The combo heuristic compensates for this with better accuracy.

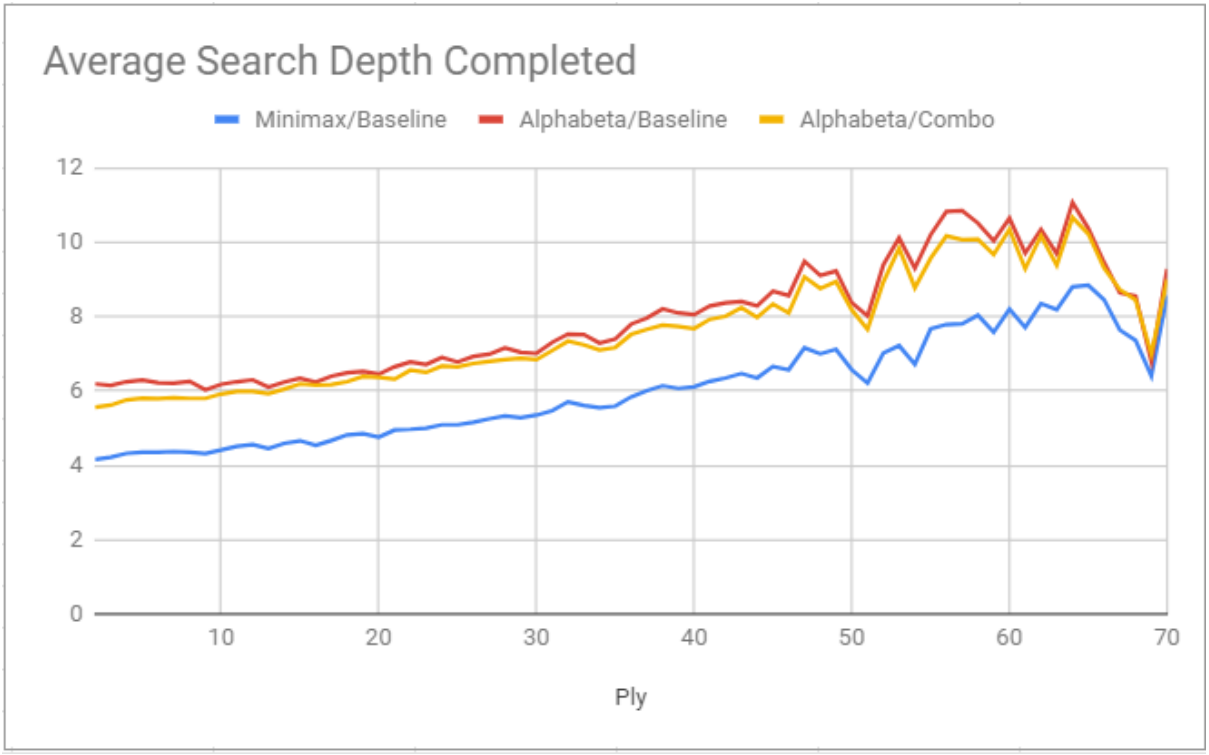
The experimentally determined average branching factor in these games was about 4, meaning that every additional ply of search depth requires evaluating on average four times as many leaf nodes as the previous ply (node pruning considerations aside). Since we would expect the score evaluation time cost to dominate the cost of traversing intermediate nodes, then --very approximately -- we would expect a heuristic that is four times faster to have around one ply advantage in search depth. In fact, the baseline is only about 2.5 times faster than the combo heuristic on average, and gains about a 0.8 average search depth advantage. This is close to what we would expect, but see Appendix One for a slightly more complex story.

