Alpha beta based Khet agents and their optimizations

Adir Zagury
Hebrew University of Jerusalem

Aviv Yaish
Hebrew University of Jerusalem

***Abstract*—Khet is a 2 player zero sum game with perfect information which is described as "Chess, only with lasers". In this work, we designed two AI agents based on the tried and true alpha beta search, both of which use heuristics (but differently), and then we tried optimizing the grading parameters for the heuristics using a genetic algorithm and a variant of stochastic hill climbing. We found that for the game of Khet, our improved alpha beta agent is better than a greedy-heuristic agent, and that the stochastic hill climbing variant is a bit better than a genetic algorithm for heuristic parameter optimization.**

I. Introduction

A. Khet's rules

Khet is a relatively new board game which is described as "Chess, only with lasers". It's rules are deceptively simple[1]: there are two players - red and silver, each with a set of figures:

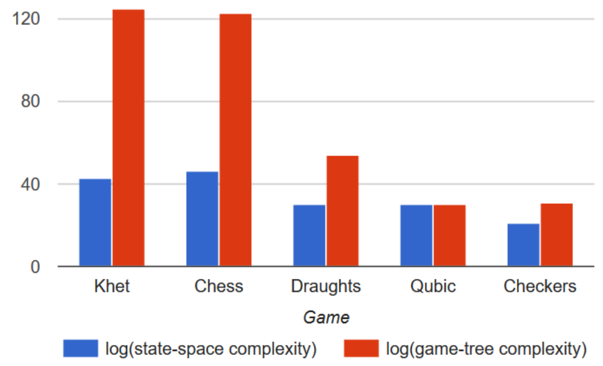
1) A sphinx. It can't move, only rotate. Has a laser that is shot at the end of the player's turn. If the shot hits a mirror on a figure, it is reflected in the according direction. If the shot hits an exposed side of the figure, that figure is captured and removed from the game board.
2) A pharaoh. If it is captured by the other player, he wins.
3) Two scarabs. They have mirrors on both sides, thus aren't captured when shot. They can displace pyramids and anubises on the game board by switching places with them.
4) Two anubises. Their front can sustain laser shots, but their other sides can't.
5) Seven pyramids. They have two exposed sides.

All figures (besides the sphinx) can either move into an unoccupied space in any direction, or rotate to any direction. The figures are placed on the game board in the starting position, and the silver player makes the starting move. If the same board arrangement appears for a third time in a row in the same game (the same pieces of the same colors occupy the same squares in the same orientations), there is a draw (also called "deadheat").

B. Why is Khet an interesting problem

1) Unexplored territory: Khet is a new (introduced to the public in the spring of 2005) and fun game that still hasn't been widely researched by AI specialists. As such, we couldn't find more than one publicly published AI agent for it (the default agent bundled with the SDK we found), and only one research paper on the subject. We saw this as an opportunity for being Khet AI pioneers.

2) Khet is hard: Khet is a complex game[2] - it's initial state space (with all figures still on the board) is 10^{43}, it's average branching factor for an alpha-beta search (of depth 4) is 68.92, meaning the game-tree complexity (how many different games can be played) is $\approx 10^{125}$. Comparing Khet's complexity to other games we get:



Meaning, at least in these parameters Khet is comparable to Chess, and that just like Chess, it's impossible for a personal computer to keep an enumeration of all possible states in memory, or for an alpha beta search agent to explore all the nodes of a game tree.

C. Project goals

In this project we sought to answer the following questions:

- Is Khet a game a computer can play well?
- Which algorithm solves it best?
- In the Khet problem world, what sort of optimization will fit best?
- If we have a set amount of time to run each optimization, should we have a larger population, or more generations?

We answered them by writing two algorithms and two optimization algorithms and run them with different parameters for population size and number of generations. Then, we pitted the resulting agents against each other and saw who won.

II. Our methods

We decided to build game tree search agents and to optimize their parameters using local search methods.

