Generating Game Strategies using Graph Transformations

Rick Hindriks
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
h.n.hindriks@student.utwente.nl

ABSTRACT

Given a completely observable game of which states are modelled as graphs, which is changed using graph transformation rules. We present a system which allows the Minimax algorithm to be applied to the game. The system supports a flexible approach for modelling the heuristic function used by the Minimax algorithm. The usability and effectiveness of the system is tested by generating strategies for a defined range of games.

Keywords

Graph modelling, Graph transformation, GROOVE, Minimax, Games, Strategy, Heuristic

1. INTRODUCTION

Games are regularly subjected to research by computer scientists and mathematicians. Chess is a famous example. Many algorithms have been developed for this particular game, up to the point where nowadays computers outperform human players [1].

The Minimax algorithm and its variants have been used with success as means to construct a computer controlled player [1]. Given a state of the game, the algorithm aims to maximise the minimum gain of the player by minimizing the maximum gain of the opponent.

The definition of gain and loss is abstract, and needs to be defined for each game. In the Minimax algorithm, this is defined by using a heuristic function. This heuristic function determines a numerical value based on a given arbitrary state of the game. This value serves as an indicator for the expected chances of winning the game from that state.

The algorithm assumes that the opponent will always play the best possible move. The output of the algorithm provides a choice between possible moves for any state of the game. As such, the output is effectively a strategy for the game that the algorithm analysed.

The states which occur during a game, the *state space*, can be represented by a graph. In this graph, the nodes of the graph correspond to the states of the game, and edges correspond to the moves from a certain state to a different

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
21st Twente Student Conference on IT June 23, 2014, Enschede, The Netherlands.
Copyright 2014, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

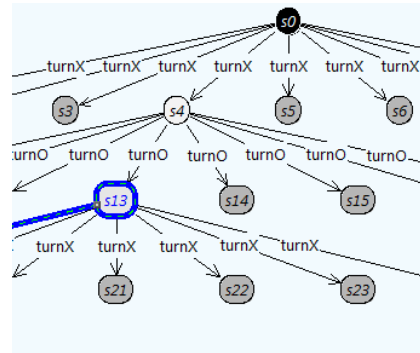

Figure 1. Part of the state space of the Tic-Tac-Toe game. The edge between s0 and s4 is a particular move, in which an 'X' is placed on the board. From s4 the player with 'O' is able to choose between a number of possible moves.

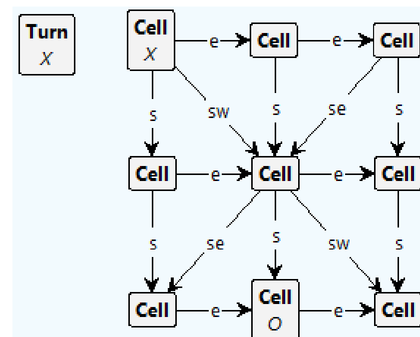

Figure 2. A state of the game of Tic-Tac-Toe modelled as a graph.

state (see Figure 1).

The game states themselves can also be modelled as graphs (see Figure 2), and the moves as transformations of this graph. The state space graph is used as the input of the Minimax algorithm.

The tool GROOVE [6] has been developed since 2004 and can be used to construct the aforementioned graphs and graph transformations. In GROOVE, a state is represented by labelled nodes and edges (as shown in figure 2). A graph transformation is defined using transformation rules, which alter the state graph. GROOVE can subsequently use a state exploration strategy to explore the state space given a starting graph and the graph transformation rules.

This research presents a design of a system which combines the flexibility of graph modelling in GROOVE with

the Minimax algorithm. This system should provide a convenient method for specifying the heuristic required by the Minimax algorithm. The new system generates a strategy for a game modelled as a graph transformation system by using the Minimax algorithm in conjunction with the specified heuristic function. The flexibility of this system is verified by generating strategies for a selected range of games.

2. BACKGROUND

2.1 Graph modelling with GROOVE

2.1.1 State graphs

As mentioned above, the tool GROOVE is used to create a system of graphs and graph transformation rules. A state graph models a logical structure, consequently the visual position of the nodes is not of importance. The node labels in GROOVE are also used for typing (see Figure 2). GROOVE also supports special node types which contain data such as integers and strings. The initial state of the game is modelled as the starting graph.

2.1.2 Graph transformation rules

State graphs are transformed to different state graphs using graph transformations. In GROOVE graph transformations are defined by means of graph transformation rules.