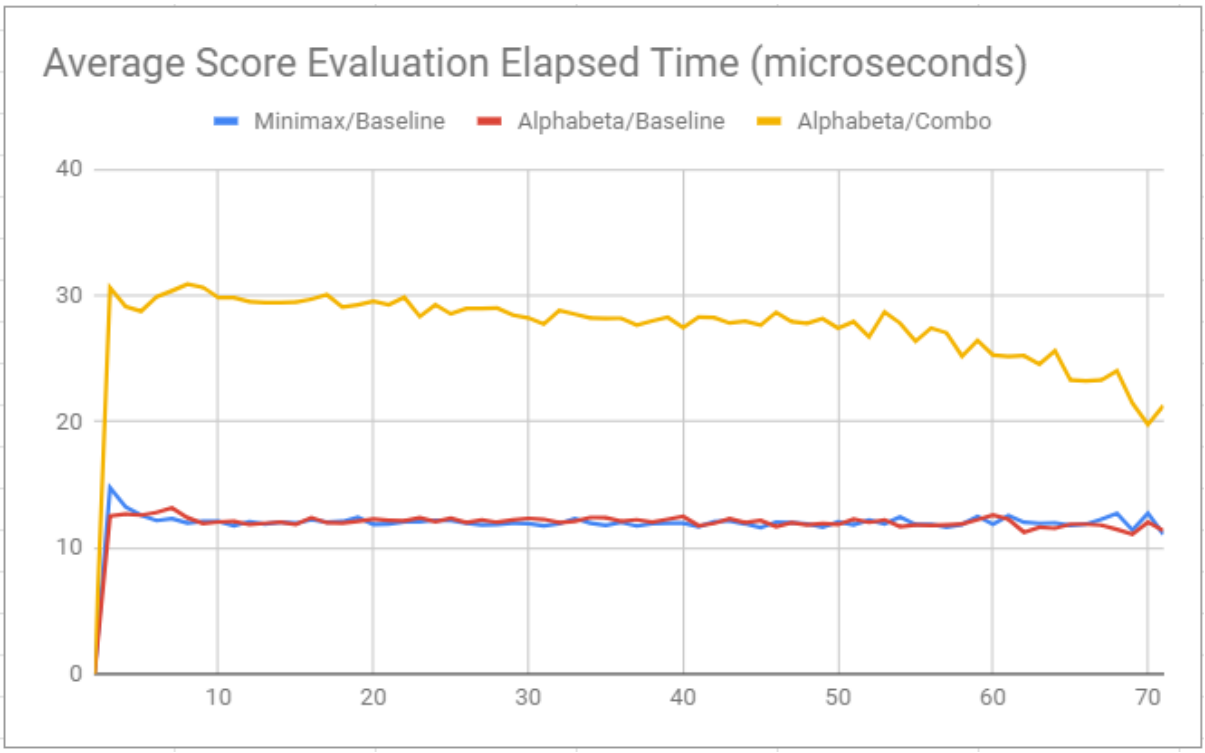
Appendix One: Metrics Broken Down by Game Ply

The graphs below show the same metrics from the Metrics Table broken down by game turn (ply).

(For instance, the first graph shows that across all 100 games the average branching factor on move 30 was about 4).







We see that as a game progresses the branching factor tends to decrease (as the position tends to become less open).

As a consequence of the decreasing branching factor, the average search depth achieved by all the players tends to increase as the game progresses. (Until around ply 65 when many games are so close to the end that they are solved by the players before the search timeout).

Alphabeta pruning confers a significant and fairly consistent search depth advantage over vanilla Minimax at every ply.

The faster baseline heuristic confers a small search depth advantage on Alphabeta/Baseline over Alphabeta/Combo.

We would expect that as the branching factor decreases the search depth advantage of the faster Alphabeta/Baseline over Alphabeta/Combo would increase. This doesn't happen. And that is explained by the pleasing fact that the combo heuristic tends to run faster the fewer reachable squares remain in the game.

Appendix Two: Sketch and Discussion of Algorithm used for the Custom Heuristic

Algorithm Description

Definition:- A player's i^{th} *wavefront* represents all the squares a player can first reach in i moves and that his opponent cannot prevent him from reaching.

The **Propagate Wavefronts** algorithm successively generates each players $i+1^{th}$ *wavefront* in turn.