A. Our agents

As Khet is an adversarial game where one's gain is the others loss, it is only natural we use an adversary AI. We used two type of algorithms: alpha-beta search and heuristic-greedy alpha-beta search. In both, each node corresponds to a different game state where an edge between a parent and a child is the move that changed the parent state into the child state. Also, both grade the states according to a grading function which we'll cover later.

1) Alpha beta search agent: Alpha beta search is an improvement over the minimax search algorithm which prunes branches that definitely won't affect the outcome of the minimax search. As such, it returns the same result as the standard minimax search, though possibly more efficiently.

Although it goes over less branches compared to the minimax search, it still goes over many branches. Especially early in the game, when there are still many figures and thus many possible moves to go over. In our implementation, we managed to make the alpha beta search agent efficient enough to perform a depth 4 (sometimes even 5, and extremely rarely even 6, as we'll explain later) search on a modern personal computer in usually much less than 15 seconds[3]. This search agent is implemented in the class "UserBot".

We made many improvements to the alpha beta search algorithm:

a) Randomness: If the agent values all the states/moves the same, it chooses randomly from them. Thus, the agent has less chances of repeating himself in a way that might lead to deadheat and thus to a tie.

b) Time limitation: Each turn is limited to 15 seconds. If the agent takes more time than that, the game server declares its opponent as the winner. As the time each move takes varies, we had to make a decision: to decrease our depth so we will not reach the limit, or get creative: we allow it to take a little longer and cut it short when needed, taking the risk we might not go over some branches. In the absolute majority of times, it won't be cut short and the gain we get from allowing it to go to a deeper level is much greater than the occasional loss of moves. Now the alpha beta search keeps track of time, returning the best move it found so far if it notices it's dangerously close to the end of the time quota.

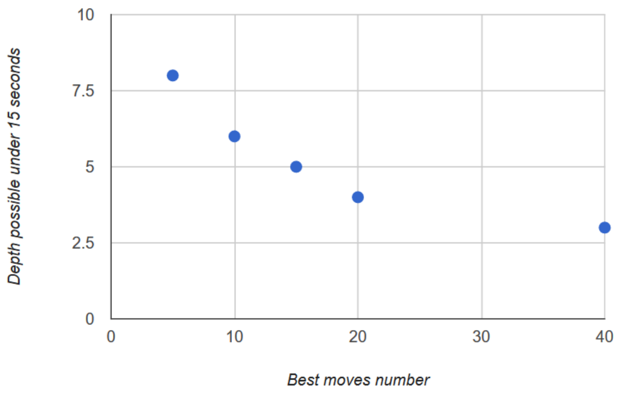
c) Variable depth: The maximum depth possible for the search to achieve is ultimately decided by the number of states it will go through and the time it needs to check each one. Later during the game, as figures get captured, there are less possible moves for each player (the branching factor decreases). As such, there are less states to go through, meaning that the agents slowly need less and less time to search until their predetermined maximal depths. Eventually, they finish their search much earlier than the end of their quota. So, it is a good idea to allow the agents to search deeper when the branching factor is small, and so we did.

d) Heuristic move sorting: States graded highly may have a better chance to reach even more highly graded states than lower graded ones. Also, trying good moves first might help end the search earlier. So, we thought it would be wise to let the agent sort the moves based on the resulting states' grades, and check the branches in order from the highest graded to the lowest. We decided to perform a comparison between a regular alpha beta search and such a proposed heuristic search.

Incorporating the sorting into the alpha beta search resulted in the search taking much more time - apparently the sorting overhead is very high, and almost always the depth 4 agent took much more than 15 seconds, even after code optimizations. Only reducing the depth to 3 allowed the agent to finish the search before the end of the quota, which means that the agent can't foresee the possible results of each possible move as well as the regular agent, meaning - it is simply worse. We decide to leave the "regular" alpha beta agent without this improvement, and to fix this we made further changes, resulting in a new search agent.