A graph transformation rule is also defined as a graph with labelled nodes and edges. GROOVE defines special labels which are added before the label of the node or edge to alter a graph (see Table 1). A transformation rule can be applied to the graph when it is matched. GROOVE applies and tests whether a rule matches the current graph using the following procedure:

1. Test whether all *reader* nodes and edges exist in the graph
2. Test whether all *eraser* nodes and edges exist in the graph
3. Test whether all *embargo* nodes and edges do not exist in the graph

When all tests succeed, the rule is applicable; its application results in the following steps:

4. Remove all *eraser* nodes and edges from the graph
5. Add all *creator* nodes and edges to the graph

When the rule is applied, a new state is generated and added to the state space. For an example transformation rule and its effects, see Figures 3 and 4. A more complex transformation rule is shown in Figure 8.

2.1.3 State exploration

In any state, zero or more transformation rules can be applied. The application of a single rule alters the existing state and creates a new state. The choice which rule to apply to which state is made according to a state exploration strategy. Examples of such strategies are: breadth-first, depth-first, or random exploration.

2.2 Minimax

The Minimax algorithm is an algorithm based on the Minimax theorem, "...which states that every finite, zero-sum, two-person game has optimal mixed strategies. " [9] This was proven by John von Neumann [3]. A finite, zero-sum,

Table 1. Main label types in graph transformation rules

Name	Appearance	Description
Reader	Black	Checks whether a node or edge exists in the graph
Eraser	Dashed blue	Checks whether a node or edge exists in the graph, and removes it when the rule is applied
Creator	Bold green	Creates a new node or edge when the rule is applied
Embargo	Striped red	Checks whether a node or edge does not exist in the graph

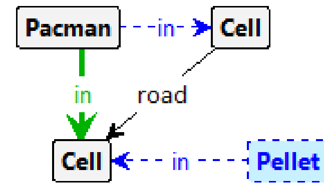

Figure 3. The transformation rule used to move *Pacman* to an adjacent *cell*, and eat (destroy) a *pellet* residing there. Note that an application of this rule is only valid when a *pellet* resides in the target *cell*.

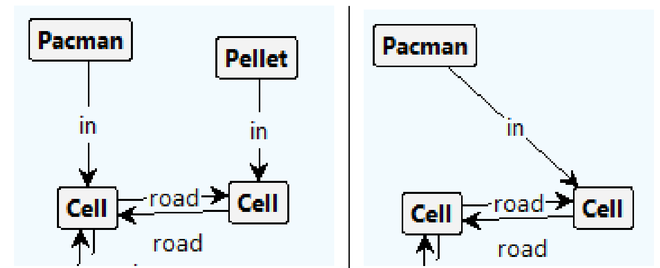

Figure 4. A situation in the game of Pacman. The situation on the left reflects the game before applying the eat-pellet rule (see Figure 3), and the situation on the right reflects the game after the rule application.

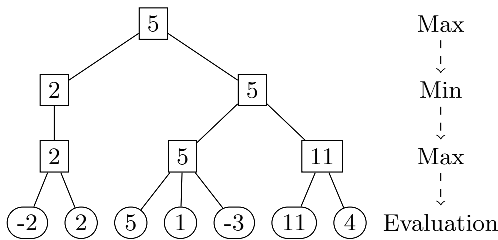

Figure 5. An example result of the Minimax algorithm. The values of parent nodes are based on the values of their child nodes.

two-person game is a game which is played by two players, which will eventually come to an end, such that the sum of the resulting scores of both players always results in the same value. An example of this is chess, in which a single point is distributed among both players (one for the winning player or half a point for both players).

We define the player, and the opponent. The Minimax algorithm determines the maximum gain of the player. In a zero-sum game, maximising the gains of a particular player results, by definition, in the minimisation of the gains of the opposing player. By assuming the opponent will always play the best possible move, the algorithm is able to predict the best or worst possible outcome of any move from a given state.

Minimax uses a heuristic (or: evaluation) function which calculates the quality of a given game state. An example of a simple (zero-sum) heuristic, given a state s, would be:

$$h(s) = \begin{cases} 1, & \textit{Player won} \\ -1, & \textit{Player lost} \\ 0, & \textit{Otherwise} \end{cases}$$

When the state space is not fully explored, it may not be possible to determine whether a move will eventually win the game. A different heuristic function which does not depend on the eventual outcome can be used in these cases. An example for Reversi (see section 5.1) might be the following heuristic function:

$$h(s) = (\textit{White pieces}) - (\textit{Black pieces})$$

The algorithm operates by performing a depth-first search (unlimited, or up to a given limit), to determine the heuristic values of states which lie deeper in the game tree. When the algorithm reaches a final state, or stops exploring from a certain state (because of configured constraints on the exploration, such as a limit as mentioned above), the heuristic value of that state is evaluated using the heuristic function. We will refer to such states as evaluation states.

