AI Search: Assignment 2

P. van Heerden Z. Mohamed

28 September 2018

Contents

1 Overview of implemented functionality

1.1 Domains

As per the project specification we have implemented domains for both the (m, n, k)-game, and the Digits of Pi example from Knuth and Moore's seminal paper[1].

1.2 Algorithms

As per the project specification, we have implemented the Negamax[2] algorithm (in addition to its branch-and-bound and $\alpha\beta$ variants), as well as an iterative deepening variant of the Negamax-with-$\alpha\beta$ algorithm that makes use of a transposition table for early cut-offs as well as move ordering.

Algorithm	Variant	Implementation	Agent
Negamax	Vanilla	`Negamax#F0`	`NegamaxAgent`
	Branch-and-bound	`Negamax#F1`	
	$\alpha\beta$	`Negamax#F2`	
	ID-$\alpha\beta$ + TT	`Negamax#F3`	`NegaDeepAgent`
Negascout	Vanilla	`Negascout#Negascout`	`NegascoutAgent`
	Vanilla + TT		
Random	-	-	`RandomAgent`
Human	-	-	`HumanAgent`

Figure 1: Taxonomy of the algorithms included in the engine, with reference to the concrete implementations and corresponding agents.

In addition to the original specification, we have also implemented the Negascout algorithm. A variant of the Negascout implementation that makes use of the transposition table for early cut-offs and move ordering has also been implemented.

1.3 Evaluation function

We implement a naive evaluation function. This evaluation function is employed by all the algorithms in this iteration of the engine. The evaluation function favours multiple, potentially shorter chains as opposed to one long chain. This increases the resiliency of the structure, and provides "a way out" in the case of a long chain being capped by the opponent.

1.4 Utilities

As per the project specification, we have implemented the required infrastructure for Zobrist hashing[3] and transposition tables. The transposition table employs the TWODEEP replacement scheme[4].

1.5 Additional features

In addition to the features outlined by the project specification and the above sections, we also implement two novel agents, and an extension to the engine that allows users to provide the initial configuration of a (m, n, k)-game as a pair of Java Properties files.

The first agent—`HumanAgent`—allows users to play against any of the available agents, prompting the user for input via standard input. Two instances of `HumanAgent` can also be used to allow for human-to-human gameplay. The second agent—`RandomAgent`—provides a means by which we can fuzz[1] the domain implementation. This agent features a deterministic mode of execution in order to provide counter-examples in the case of a bug exposure.

[1]Fuzzing in this context refers to a means by which we can to maximize the test coverage of an API in order to expose bugs.

2 Architecture of the engine

2.1 Domains

Domains act as a layer of abstraction between the search algorithm implementations and the true nature of the domain. It allows the algorithms to generate successors from a node, query whether the node is terminal or not, query a node for its value, and to apply and undo moves to the underlying domain implementation.

The `Domain` class is an abstract class that defines the interface that domains must comply to. All new domains must extend this class.

2.2 Agents

Agents act as bookkeepers for the duration of the game. An agent is primarily responsible for providing a move to the referee when it is the agent's turn-to-play. The agent maintains a reference to the active domain, and is responsible for applying moves made by the opponent as they are communicated via the referee. After a game has been played to completion the agent can be queried for a statistical report. The content of the report is specified by the agent, as to include relevant statistics that might not be applicable to other agents[2].

The `Agent` class is an abstract class that defines the interface that agents must comply to. All new agents must extend this class.

2.3 Referees

Referees act as the intermediary between agents. Since agents are not in any way aware of each other, the referee is responsible for passing the moves elected by the agents between each other. At the end of the game the referee is also responsible for determining the winner and prompting the agents to display statistics.

The `Referee` class is a concrete class that has a single constructor and takes two objects that belong to the `Agent` class hierarchy. It exposes a single method, `runGame`, that starts the game between the two agents.

2.4 Hashing

Zobrist hashing functionality is implemented in the `Zobrist` class. The length of a Zobrist hash is 64 bits, and this is not configurable at present.

2.5 Transposition table

Transposition table functionality is implemented in the `TranspositionTable` class. The default constructor sets the capacity of the table to $2^9 = 512$ entries. A second constructor allows the user to supply a capacity by passing an integer n. The size of the table is set to be 2^n.

[2] In the case of `NegaDeepAgent`, for example, more information is provided about the cut-offs enabled by the transposition table.

```
file           ::= board_settings? depth seed utilties strategy
strategy       ::= "Strategy" "=" strategy_name
strategy_name  ::= "Negamax-F1"
                 | "Negamax-F2"
                 | "Negamax-F3"
                 | "Negascout"
                 | "Random"
depth          ::= NUM?
seed           ::= NUM?
board_settings ::= "m" eq_num "n" eq_num "k" eq_num
utilties       ::= use_tt? use_id?
use_tt         ::= "TT" "=" ("true"|"false")
use_id         ::= "ID" "=" ("true"|"false")
eq_num         ::= "=" NUM
```

Figure 2: Grammar of the configuration file.