2) Heuristic-greedy alpha beta search agent: So, the overhead of sorting all the possible moves and going over **all** of them is so high that only reducing the search depth allows the agent to finish the search in a reasonable time. But, if looking first at the moves that give the best graded states is supposed to be better, is there even a reason to look at the "worst" moves? Or, will ignoring moves could make the agent miss moves that might turn out to be good only deeper down their branch?

We decided to check this by creating a "heuristic-greedy" alpha beta search - we keep only the N best moves according to their resulting states grades, and let the search recursively check only them. Doing so and keeping N small (keeping the maximum branching factor small) allowed us to increase the search depth possible under 15 seconds:



Note that for $N \geq 40$ the possible depth is ≤ 3, while that for $N \leq 15$ the possible depth is ≥ 5.

This search agent is implemented in the class "HeuristicBot".

3) Unused improvements: There were several improvements that we have tried that seemed promising, but, for various reasons we decided not to implement them:

a) Time based variable depth: Ideally, we would like to make the agents divide their allocated time between all the branches equally and to allow them search deeper if they have enough remaining time.

However, because the alpha-beta may cut down many possible branches we can't divide the time equally - it is highly dependent on the number of branches cut down. The result of trying to use the optimization was that either we sometimes let the agents search much too deeply (causing them to take much more time than their quota) or much too shallow (reaching a depth of less than 4). Trying to measure the saved time resulted in using too many system calls that cost even more time.

b) Incremental move generation: When using the SDK's function for obtaining the possible moves, it generates them all at the same time. But, there is no need to do this - it is better to generate them incrementally: if we stumbled upon a very good move (for example, capturing the enemy's Pharaoh), there's no reason for looking at the next moves, thus there's no reason to even generate them in the first place.

We decided against implementing this as it would have required making major changes to the SDK's code.

B. Grading states - our heuristics

In order for our search agents to eventually reach a good conclusion, we need a good grading function that will receive a state and output a numerical representation of how good/bad we believe the state to be.

We thought of a few parameters for grading the states:

- the obvious - if the agent captured his opponent's Pharaoh he wins, so this is always valued at positive infinity. If the agent's Pharaoh is captured by the opponent, the agent loses - so this is always valued at negative infinity.
- how many figures of each type both the agent and his rival have.
- how many possible moves both the agent and his opponent have - it's good to limit your opponent, but it might be bad to limit yourself.
- how many figures surround both the agents' and his opponents' Pharaohs. The right figures in the right orientation around the Pharaoh can protect it against lasers.

1) You've got the power: But, how much better is having 4 Pyramids when compared to 2? How much better is having 3 figures protecting our Pharaoh when compared to 4? To take that into account it might be good to use a root or a square of the grade, or even different powers.

2) Defensive vs. offensive strategies: If this is a zero sum game, why even consider different grades for our possible moves and for our opponents'? Why not for a given parameter, take the grade with a plus sign when considering it for ourselves, and a minus sign for considering it for our opponent?

But, if for example we give a higher score for the number of Pyramids our opponent has compared to the score we give for our Pyramid count, we're encouraging the agent to play offensively. So, there is value in differentially grading the same parameter when it comes to ourselves and our opponent.

3) Heuristics we didn't use:

a) Manhattan distance of enemy laser: We thought about measuring the Manhattan distance of the enemy's laser at it's closest point from our figures, but the computation turned out to be too expensive. Then, we tried reducing it to only the laser's Manhattan distance from our Pharaoh, but that too turned out to be expensive and not beneficial - if the Pharaoh is well protected, the closeness of the laser has no meaning. Also, sometimes the enemy laser can be very far at a certain time, but then the enemy moves just one figure and that enables him to capture our Pharaoh.

b) Minimum steps needed to capture enemy Pharaoh: We though about counting the number of steps needed to capture the enemy Pharaoh while ignoring any of his figures currently on the board. Again, the computational cost of such a heuristic isn't worth the results that it gave - calculating such a thing requires performing a "short" search on the game board. But, the time used for such a search is better spent on letting the alpha beta search delve deeper into the game tree.