Initial State:

- *Active Wavefront*: the position of the knight of the player next to move
- *Inactive Wavefront*: the position of the knight of the player second to move
- *Empty Squares*: the set of squares still empty on the game board

Procedure:

While either *Wavefront* is non-empty
(i.e. still contains at least one knight):

a) Play all valid moves for all knights in the *Active Wavefront* (a move is valid if it is one of the eight possible knight moves and it ends on one of the set of squares in *Empty Squares*)
   - The set of squares containing a knight after these moves becomes the new *Active Wavefront*

b) Remove all the squares in the (new) *Active Wavefront* from the set of *Empty Squares*

c) Repeat a) and b) for the *Inactive Wavefront*

Final State:

- *Cumulative Active Wavefront*
  - All of the squares in all of the *Active Wavefronts* generated by the procedure
- *Active Wavefront Count*
  - The number of times the procedure generated a new non-empty *Active Wavefront*
- *Cumulative Inactive Wavefront*, *Inactive Wavefront Count*
  - Derived in the same way as the equivalent active versions.

Sketch Proofs of the significance of elements in the Final State

Cumulative Wavefront

Because every possible move is made at every ply (where a ply is the propagation of either wavefront), every square enters a wavefront at the earliest possible ply. Therefore, when a square enters a player's wavefront that player must be able to reach that square first.

The player's *Cumulative Wavefront* therefore contains exactly the squares a player can reach first.

The number of squares in each *Cumulative Wavefront* is the metric required by the *Relative Control* heuristic.

Wavefront Count

By construction, every square in the player's $i+1^{th}$ wavefront is reachable from some square in the player's i^{th} wavefront, and no square in his $i+1^{th}$ wavefront can have been reached by his opponent before the player's $i+1^{th}$ move.

Therefore, if a player can reach any square in his i^{th} wavefront by some combination of i moves, he can reach any square in his $i+1^{th}$ wavefront by some combination of i+1 moves.

The induction is grounded by noting that the player's initial position represents his 0^{th} wavefront, reached after 0 moves.

Therefore, if the procedure generates N non-empty wavefronts, the player can reach any square in the Nth wavefront by some combination of N moves. Therefore , the player can make at least N moves, regardless of what moves his opponent plays.

The *Wavefront Count* is the metric required by the *Minimum Remaining Moves* heuristic[2].

Time Complexity

Given a game board of size NxM, all of the knight moves for a set of knights (i.e. a one-step propagation of a wavefront) can be accomplished in constant time using a bitboard implementation. (See the `generate_all_knight_moves` code).

Therefore every iteration of the procedure is constant time.

Since every iteration removes at least one reachable square from the Empty Squares set and the algorithm stops when there are no more reachable squares (a subset of the Empty Squares), the worst case time complexity is linear in the number of Empty Squares (given a board of size NxM).

Weaknesses

The heuristic tends to generate a much better score for the active player than the inactive player (having the first wavefront is a significant advantage) and so the heuristic score tends to oscillate between two different values at each alternate depth of search. The score is also highly dependent on the number of reachable squares remaining, and so the heuristic tends to average lower scores at greater search depths.

This means that although the heuristic is great for basic Iterative Deepening Alphabeta, it is not suitable for search algorithms that compare positions at different depths of search. (e.g. quiescent search or following a forcing sequence).

Potential Extensions

The propagate wavefronts algorithm is capable of providing more information than used in the combo heuristic

Endgame Detection

The combination of the two Cumulative Wavefronts is exactly the squares still reachable in the game.[3] Endgame detection could be used for instance to transition to endgame databases, or to solve nearly complete games with proof number search.

Partition Detection

The algorithm can be trivially extended to detect partitions (simply verify that each newly generated player wavefront does not intersect with his opponent's current cumulative wavefront).
Partition reduces the complexity of solving the position to (merely!) the simpler of two NP-hard problems: which player has the shorter longest path in its graph of remaining moves. In most games in practice, by the time a game reaches partition the longest path of at least one player can probably be solved relatively quickly[4].

---

[3] Because if a square is reachable first by some player then it is reachable, and if a square is reachable then it is reachable first by one player or the other

[4] And note one particularly simple special case: After partition a player's cumulative wavefront is exactly his reachable squares, and the number of squares in the cumulative wavefront is therefore an upper bound on his maximum possible remaining moves. If this equals the wavefront count (a lower bound on his minimum remaining moves) then the length of his longest path is already known.