2.6 Configuration files

As stated in Section 1.5, users can supply a configuration file describing the algorithms to be used and utilities to be enabled, as well as the size of the board. The input grammar for the configuration file is listed in Figure 2. Each key-value pair should appear on a new line. Nine sample configuration files are provided in the repository[3].

3 Experiments

3.1 Digits of Pi from Knuth and Moore

We implemented a new domain to verify the pruning results from Figures 1, 2, and 3 in Knuth and Moore's experiment[1]. The terminal nodes in this domain are able to sample a digit of Pi based on the node's position in the total order. For this experiment, move ordering kept as naive, left-to-right.

Results

We were successfully able to recreate Knuth and Moore's results. Three JUnit tests are provided in order to support our claims[4]. These can be invoked by running `./gradlew test --tests TestDoPAgent` in the root of the project directory. Invocation of these tests prints out the variant of the Negamax algorithm being used, as well as the digits and the indices of the digits that have been sampled.

[3] `https://github.com/PhillipVH/adversarial-search`

[4] The tests each contain an assertion that will fail if the number of digits sampled by the algorithm is different than the expected number from the Knuth and Moore experiments.

Remarks

Initially we were unable to reproduce the results. This was due to a programming error where the initial value of a bound was `Integer.MIN_VALUE`, and the bound was then passed to a recursive call as `-bound`. This caused a silent integer overflow, since `-Integer.MIN_VALUE == Integer.MAX_VALUE + 1`[5]. We fixed this by replacing the minimum value assignment with `Integer.MIN_VALUE + 1`.

3.2 Effects of the transposition table on search efficiency

We compare the number of nodes explored during a full game by the implemented algorithms and their variants, on boards with a specified initial configuration. The random agent was employed as the adversary in all of the experiments, operating deterministically with a seed value of 42. The transposition tables were given a capacity of $2^{17} = 131072$ entries.

We run three sets of tests, increasing the size of the board for each new set while keeping k constant.

Results

Algorithm	**Variant**	**Explored Nodes**
Negamax	$\alpha\beta$ + TT	14091
Negascout	Vanilla + TT	54011
	Vanilla	231064
Negamax	$\alpha\beta$	244533
	Branch-and-bound	323388
	ID$\alpha\beta$ + TT	809082
	Vanilla	7822215
	ID$\alpha\beta$	11891917

Figure 3: Number of nodes explored during a $(3, 3, 3)$-game, searching to a depth of 10.

Algorithm	**Variant**	**Explored Nodes**
Negamax	$\alpha\beta$ + TT	164979
	ID$\alpha\beta$ + TT	826432
Negascout	Vanilla + TT	4636629
Negamax	$\alpha\beta$	6969244
Negascout	Vanilla	6620019
Negamax	Branch-and-bound	8611281
	ID$\alpha\beta$	23508380
	Vanilla	526967886

Figure 4: Number of nodes explored during a $(5, 5, 3)$-game, searching to a depth of 4.

Algorithm	Variant	Explored Nodes
Negamax	$\alpha\beta$ + TT	385635
	ID$\alpha\beta$ + TT	1087087
Negascout	Vanilla + TT	23884055
Negamax	$\alpha\beta$	28366889
	Branch-and-bound	28366889
Negascout	Vanilla	28522059
Negamax	ID$\alpha\beta$	73324879
	Vanilla	987592590

Figure 5: Number of nodes explored during a $(7, 7, 3)$-game, searching to a depth of 3.

Remarks

As expected, due to the fact that it does not prune, vanilla Negamax explores the most nodes by a considerable margin (with the exception of the first set). With regards to the effectiveness of the transposition table, it follows from the results that it does contribute significantly. In Figures 4 and 5, the three algorithm variants that explored the least number of nodes were all employing a transposition table. The exception is, once again, the first set.

We suspect that the relatively small nature of the configuration used for the first set of samples hinders the effectiveness of the transposition table. This might be an indication that the transposition table requires a period to improve the relative quality of its entries.

References

[1] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370275900193

[2] A. A. Elnaggar, M. Abdel, M. Gadallah, and H. El-Deeb, "A Comparative Study of Game Tree Searching Methods," *International Journal of Advanced Computer Science and Applications*, vol. 5, no. 5, pp. 68–77, 2014. [Online]. Available: http://thesai.org/Publications/ViewPaper?Volume=5&Issue=5&Code=IJACSA&SerialNo=10

[3] A. L. Zobrist, "A new hashing method with application for game playing," The University of Wisconsin, Tech. Rep. 88, 4 1970.

[4] D. M. Breuker, J. W. H. M. Uiterwijk, and H. J. v. d. Herik, "Replacement schemes for transposition tables," *ICCA Journal*, vol. 17, no. 4, pp. 183–193, 1994.

[5] "Integer (Java Platform SE 7)." [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html