C. Optimizing the grading parameters

We represented the parameters as vectors with weights given to each one, and a power to raise the grade by. But, how much weight should be given to each parameter? Which power for each parameter should we use? To decide this we use optimization algorithms, which are meta-heuristics for finding and approximating the global optimum for a given function, in our case - the grading function.

It could be considered as a well known AI problem: maximizing the value of a function that receives a vector. We assume that though multidimensional, we are in somewhat flat space - small changes in the vector corresponds to small changes in the outcome, and generally the outcome of the function on average of two point is close (more or less) than the average of the function of those points ($f(\frac{v_1+v_2}{2}) \approx \frac{f(v_1)+f(v_2)}{2}$). Thus it is logical to assume that stochastic hill climbing and genetic search are good candidates, assumptions we will check in our work. Each type of optimization has its advantages and disadvantages, in the form of run time and results regarding the function outcome world.

1) Stochastic-steepest-ascent hill-climbing: The algorithm we use is not a classic stochastic hill climbing algorithm, rather a variant of it of our own devise which combines stochastic hill climbing, steepest ascent hill climbing, and a bit of simulated annealing. Rather than starting in a random "location" (grade vector), as is the usual for hill climbing, it is given a choice of a few random starting locations, from them it chooses the best according to the grading function (which simply pits the various grade vectors against each other using an agent).

Then, after choosing a starting place, it selects a number of random neighbors for it. Here comes the bit of annealing: at the beginning, the neighbors are allowed to be a bit far from the current location, and as more steps are taken only closer neighbors are looked upon. After the random neighbors are generated, like in steepest ascent hill climbing, the algorithm picks the best neighbor.

These steps are repeated over and over until the maximum number of steps is achieved.

We like to call this algorithm stochastic-steepest-ascent annihill-climbing.

a) Improvements: As you can see, the first improvement is that the algorithm is given a choice as to where it wants to start the hill climbing. This hopefully will allow it to wisely pick a "place" closer to the global maxima. But, even if it isn't close, the "annealing" of randomly generating neighbors that might be far will assure us that the algorithm won't be stuck to a local maxima (at least at the beginning). As we studied, looking at successors uniformly at random is known to be complete. Choosing among the random neighbors the best one will hopefully make the search arrive at the maxima faster then it would've otherwise.

2) Genetic algorithm: The genetic algorithm starts with a random population of grade vectors, and pits them against each other using an agent. Those who won the most matches are given larger probabilities for breeding, and when breeding those who won more get a higher probability to pass more of their genes (grades) to the offspring. The offspring has a certain chance to mutate randomly.

This process is repeated for as many generations as wanted.

a) Improvement: A slight modification we've done is that as time goes on, the mutation can change the grade values less and less. We did this so that at the beginning, the grades could mutate "wildly" at the beginning, to make sure the algorithm will "look around" its state space and won't be confined to a small area of it.

III. Code

A. The SDK

We found an SDK^4 for the game which was developed by a company called EPAM located in Brest, Belarus. It was used as part of an AI contest held last year to find the best AI agent for Khet. Unfortunately, the contest ended, so we couldn't test our agent against the other agents submitted.