Depending on which mode the algorithm is in (minimisation or maximisation) the value of a parent node is calculated by selecting the minimum or maximum value of its child nodes (see Figure 5). The minimisation or maximisation mode is switched if a different player can make a move, and the process is repeated until the algorithm has returned to the starting node.

The output is the evaluated tree with the calculated values. These values effectively form a strategy based on the heuristic function, as the algorithm has calculated which move is the best move in every situation.

3. RELATED WORK

This research uses the GROOVE program [6], which is also used to model non-game systems. Different tools based on model transformations exist, but no graph transformation tools similar to GROOVE have been found.

The Minimax algorithm is known to be used in practice as a means to construct computer controlled players, Deep Blue used a variant of the Minimax algorithm [1]. However, no specific scientific implementations of the Minimax algorithm for specific games were found by searching with combinations of the keywords *Minimax*, *Player*, *Implementation* and *Game(s)*.

Specific implementations of the Minimax algorithm have been designed for areas other than games. Jiang et al. have constructed a speech recognition algorithm by using the Minimax algorithm [2].

4. EXPLORATION STRATEGY DESIGN

4.1 Overview

To implement the Minimax algorithm, the exploration strategy performs a depth-first search on the state space graph while generating the Minimax value for each encountered evaluation state. When the exploration of the state space is finished, or has reached the specified maximum depth, the Minimax value of all states in the state space graph has been calculated.

The following parameters of the exploration strategy can be configured:

- A list of transformation rules which carry the values from the evaluation states.
- The (optional) maximum search depth, this limits how deep the state space graph is explored.
- A turn evaluation rule, which is used to determine which player is the active player in the explored state. This rule must be applicable in every state.
- Whether the algorithm should attempt to maximise or to minimise the maximum gains.

4.2 Implementation

The Minimax algorithm is implemented as an exploration strategy for GROOVE. This enables the algorithm to control the order in which states are explored and evaluated. The algorithm explores the state space by performing a depth-first search, as this minimises memory usage.

The interface provided by GROOVE disallows adding information to the state space graph. As a consequence, the exploration strategy has to maintain its own datastructure to keep track of intermediate data such as intermediate values and dependencies of nodes (see Figure 5 for a visualisation of example data). The algorithm uses a tree (see section A.1) structure to maintain this intermediate data, and evaluates values stored in this tree to determine the final node values. The designed implementation calculates this value after the exploration is finished.

Graph transformation rules in GROOVE are able to export values when they are applied. This feature is used by a heuristic evaluation rule to calculate the heurstic value and store it in the exploration strategy. The feature is also used to determine which player is allowed to make a move in a given state. This information is used to determine whether the algorithm has to maximise or minimise the gains for that state.

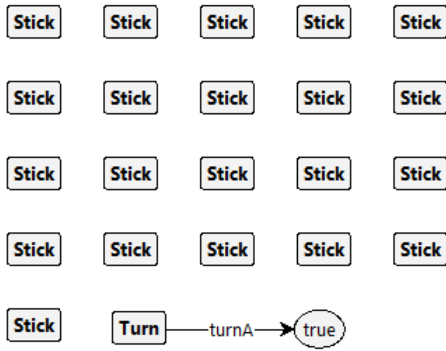

Figure 6. The starting state of the game of *21*.

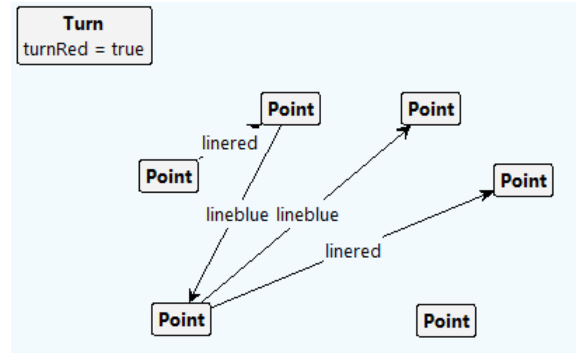

Figure 7. An intermediate state of the *Sim(6)* game.

