I have to say this is quite a fun project. I did enjoy being thrown into a new codebase head first, and figuring out what actually mattered.
The simulation results are summarized in the table below. To get at least some statistical significance, I ran 400 simulations (for all the different strategies evaluated, that took just about a night to run); using the formula for the standard deviation of the binomial distribution, sqrt(p*(1-p)), I get an approximate standard deviation of the average percentage win over the 400*7 (random + 3x MM + 3x AB) simulations is approximately sqrt(0.8*0.2)/sqrt(400*7) = 0.75%, actually a bit more as its distribution is a mixture of binomials with different p's for the 7 test strategies; so if the difference between ID_Improved and my strategy is more than 2.5%, we can definitely say there's an improvement.

I began with a trivial variation mentioned in the lecture, namely #own_moves minus 2 * #opponent_moves. After profiling that, I saw that a lot of time was spent in game.get_legal_moves() - it's not that slow but it gets called a lot - so I optimized it by caching all legal knight moves on that board at agent initialization time, so that I only needed to look these up and call move_is_legal() to check that square hasn't been used yet. That is called Faster Improved in the table, and did not produce a statistically significant improvement.

The next logical step is to look ahead more than one move, for example two moves. As speed is still paramount, I produced two versions of that heuristic: one that corrected for the fact that you could arrive at the same position as a sequence of two different moves, and one that didn't (called 'Improved, two steps naive' and 'Improved, two steps exact' in the table, corresponding to functions improved_score_fast_x2_naive and improved_score_fast_x2. These produced a very substantial improvement (from 78.6% for ID_improved to about 82% for both these strategies), but with the difference in performance between the two variants being insignificant.

After that, I thought it would be good to try something more interesting. I generalized the partition idea briefly mentioned in the lecture. To wit, the new heuristic looks at the board at each step, and sees whether the area of the board that each player can reach (with an unlimited number of moves) is the same for both players (clearly these two areas are either identical or non-overlapping). If these areas are non-overlapping, the board is considered 'partitioned' - in this case, the moves of the other player are no longer relevant. To save processing time, I pretend that if a board is partitioned, and my partition has N squares, I have N moves left (I'm sure it's possible to construct counterexamples, but I'm assuming they're not frequent enough to matter).

That leads to the following heuristic: if the CURRENT board state is not partitioned yet, it values all possible partitioned future states as -inf if the opponent's partition is larger, and inf otherwise - that is to save time further evaluating a node if we already found one partition there (inf for maximizing_player, -inf for not maximizing_player, that is).
Once the current board state is partitioned, we switch to the naive heuristic from above; in fact since we can't influence the opponent anymore, we just need to count our own moves. As there's no other calculation overhead at this time, we calculate our own moves 2 steps ahead, approximately.

We evaluated two versions of that: one using the simple (one-step look-ahead) Improved score for non-partitioned nodes, another using the naive two-step look-ahead Improved score.

The results are somewhat disappointing: while including the partitioning logic does raise the score to 80.4% when using the naive improved heuristic, it only raises the score to 81.9% for looking two steps ahead, that is pretty much to exactly the same value as looking two steps ahead with no partitioning logic at all. I would say that happens because the extra time we spend on the partitioning check allows us to search less deeply on average.

In a final tweak, I looked at what happens if we try caching search results during iterative deepening. The idea is that while scores derived from heuristics will change as depth of tree increases, the sure wins (with me being active) and sure losses (with the opponent being active) don't depend on heuristic values and so can be reused in the next turn. That is implemented in game_agent_3 (with one-step look ahead)

Somewhat to my surprise, that version performed just about the same than the one without caching, at just under 80%, so the overhead of maintaining the cache outweighs the time savings of using it.

Based on the above, I would say that neither partitioning nor caching is worth recommending; and would go with the 'Improved, two steps naive' function, that is looking at both my own and the opponent's moves two steps ahead, without correcting for being able to reach same destination in different ways, on the basis that simpler is better.

On a final note, that function (two steps ahead naive) can actually be interpreted as a naive three-layer convolutional neural network, with the convolution in a cell happening not over its neighbors on the board, but over the cells that can be legally reached from it, with all the weights being ones, and after two layers of that we subtract the value of the cell with the current opponent location from the value of the cell with my current location. I am rather curious what would happen if I tried to optimize that network using RL, but will for now submit this as is ;)

Opponent\ strategy	ID_Improved (benchmark)	Faster improved	Improved, two steps naïve	Improved, two steps exact	Improved + partitioning	Improved two steps ahead naive + partitioning	Improved + partitioning + caching
Random	93.3%	92.0%	93.8%	95.3%	94.0%	94.5%	94.3%
MM_Null	83.8%	84.0%	90.8%	88.0%	86.8%	88.3%	88.8%
MM_Open	73.5%	74.3%	79.5%	79.0%	79.3%	81.0%	75.5%
MM_Improved	72.8%	73.8%	77.0%	74.3%	73.8%	76.5%	72.0%
AB_Null	84.3%	83.0%	87.3%	85.3%	85.8%	83.5%	85.3%
AB_Open	73.3%	75.5%	78.0%	74.3%	71.5%	78.3%	74.5%
AB_Improved	69.3%	67.8%	74.3%	74.3%	71.5%	71.5%	67.5%
Avg percent win:	**78.6%**	**78.6%**	**82.9%**	**81.5%**	**80.4%**	**81.9%**	**79.7%**

Each cell shows average percentage of wins of the column strategy against the row strategy out of 400 games played

Approximate standard deviation of the percentage in the bottom line is sqrt(0.8*0.2)/sqrt(400*7) = 0.75%