1) Changes we made to the original code: The SDK as was downloaded wasn't a perfect fit for the agents we wished to program, so we had to change it a bit.

a) States: The original SDK made many unnecessary passes on the game board to calculate certain things, or couldn't calculate them in the first place. For example, it needed more than two passes on the game board to calculate the possible moves for both the current player and its rival. Also, the SDK didn't have any function to get the number of figures around any of the players Pharaohs. We created a getState() function that calculated all this information in one pass and returned a State class that included all the necessary information.

b) Board: As the SDK is a server-client affair, the game board originally was updated only by the server, and sent to the client during it's next turn. Even the Board class' built in board.makeMove(move) function didn't update the board. In order for our agents to test their moves we needed to add a function called board.makeMoveNew(move) that actually updated the board.

c) IBot: IBot is the parent of all the bots/agents that are to be used with the SDK. We expanded it with many functions that we used both with our agents.

d) Server side changes: For some reason, the client doesn't close when a match ends. Also, a log of the wins is never saved anywhere. In order for our optimization algorithms to run many games and understand which parameters are better than others, we needed to implement both automatic closing of the game client when a game ends and keeping a log of who wins a match.

B. Our classes

Besides programming UserBot and HeuristicBot, we also programmed the class OptimizationModifier which contains the code both for the genetic algorithm and the stochastic hill climbing. Also, it contains code to pit the resulting grades/agents against each other.

IV. Data

The parameters we decided to use for each optimization algorithm, and the fact that each agent uses on average a few seconds for each move means that running a single optimization on a single agent takes an upward of 12 hours, even when letting the generations/steps and children/neighbors be relatively small numbers.

In order to get as many data points as we could in the time we had, we ran many concurrent optimizations on different computers. Fun fact: at one time we used 8 different computers at the aquarium, plus our two laptops! Unfortunately, even with all this computing power, we didn't get nearly as much data as we wanted, and if we had more processing power or computers, we believe we could've obtained much more definitive results.

Table 1.

White\Red	A(3,12)R	A(4,6)R	A(3,12)H	A(4,6)H	G(3,12)R
A(3,12)R	(2,0,2)	(2,1,1)	(4,0,0)	(3,1,0)	(3,0,1)
A(4,6)R	(2,1,1)	(3,1,0)	(4,0,0)	(4,0,0)	(2,1,1)
A(3,12)H	(0,0,4)	(0,1,3)	(4,0,0)	(2,2,0)	(0,0,4)
A(4,6)H	(0,1,3)	(0,0,4)	(1,3,0)	(3,1,0)	(1,1,2)
G(3,12)R	(3,0,1)	(2,0,2)	(4,0,0)	(4,0,0)	(1,2,1)
G(4,6)R	(1,2,1)	(0,3,1)	(4,0,0)	(2,2,0)	(1,2,1)
G(3,12)H	(0,1,3)	(0,2,2)	(1,3,0)	(0,4,0)	(0,0,4)
G(4,6)H	(0,1,3)	(0,2,2)	(3,1,0)	(1,3,0)	(0,3,1)
P-R	(0,1,1)	(0,2,0)	(2,0,0)	(2,0,0)	(0,2,0)
P-H	(0,0,2)	(1,1,0)	(2,0,0)	(2,0,0)	(0,0,2)

White\Red	G(4,6)R	G(3,12)H	G(4,6)H	P-R	P-H	Total:
A(3,12)R	(2,0,2)	(2,2,0)	(4,0,0)	(0,0,2)	(2,0,0)	(24,4,8)
A(4,6)R	(2,1,1)	(3,1,0)	(4,0,0)	(1,1,0)	(2,0,0)	(27,6,3)
A(3,12)H	(0,0,4)	(3,1,0)	(1,3,0)	(0,0,2)	(2,0,0)	(12,7,17)
A(4,6)H	(0,1,3)	(2,1,1)	(2,2,0)	(0,0,2)	(1,1,0)	(10,11,15)
G(3,12)R	(3,0,1)	(4,0,0)	(4,0,0)	(1,1,0)	(1,1,0)	(27,4,5)
G(4,6)R	(4,0,0)	(4,0,0)	(3,1,0)	(1,0,1)	(1,1,0)	(19,11,6)
G(3,12)H	(0,2,2)	(2,2,0)	(0,4,0)	(0,1,1)	(0,2,0)	(3,21,12)
G(4,6)H	(0,3,1)	(3,1,0)	(0,4,0)	(0,2,0)	(1,1,0)	(8,21,7)
P-R	(1,1,0)	(2,0,0)	(2,0,0)	(0,1,0)	(1,0,0)	(10,7,1)
P-H	(0,0,2)	(1,1,0)	(2,0,0)	(0,0,1)	(0,1,0)	(8,3,7)

As we ran each optimization twice with the same parameters, each row/col in the table is a summation of both agents with the same parameters.