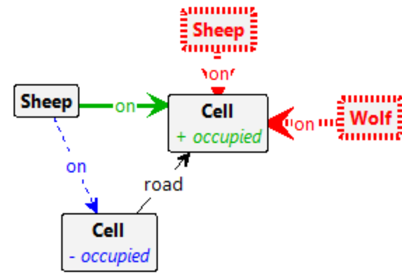

Figure 8. In this transformation rule, a sheep moves between cells connected by a *road* edge. Note the embargo rules which specify that the destination *cell* should not harbour any wolves or sheep.

The exploration strategy is able to limit the search depth as a configurable setting. The algorithm updates whenever a transformation rule (a transition between states) is evaluated:

1. Allocate a tree node for the destination state of the transition
2. Whenever a rule should be evaluated, evaluate it and update the tree:
   - A heuristic evaluation rule is examined, and its value is stored in the corresponding tree node.
   - A turn evalution rule which determines which player is allowed to make a move is evaluated, and its value is stored in the corresponding tree node.

5. VALIDATION

5.1 Validation Games

The flexibility and correctness of the new system is validated by generating strategies for a range of games (see Table 2). The selected games vary in game tree depth and branching factor.

To verify whether the system supports the development of heuristics for small games, the games of *Tic-Tac-Toe* and *21* have been selected. To verify whether the system supports heuristics which function with a limited exploration depth, the games of *Sim(6)*, *4x4 Reversi*, and *Wolf and Sheep* have been selected.

- The game of *Tic-Tac-Toe* has a relatively small state space, which is useful for testing purposes. An example state of *Tic-Tac-Toe* is shown in Figure 1.
- The game of *21* is a variant of the *Nim* game, in which two players draw one, two, or three sticks from a stack of *21* sticks after each other. The player who draws the last stick loses the game. The game can also be played with less sticks, which decreases the state space complexity.
- The game of *Sim(6)* [7] consists of six vertices in which two players draw edges between vertices one after another. The first player who creates a cycle of length 3 (a triangle if drawn on paper) loses the game. This game has a limited amount of options each turn, but has a moderately large state space. An example state of *Sim(6)* is displayed in Figure 7.
- *Reversi* (also known as *Othello*) is a game in which players take turns to place colored marbles on a square board. Marbles of the opponent can be converted by bounding them in a straight line with two marbles of the player. The version which is played on a 4x4 board is used to limit the size of state space.
- The game of *Wolf and Sheep* is a game which is played on a Draughts board (50 diagonally aligned board positions). Five sheep (white discs) start facing one wolf (black disc) at the opposite sides of the board. The sheep can only move forward, the wolf can move in all diagonal directions. The sheep win when the wolf cannot make a move, the wolf wins when it reaches the opposite side of the board. This game has a limited amount of moves each turn, but may need many moves to complete. An example transformation rule for moving sheep in this game is shown in Figure **??**

5.2 Evaluation

5.2.1 Usability

Given an existing graph transformation system for a game, a certain amount of effort is required to prepare the graph

Table 2. An overview of the testing games

Name	# moves each turn	State space size
Tic-Tac-Toe	Small (max 9)	Small
21	Small (max 3)	Small
4x4 Reversi	Medium	Medium
Sim(6)	Medium (max 30)	Large
Wolf and Sheep	Small (max 10)	Large

Table 3. An overview of the graph transformation systems of the games

Game	# States	# Transitions	Finished
Tic-Tac-Toe	6436	21645	Yes
21	44	89505	Yes
Sim(6)	3668k+	8241k+	No
4x4 Reversi	1144k+	3735k+	No
Wolf and Sheep	2485k+	6294k+	No

transformation system for use with the Minimax exploration strategy. The following steps are needed to perform a conversion:

- The turn evaluation rule must be created, and the grammar must be altered to enable this rule to read the turn information from a state.
- The heuristic evaluation function must be constructed: for each possible evaluation state, a rule needs to exist to enable the exploration strategy to determine the value of the heuristic function in such a state.

5.2.2 *Effectiveness*

We constructed graph transformation systems for every game mentioned in section 5.1. For an overview of the results for each game, see Table 3. This section describes the process of constructing and testing each game. The tests were run with GROOVE with a memory limit of 3GB (Java Maximum Heap limit). The exploration was halted as soon as the duration of the exploration exceeded 15 minutes, this is indicated in the table in the '*Finished*' column, the values in the table correspond with the measured values at this point.