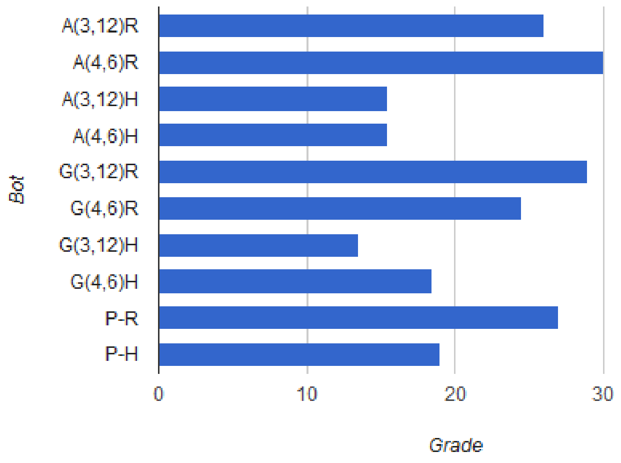
Lets define:

- Grading Type:
  A = stochastic hill climbing ('A' for **A**dir**A**viv)
  G = genetic
  P = grades we hand tuned ourselves
- (x,y) = (kids in generation, number of generation)
- Algorithm:

R = regular alpha-beta
H = heuristic-greedy alpha-beta

- (a,b,c) = (row's wins, ties, row's losses)

Table 1 includes the results of letting the different agents use the various grading vectors outputted by the optimization algorithms. Lets define a scoring method to asses the different agents: each agent gets 1 point for each win, $\frac{1}{2}$ a point for a draw and 0 for loosing. Lets look at a graph of the agents and their scores:



Note: the grades of P-R and P-H are extrapolated to match the number of games they played compared to the other agents.

Even though the A(4,6)R agents played exactly twice as many games as the P-R agent, they won almost 3 times as much, had one tie less, but one loss more. Also, looking at the results file, the best agent is indeed one of the A(4,6)R agents - he has a score of 10.5, and he has the following grades vector:

Parameter	Value
myPossibleMoves	[11.243, 0.045]
opponentPossibleMoves	[5.337, 0.321]
myPharaohDefense	[19.867, 0.437]
enemyPharaohDefense	[3.742, 0.57]
enemyNum	[41.552, 121.553]
alliesNum	[50.599, 1.087]
powerEnemy	[0.17, 0.242]
powerAllies	[1.270, 2.456]

The grade files for both $A\left(4,6\right)R$ agents can be found in the folder: AI/results/A46R, and the full test log can be found at AI/results/testLog.txt.

V. Conclusions

Analyzing the results of the matches between all of the agents, we had reached the following conclusions, as far as they are concerned to the world of Khet:

a) Regular alpha beta vs. heuristic-greedy alpha beta: As a rule, the regular alpha-beta agents are much better than the heuristic-greedy alpha beta agents. There can be many reasons for that - the heuristic-greedy agents don't even bother looking at certain moves that might not look good at the start, but eventually might turn out to be good. Another option is that the heuristics used don't grade states well enough.

b) Stochastic hill climbing vs. genetic algorithms: The top two agents "classes" were $A\left(4,6\right)R$ and $G\left(3,12\right)R$. The first was generated by the stochastic hill climbing algorithm, the latter by the genetic algorithm. Their ranking is very close, both won the same number of games (but the genetic had less ties, and also 2 more losses) and when pitted against each other both win and lose the same number of games, so it's hard to tell which is better. Probably only longer runs will allow us to clearly find the strengths and weaknesses of each in the case of using the regular alpha beta agent.