- For the game of Tic-Tac-Toe, using the first evaluation function described in section 2.2, the algorithm generated a tree with correct values for a regular game with the maximizing player making the first move. The algorithm was able to explore the entire game tree. The result of the exploration strategy showed that when all players play with a perfect strategy, the game can only end in a draw.
- The game of 21 does not allow games which end in a draw. This was reflected in the used evaluation function by removing the corresponding alternative. This resulted in the following evaluation function:

  $$h(s) = \begin{cases} 1, & \textit{Player won} \\ -1, & \textit{Player lost} \end{cases}$$

  The results of the exploration strategy showed that the starting player can always win. The algorithm was able to explore the entire game tree.
- No problems were encountered during the construction of *Sim(6)*. The size of the state space of the game limits the amount of states which can be evaluated in a short time with simple heuristic functions, similar to the function of *21*. Not all states could be explored with the used memory limit.
- No significant problems were encountered while constructing *4x4 Reversi* in GROOVE. The heuristic function used to evaluate the states of this game did not depend on the eventual outcome of the game, but rather estimated the winning chances with the difference in piece counts of both players. Not all states could be explored with the used memory limit.

  $$h(s) = (\textit{White pieces}) - (\textit{Black pieces})$$
- The implementation of the game of Wolf and Sheep did not cause any notable errors. The evaluation function used was similar to the function used for the game of *21*. Not all states could be explored with the used memory limit.

6. FUTURE RESEARCH

The Minimax algorithm has been used in the wild to construct computer controlled players. As a result, many optimisations of the algorithm exist. These variants of the Minimax algorithm avoid exploring the entire state space graph. Probably the best known optimisation of the Minimax algorithm is $\alpha - \beta$ pruning. Other optimisations are the NegaScout [5], SSS* [8] and MTD(f) [4] algorithms. The current implementation of Minimax has not been optimized. Future research could aim to improve the performance of the current implementation by using one of the aforementioned optimisations.

Graph transformation tools such as GROOVE currently lack interactivity during the exploration. This limits the feasibility of assessing the quality of the generated strategies. For example, playing a game of chess against a computer controlled player which uses a strategy generated by a graph transformation system using GROOVE is cumbersome at the very least. Future research could attempt to improve the interactiveness of graph transformation tools.

We are able to apply algorithms which use heuristic functions to games represented as graph transformations, different search algorithms (A^* for example) might be suitable for searching the game state space. Future research could employ the designed framework for heuristic functions to test whether different algorithms are suited for use with graph transformation systems.

Existing game models must currently be transformed for use by the exploration strategy by hand. It might be possible to automate (parts of) this process. Future research could aim to construct a system which achieves this.

7. CONCLUSION

The designed system shows that the Minimax algorithm is compatible with graph transformation systems such as GROOVE. The system requires only small adjustments of existing graph transformation systems before use, and can consequently be used for a wide variety of problems.

The new system can be used as an effective means to generate strategies for a wide variety of zero-sum games (those which can be modelled as a graph transformation system) with little effort. The system accomplishes this by combining the flexibility of graph transformation system design of GROOVE with the power of the Minimax algorithm.

For less complex games (*Tic-Tac-Toe* and *21*), correct strategies have been generated using the designed system. The correctness of generated strategies could not be evaluated for more complex games (*Reversi* and *Sim(6)*)) as this proof requires the analysis of the complete state space.

8. REFERENCES

[1] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu. Deep blue. *Artificial intelligence*, 134(1):57–83, 2002.

[2] H. Jiang, K. Hirose, and Q. Hue. A minimax search algorithm for robust continuous speech recognition. *Speech and Audio Processing, IEEE Transactions on*, 8(6):688–694, 2000.

[3] J. Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[4] A. Plaat, J. Schaeffer, W. Pijls, and A. d. Bruin. A new paradigm for minimax search. Technical report, Erasmus School of Economics (ESE), 1995.

[5] A. Reinefeld. An improvement of the scout tree-search algorithm. *ICCA Journal*, 6(4):4–14, 1983.

[6] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[7] W. Slany. Graph ramsey games. *CoRR*, cs.CC/9911004, 1999.

[8] G. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179 – 196, 1979.

[9] E. W. Weisstein. Minimax theorem. `http://mathworld.wolfram.com/MinimaxTheorem.html`.

APPENDIX

A. IMPLEMENTATION DETAILS

A.1 Internal Tree datastructure

The used internal tree datastructure is implemented as a Java class. The class functions either as a leaf node, containing a value, or as a parent node. The class contains both behaviours and can be transformed to either type of node. Each node maintains a boolean variable (which is set by the exploration strategy) to determine whether the heuristic value of its children should be maximised or minimised.

The heuristic value of a node is recursively calculated. Whenever the heuristic value is requested from a parent node, the node calculates a minimum or maximum value from the values of its children. When the heuristic value is requested from a leaf node, the heuristic value contained in that node is returned.