Their game performance is comparable, so lets try to compare the runtime the optimization algorithms needed to calculate these grades - as we can see, the runtime of $A\left(4,6\right)R$ and $G\left(3,12\right)R$ should be identical: both play $\binom{4}{2}\cdot 2\cdot 6 = 72 = \binom{3}{2}\cdot 2\cdot 12$ games using the same alpha beta agent. As we can see, given the same runtime, but different parameters - the algorithms perform roughly the same.

c) Population vs. generation tradeoff: Running both optimization algorithms is very time consuming, where enlarging any one of the population or generation can increase the run times by a lot. So, given we have a limited run time, where would we better invest in? At the beginning of the paper we asked the same question. In order to answer it, we tried to test the optimization algorithms on parameters that would take approximately the same time to finish (as we saw in the previous subsection).

For the heuristic-greedy algorithm it seems to be better to invest in a larger population rather than in more generations when optimizing it using hill climbing - although $A(3,12)H$ loses two more games than $A(4,6)H$, he also wins two more games and doesn't tie as much as $A(4,6)H$. For the genetic algorithm it is not so clear as to what is better, probably as a result of the small data set. The performance of both optimization algorithms here is very variable - both produce agents that lose more than they win (although the genetic produced one that won one more match than he lost). Even the best heuristic agent (which coincidentally was our hand optimized one) doesn't match the worst regular alpha beta agent. Again, our grading function might be to blame here. Maybe a better heuristic would've helped.

d) Strategy: The agents that won the most games are both from the A(4,6)R "class", and they both played a decidedly more offensive strategy - preferring much more to capture their opponent's figures than protecting their own. Also, both put more emphasis on making moves that increase their possible moves number than making moves that decrease their opponent's. As such, it might be of interest for further research in the field of Khet AI agents.

e) Luck: The first player has an advantage - in our comparison, when an agent is the starting player, usually he will win more than he will lose. Like Joseph Bertin wrote in 1735 in his *The Noble Game of Chess* - "He that plays first, is understood to have the attack." So, in a sense - being lucky is important for Khet AI agents.

f) So, to sum it all up: is Khet a game a computer can play well?: Yes, a computer can play Khet well. Every time we saw a match between two of our agents, we were amazed at the complexity of their moves. Sometimes they even sacrificed their own figures for future gains! It would be interesting to test in future works whether a sacrificing agent is better than a “no friendly fire” agent.

ACKNOWLEDGMENTS

The authors would like to thank Professor Michael Larson, Del Segura and Luke Hooper for inventing the game Khet. We would also like to thank HAL 9000, may he rest in peace, for giving us the inspiration to enter the AI world.

REFERENCES

[1] Khet 2.0 Instructions: http://goo.gl/XBb8mG

[2] J.A.M. Nijssen, *Using intelligent search techniques to play the game Khet*, 2009. Link: https://goo.gl/wXvwtx Note: the version of the game used for these calculations is older and a bit different, but not too different: some figures have been renamed, two figures have been removed, and the “stacking” of figures have been removed. Using a similar computation to Nijssen we are left with a state space of at least:

$$\binom{70}{1}\cdot 4^7\binom{69}{7}\cdot 2^2\cdot\binom{62}{2}\binom{60}{2}\times\binom{58}{1}\cdot 4^7\binom{57}{7}\cdot 2^2\cdot\binom{50}{2}\binom{48}{2}\approx 2.3\cdot 10^{43}$$

Based on our runs, the average branching factor and game length are approximately the same, so the game tree complexity should be too.

[3] According to the only published academic work that we found on the subject of Khet AI agents (Nijssen’s thesis), no-one managed to achieve more than a depth 4 alpha beta search on modern personal computers

[4] www.aichallenge.rocks